

Vikash Kumar

AI-Assisted Hardware Verification

Cognitive Verification Architecture with
the VEGA Framework

 Springer

Vikash Kumar

AI-Assisted Hardware Verification

Cognitive Verification Architecture with the VEGA Framework



Vikash Kumar
Leander, TX, USA

ISBN 978-3-032-34545-5 e-ISBN 978-3-032-34546-2

<https://doi.org/10.1007/978-3-032-34546-2>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2026

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Unless otherwise expressly agreed by the Publisher in writing, the Publisher does not authorize use or reproduction of all or any part of this work in any manner for the purpose of training, fine-tuning, evaluating, or developing artificial intelligence models, technologies, or systems. In accordance with Article 4(3) of the EU Directive 2019/790 on copyright and related rights in the Digital Single Market, the Publisher expressly reserves all rights with respect to this work under the text and data mining exception of the directive.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG

The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Every device you use, the phone in your pocket, the server handling your data, the chip accelerating your AI workload, exists because somewhere, a team of engineers spent years turning a specification into silicon. Not lines of code that can be patched after shipping. Not a prototype that can be revised next sprint. Silicon. Once it is manufactured, the logic is fixed. A bug that survives to tape-out ships to millions of devices, costs tens of millions of dollars to correct, and in safety-critical systems can cause real harm. The stakes in hardware engineering are permanent in a way that software rarely is. Modern SoCs contain billions of transistors, dozens of independently developed IP blocks, and protocol interfaces that must operate correctly under conditions no single engineer can fully enumerate. The complexity of what gets built today would have been unimaginable a decade ago. And yet the fundamental challenge has not changed: before silicon can be trusted, it must be verified. Verification has always been the hardest part of building silicon. Not because the tools are insufficient, and not because engineers lack skill. It is hard because verification is fundamentally a reasoning problem, and reasoning does not scale the way compute does. Across more than fifteen years of verification work at companies building some of the world's most complex semiconductor systems, I have watched the same pattern repeat: projects begin with confidence, and somewhere between the first regression run and silicon tape-out, verification becomes the dominant schedule risk. Debug cycles stretch. Assumptions that seemed obvious at specification time turn out to have been invisible to the testbench. Intent gets lost in the distance between the architecture document and the assertion that fires at four in the morning. This book is my attempt to address that pattern directly.

Who This Book Is For

This book is written for anyone who works in or is entering the field of hardware verification and wants to understand how modern AI tools can be applied with discipline and structure. That includes the verification engineer who already knows how to run a simulation and is ready to think about whether the simulation is actually verifying anything. The student learning UVM and SystemVerilog for the first time, who wants to build good habits before bad ones form. The experienced engineer who has signed off on programs and wondered, afterward, whether the argument could have been made more honestly. The verification lead who wants to introduce AI-assisted flows to their team without losing control of quality or traceability. And the architect who needs to explain to management why structured verification costs more upfront and saves more downstream. **VEGA**—the Verification Engineering through Guided AI framework—is the practical implementation of Cognitive Verification Architecture. It is a structured, intent-centric verification framework built around a single organizing principle: that every verification decision should be traceable to explicit design intent and that modern tooling—including AI—can be used to preserve and operationalize that intent across the entire verification lifecycle. VEGA does not replace verification engineers; it amplifies what experienced engineers know how to do, and makes that knowledge explicit enough to be governed, reviewed, and reused. The framework presented here emerged from real verification programs. The worked examples in this book are drawn from actual design verification challenges, including the Educational Bridge Model, an educational AHB-to-APB bridge specified in Appendix A and used as the canonical worked example throughout the chapters. The prompts, templates, and quality gates are not theoretical constructs—they are instruments I have refined through repeated application to protocol-level and subsystem-level verification at industrial scale.

What This Book Covers

This book presents the VEGA framework in eight technical chapters, each corresponding to a distinct phase of the verification lifecycle:

- Chapter [1](#) introduces the VEGA framework, its foundational principles, and the industry context that motivates it.
- Chapter [2](#) covers Design-Intent Understanding and XTP synthesis: extracting behavioral requirements from specifications and converting them into a structured, reviewable verification plan.
- Chapter [3](#) addresses testbench architecture and environment development through a manifest-driven approach that enforces an explicit boundary between infrastructure and test logic.
- Chapter [4](#) presents structured test scenario design and implementation, including the scenario contract that governs AI-assisted generation.
- Chapter [5](#) covers structured debug, failure analysis, and regression management, grounded in classification before investigation.
- Chapter [6](#) extends VEGA to subsystem verification through intent composition and the Subsystem Intent Graph.
- Chapter [7](#) introduces quality metrics for AI-assisted verification, organized into three families that measure the generation quality, specification coverage, and process trustworthiness of AI-generated testbenches.
- Chapter [8](#) covers coverage analysis and verification sign-off as an intent-driven reasoning process, culminating in the sign-off argument.

How to Use This Book

Each chapter is self-contained but builds on the artifacts produced in earlier chapters. Practitioners applying VEGA to an active verification program will find the most value by working through the chapters in order.

The Method First, Then the Tool

This book teaches a methodology, not a catalog of prompts. Each chapter develops the idea, the governed flow, and the judgment the Verification Strategist exercises at each gate and only then introduces the VEGA tool that puts the flow into practice. The order is deliberate: you should understand what a stage decides, and why its gate exists, before you let a tool run it. The VEGA toolset is one governed tool per stage:

- `vega_xtp_gen.py` turns a specification into a signed-off Executable TestPlan (Chap. [2](#)).
- `vega_llm_tbgen.py` turns the design's interfaces and verification intent into a compiling, simulating UVM testbench (Chap. [3](#)).
- `vega_test_generator` turns each test plan row into a committed, traceable scenario (Chap. [4](#)).

Each tool drives a generative AI, the Claude models and Claude Code for the agentic compile-and-fix loop, under the Strategist's governance. The instructions that steer the AI are encoded inside the tools, so you do not paste prompts by hand. You supply the specification and the governed inputs, the tool produces a draft artifact, and you review, refine, and approve it at the gate. The AI generates. The Verification Strategist governs.

How Much to Automate

The methodology is the same whether you walk it slowly or run it fast. What changes is how much you review at once.

- **Gate by gate.** Stop at each step, inspect the draft, and approve before the next. This is the right mode for a new domain, a specification that leans on tables and timing diagrams a model may misread, a safety-critical design, or any case where a missed feature is expensive.
- **End to end.** Let the tool run a stage in a single pass and review the final artifact once. This is the right mode for a familiar protocol, a

well-structured specification, and prototyping or iterating on a known design.

Both modes implement Cognitive Verification Architecture, and neither removes the Strategist from the loop. They differ only in where the review happens. The choice depends on team maturity, tooling investment, and the cost of a missed bug.

The Companion Repository

The VEGA toolset and the complete bridge worked example live in the companion repository at https://github.com/vksabnima/vega_framework. For step-by-step instructions, including exact commands, expected outputs, and troubleshooting for each chapter, see the `README.md` at the root of the `vega_framework` folder.

A Note on AI and Accountability

VEGA treats AI as a governed accelerator, not an autonomous decision-maker. Every AI-generated artifact in this framework is subject to explicit human review before it influences downstream verification decisions. The Verification Strategist—the human role at the center of VEGA—retains full accountability for scope, risk, and sign-off. This is not a limitation of current AI capability. It is a principled architectural choice. Verification is an engineering discipline. AI is a tool. This book is about the discipline. Verification will continue to grow in complexity as chiplet integration, heterogeneous architectures, and AI-driven design push the boundaries of what silicon can do. My hope is that VEGA provides a durable framework for meeting that complexity with structured reasoning rather than heroic effort.

Vikash Kumar
Leander, TX, USA

Competing Interests The author has no competing interests to declare that are relevant to the content of this manuscript.

Contents

[1 Introduction: The Evolution of Hardware Verification and the VEGA Framework](#)

[1.1 The Skill Worth Having](#)

[1.2 Why Verification Is the Hard Part](#)

[1.3 Verification Is a Reasoning Problem, Not a Typing Problem](#)

[1.4 The Verification Gap](#)

[1.5 The VEGA Framework](#)

[1.6 Why We Start with Protocols](#)

[1.7 From One Block to a Subsystem](#)

[1.8 From Verification Engineer to Verification Strategist](#)

[1.9 How This Book Is Organized](#)

[2 Design-Intent Understanding and TestPlan Synthesis](#)

[2.1 The TestPlan Is the Verification Strategy](#)

[2.2 The XTP Generation Flow](#)

[2.3 Step 1: Feature Extraction and Clarification](#)

[2.4 Step 2: Feature List Review](#)

[2.5 Step 3: Cross-Feature Interactions and the Test Plan](#)

[2.6 Step 4: Generate and Validate the XTP](#)

[2.7 XTP Reference: One Test Case in Full](#)

[2.8 Running the Whole Flow with the VEGA Tool](#)

[2.9 Conclusion](#)

[3 Testbench Architecture and Environment Development](#)

[3.1 From a Plan to a Bench That Runs](#)

[3.2 The Core Principle: Infrastructure and Test Logic Are Separate](#)

[3.3 The VEGA LLM Testbench Generator](#)

[3.4 Setup](#)

[3.4.1 The Generator: Python and an API Key](#)

[3.4.2 The Fixer: Node and Claude Code](#)

[3.4.3 The Simulator and Git](#)

[3.5 The Four Inputs](#)

[3.5.1 IP-XACT XML](#)

[3.5.2 Manifest.json](#)

[3.5.3 Verification intent.txt](#)

[3.6 Running the Generator](#)

[3.6.1 Project Directory Layout](#)

[3.6.2 Preview Before Spending API Credits](#)

[3.6.3 Generate the Testbench](#)

[3.7 Worked Example: The Educational Bridge Model](#)

[3.7.1 What the Generator Produced](#)

[3.7.2 Fix Session: From Scaffolding to TEST PASSED](#)

[3.8 Two Tests, Two Purposes](#)

[3.9 The CLAUDE.md Context File](#)

[3.10 Claude Code in Practice: Fixing, Recording, and Waveforms](#)

[3.10.1 Three Operating Modes](#)

[3.10.2 What Claude Code Already Knows](#)

[3.10.3 Git as the Experiment Record](#)

[3.10.4 Viewing Waveforms](#)

[3.11 Quality Gate: Scenario-Ready Checklist](#)

[3.12 Conclusion](#)

[4 Test Scenario Design and Implementation](#)

[4.1 A Simulation Is Not a Test](#)

[4.2 Why Tests Exist: Four Purposes](#)

[4.2.1 Find Bugs the Designer Did Not Anticipate](#)

[4.2.2 Confirm Traceability to the Specification](#)

[4.2.3 Build the Regression Net](#)

[4.2.4 Give Management Confidence to Tape Out](#)

[4.3 AI Augmentation: From Writing Tests to Reviewing Them](#)

[4.4 The XTP as the Test Contract](#)

[4.5 Stimulus: What It Means to Drive a Design](#)

[4.5.1 The Four Scenario Classes](#)

[4.5.2 Stimulus Quality: Decision Points](#)

[4.6 Randomization: Exploration Versus Confirmation](#)

[4.6.1 Directed Tests](#)

[4.6.2 Constrained-Random Tests](#)

[4.6.3 The Over-Constraint Failure Mode](#)

[4.7 Anatomy of a UVM Test](#)

[4.8 What AI Gets Wrong in Test Generation](#)

[4.9 Scenario Readiness: Checks Before Regression](#)

[4.10 Generating the Suite with vega test generator](#)

[4.10.1 The Generator](#)

[4.10.2 From an XTP Row to a Scenario](#)

[4.11 Result: Generating the Bridge Suite](#)

[4.12 Conclusion](#)

[5 Debug, Failure Analysis, and Regression Management](#)

[5.1 The Problem Regression Creates](#)

[5.1.1 Where AI Accelerates Debug](#)

[5.2 The Governing Principle: Classify Before You Investigate](#)

[5.2.1 Flaky Failures Are Infrastructure Failures Until Proven Otherwise](#)

[5.3 Log First, Waveform Second](#)

[5.3.1 The First Failure Is the Only Failure](#)

[5.3.2 Assertion-First Triage](#)

[5.4 The Expected/Observed/Deviation Triple](#)

[5.4.1 Transaction-Level Before Signal-Level](#)

[5.5 The Bridge Debug Session as a Classification Exercise](#)

[5.6 From Failure to Minimal Reproduction](#)

[5.7 Anchoring the Failure to Intent](#)

[5.7.1 The XTP Lookup](#)

[5.7.2 Intent Deviation Classification](#)

[5.8 Debugging Timeouts: The Liveness Deviation](#)

[5.9 Regression Triage: Order Matters](#)

[5.10 Fix, Verify, Update](#)

[5.11 Debug as XTP Validation](#)

[5.12 Quality Gate: What Regression Clean Means](#)

[5.13 Conclusion](#)

[6 Subsystem Verification Through Intent Composition](#)

[6.1 The Bridge Passed. What Can Still Fail?](#)

[6.2 Where Subsystem Bugs Live](#)

[6.3 The Ten Categories of Subsystem Intent](#)

[6.4 Integration Bug Patterns That IP-Level Verification Misses](#)

[6.5 The Subsystem Intent Graph](#)

[6.5.1 Building the SIG: A Four-Step Process](#)

[6.5.2 SIG Construction Inputs](#)

[6.5.3 Building the SIG: The Assumption Harvest](#)

[6.5.4 SIG Structure: Nodes and Edges](#)

[6.6 From Intent to Assertions](#)

[6.7 Extending the Manifest to Subsystem Scope](#)

[6.8 Three Scenario Classes That Do Not Exist at IP Level](#)

[6.8.1 Boundary Handoff Scenarios](#)

[6.8.2 Failure Mode and Error Propagation Scenarios](#)

[6.8.3 Shared Resource Contention Scenarios](#)

[6.9 How the Testbench Model Changes at Subsystem Level](#)

[6.10 Debug at Subsystem Level: Edge-Following Fault Localization](#)

[6.11 The Subsystem Quality Gate](#)

[6.12 Quantified Impact at Subsystem Scope](#)

[6.12.1 Per-ECO Validation Cost Amplification](#)

[6.12.2 Late-Stage Bugs Are Bounded, Not Eliminated](#)

[6.13 Alignment with Industry Standards and Specifications](#)

[6.14 VEGA Between IP and SoC Verification](#)

6.15 Conclusion

7 Quality Metrics for AI-Assisted Verification

7.1 Why Measurement Matters

7.2 Two Failure Modes: Generation vs. Specification

7.2.1 Generation-Side Failures

7.2.2 Specification-Side Failures

7.2.3 Why the Distinction Matters

7.3 Generation-Side Failures: The Eight-Mode Taxonomy

7.3.1 Error Clustering

7.4 Generation-Side Metrics: RES, VG, SCR, HR

7.4.1 Repair Efficiency Score (RES)

7.4.2 Verification Gap (VG)

7.4.3 Specification Coverage Ratio (SCR)

7.4.4 Hallucination Rate (HR)

7.4.5 The Metrics Read Together

7.4.6 The Bridge, Scored

7.5 Specification-Side Failures: The Firmware-Hardware Contract

7.5.1 What the AI Generated

7.5.2 The Programming Contract

7.5.3 Six Real Bugs the AI Testbench Found Anyway

7.6 Specification-Side Metrics: PMC, CVL, DFS, OBS

7.6.1 Programming Model Coverage (PMC)

7.6.2 Contract Violation Latency (CVL)

7.6.3 Descriptor Fidelity Score (DFS)

[7.6.4 Reading the Three Sub-Scores Together](#)

[7.6.5 Observability Score \(OBS\)](#)

[7.7 Process-Side Metrics: Autonomy and Reproducibility](#)

[7.7.1 Autonomy Index \(AUI\)](#)

[7.7.2 Regeneration Stability \(RGS\)](#)

[7.8 The Unifying Thesis: Specification, Not Model](#)

[7.9 Evaluation and Adoption Guidance](#)

[7.9.1 Evaluating LLM-Based Verification Tools](#)

[7.9.2 Measuring an Existing Verification Pipeline](#)

[7.9.3 Writing the Specification That Feeds AI](#)

[7.9.4 Connection to the Sign-Off Argument](#)

[7.10 Conclusion](#)

[8 Verification Closure: From Coverage Evidence to Sign-Off Argument](#)

[8.1 The Sign-Off Problem](#)

[8.2 What Done Actually Means](#)

[8.3 Coverage as XTP Evidence](#)

[8.3.1 Deriving Coverage Goals from XTP Rows](#)

[8.3.2 Coverage Goals Written at Plan Time, Not Regression Time](#)

[8.4 The Bridge Coverage Architecture](#)

[8.5 The Three Kinds of Unclosed Coverage](#)

[8.5.1 Worked Example: Defining and Closing Bridge Coverage with AI](#)

[8.6 Structural Coverage and What It Proves](#)

[8.6.1 Structural Coverage Exclusion Governance](#)

8.7 The Regression Plateau and Directed Closure

8.7.1 Moving from Plateau to Directed Closure

8.7.2 Bug Find Rate as a Sign-Off Leading Indicator

8.8 The Sign-Off Argument: Structure and Required Components

8.8.1 XTP Row States at Sign-Off

8.8.2 The Residual Risk Register

8.9 Sign-Off as a Forward-Looking Artifact

8.10 Conclusion

Appendix A: The Educational Bridge Model

Afterword

References

Index

About the Author

Vikash Kumar is Senior Verification Architect with more than 15 years of experience building and leading verification programs for some of the most complex semiconductor designs in the industry. His career spans multiple generations of silicon at companies including Arm and Intel, where he has worked on IP-level, subsystem-level, and full-SoC verification across a broad range of protocol interfaces including AXI, CHI, PCIe, CXL, UCIe, AHB, and APB. He has designed and built UVM testbench environments from the ground up for complex IP blocks and subsystems, led verification teams across multiple programs, and driven verification closure on designs that have shipped to production silicon. His work covers the full verification lifecycle: from specification analysis and test plan development through regression closure, debug, and sign-off. Beyond his industry work, Vikash is Active Contributor to the broader verification community. He is IEEE Senior Member and participates in the Senior Member elevation process, reviewing applications and supporting the recognition of engineering excellence across the profession. He reviews technical papers for IEEE journals and conferences, contributing to the quality of published research in hardware verification and AI-assisted design methodologies. He has delivered voluntary technical training to engineers entering and advancing in the verification field and has presented his work on structured, AI-assisted verification methodology at international technical conferences. The VEGA framework described in this book is the product of that experience—a distillation of what structured verification looks like when it is practiced at industrial scale, across real designs, under real schedule pressure, with real consequences for getting it wrong.

LinkedIn: <https://linkedin.com/in/vikash-kumar-95153b49>

1. Introduction: The Evolution of Hardware Verification and the VEGA Framework

Vikash Kumar¹✉

(1) Leander, TX, USA

✉ **Vikash Kumar**

Email: vikash.singh261@gmail.com

Hardware verification routinely consumes the majority of a chip program's effort and remains its least predictable phase, yet much of that effort is mechanical: writing the testbench scaffolding that drives a design and observes how it responds. Modern AI can now generate that scaffolding in minutes, which relocates the value of verification skill from typing to judgment. The relocation also introduces a characteristic failure mode, the verification gap: the distance between a testbench that compiles, runs, and reports green and one that actually exercises the behaviors a specification requires. This opening chapter defines that gap and introduces VEGA (Verification Engineering through Guided AI), a human-centered framework that implements a cognitive verification architecture as a governance layer above UVM, SystemVerilog, formal analysis, and existing EDA tooling rather than a replacement for them. It states VEGA's three governing principles, recasts the engineer as a Verification Strategist who directs and reviews AI-generated artifacts at every quality gate, and introduces the Educational Bridge Model used as the worked example throughout the

book. It closes by mapping the progression from design-intent extraction through measurement to a defensible sign-off.

1.1 The Skill Worth Having

You have heard the line by now. AI will not replace engineers. But engineers who use AI will replace those who do not.

It is true. It is also close to useless on its own, because almost no one tells you *how*. The advice stops at “use AI” and leaves you to work out the rest, which is the part that takes years.

This book is the *how*. It is the *how* for one of the hardest and most valuable jobs in building a chip: verification, the work of proving that a design does what it is supposed to do before it is manufactured into silicon that can never be patched.

If you have wanted to do this work but felt locked out by the wall of prerequisites, the UVM frameworks, the SystemVerilog, the decade of accumulated convention, this book is written for you. Not because those things stopped mattering. Because the wall was never the real work.

Here is what most people do not realize about verification. The job has two layers. The first is mechanical. It means writing thousands of lines of scaffolding, the *testbench*, which is the code that drives the design and watches how it responds, then wiring it together and getting it to compile. This is the layer that takes a newcomer two years to become fluent in. It is also the layer AI is now good at.

The second layer is judgment. It means deciding what is worth testing, looking at a result and knowing whether it proved anything at all, and deciding when a design is safe to manufacture. This layer cannot be automated, because it is accountability, not typing. It is also the layer that was always the point.

AI did not lower the value of verification skill. It moved the entrance. The part that used to gate you out, the boilerplate, is now the part you can hand to a machine. What remains is the part worth having.

The loop this book teaches is simple to picture, even if you have never written a line of SystemVerilog. You take a *specification*, the document that says what the chip is supposed to do. You hand it to an AI, under a structure that keeps it honest, and it produces a working testbench in minutes instead of weeks. Then you do the human part. You read what it built. You ask whether it is testing the right things. And you push on it until it surfaces a real bug (Fig. 1.1).

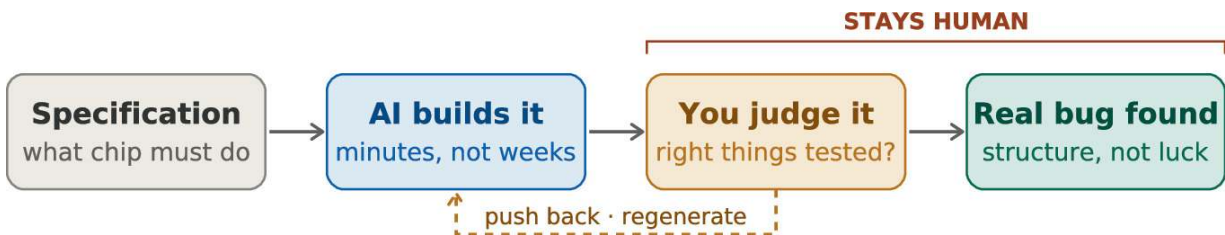


Fig. 1.1 The AI-assisted verification loop. The AI does the building; you decide whether it built the right thing

That last step is not hypothetical. AI-assisted verification, with a human reading and judging the output, finds real RTL bugs, not just simulations that pass, and it surfaces them faster than a hand-written testbench would.

You will learn UVM and SystemVerilog in this book. But you will learn them the way anyone retains anything, by using them to build something real, with AI carrying the weight that used to make people quit before they started.

1.2 Why Verification Is the Hard Part

A chip does not ship when the design is finished. It ships when the design has been verified. That single distinction defines the real schedule of every hardware program.

When people talk about building a chip, the work that gets the attention is design: writing the logic that makes the thing work. But across real programs, verification, the work of checking that the logic is correct, routinely consumes the majority of total effort. Practitioners

and program managers consistently report it in the range of sixty to seventy percent. It is also the least predictable phase. A design task can be estimated. A verification task often cannot, because you do not know how many bugs are hiding until you go looking, and you do not know how long each one will take to find.

The reason is not a shortage of tools. Simulation, formal analysis, and regression all have decades of mature automation behind them. The reason is that the hard part of verification is not running tests. It is deciding what to test, in what order, and knowing when you are done. Those are reasoning problems, and reasoning does not get faster just because the computer does.

Traditional methods lean on probabilistic exploration. Constrained-random stimulus, coverage-driven closure, and assertion-based checking have carried the field a long way. Each of these is explained when you need it; for now the point is only what they have in common. They infer completeness from indirect measurements, and they rarely tie those measurements back to the original specification in any explicit way. For protocol designs this matters, because correctness there depends on global properties such as ordering and consistency across separate agents, and no single coverage point captures those.

Consider a common situation. A verification engineer is handed a two-hundred-page specification and asked to produce a test plan in a week. There is no structured process for extracting the features, resolving the ambiguities, or finding the interactions between blocks. The plan that results reflects individual instinct more than systematic coverage of the document. This is not a failure of effort. It is a failure of method.

In most programs, planning, building the testbench, writing scenarios, debugging, and signing off are treated as separate, loosely connected jobs, each resting on individual experience. The result is verification that is hard to forecast, where a gain in one phase does not reliably carry into the next.

1.3 Verification Is a Reasoning Problem, Not a Typing Problem

If verification is fundamentally about reasoning, then its failures are usually failures of reasoning, not failures of effort. When the intent behind a test, the assumptions it makes, and the reason it exists are all left unwritten, the work becomes impossible to review and impossible to scale. Point automation at an unstructured process and it does not fix the disorder. It multiplies it.

This is the central risk of applying AI to verification without a governing framework. A language model given a specification and asked to generate tests will produce tests, syntactically correct, structurally plausible, and largely disconnected from what the design actually needs to prove. It will confirm the behaviors it was trained to expect and miss the corner cases it was not. Without structure, AI does not accelerate verification. It accelerates the production of passing simulations that verify nothing.

The answer is to make the reasoning explicit. Write the intent down. Capture the decisions in a form a human can review and a machine can act on. Once the reasoning is on the page, people and tools can work against the same shared understanding, and automation finally has something solid to stand on.

That is the idea behind a *cognitive verification architecture*: organizing verification so the thinking is visible, reusable, and governed, rather than locked in one engineer's head. The goal is not to automate verification. It is to make verification reasoning structured enough that automation becomes worth having.

1.4 The Verification Gap

The cognitive limits described in the previous section produce a specific, measurable failure mode in modern AI-assisted verification.

We name it here explicitly because the rest of this book is organized around closing it.

The Verification Gap

A testbench that compiles is not a testbench that verifies. A simulation that passes is not a simulation that proves correctness. The verification gap is the distance between syntactic plausibility, code that builds, runs, and reports green, and semantic correctness, code that actually exercises the behaviors a design must guarantee.

The verification gap is not a theoretical concern. It manifests concretely when verification artifacts are produced without an explicit model of what the design is required to do. A language model, an inexperienced engineer, or a hurriedly assembled regression suite will all produce the same failure mode: tests that exercise the design's typical paths, miss its critical boundaries, and report passing results that mean far less than they appear to.

Three properties characterize the gap:

- **Syntactic correctness without semantic grounding.** Generated testbenches compile, simulate, and produce coverage reports. None of these outputs guarantee that the tests reflect specification intent. A constraint that randomizes within legal bounds is not the same as a constraint that targets architecturally meaningful scenarios.
- **Coverage closure without intent closure.** Functional and code coverage metrics report what was exercised. They do not report whether what was exercised is what the design was required to guarantee. Coverage of a behavior the specification never required is wasted work. Absence of coverage on a behavior the specification mandates is undetected risk.
- **Pass-rate confidence without sign-off defensibility.** Passing regressions create the impression of progress. They do not, by themselves, support a sign-off argument. The question is not whether

tests pass. It is whether the tests that pass are the right tests, and whether their passing constitutes evidence against the specification.

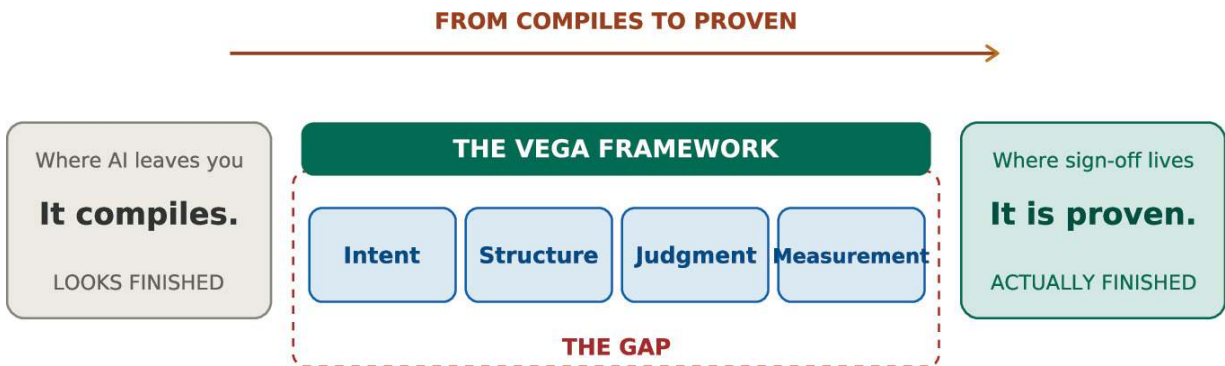


Fig. 1.2 Closing the verification gap. Explicit intent, governed structure, human judgment, and measured quality are the supports that carry a design from “it compiles” to “it is proven”

The gap exists because conventional verification flows, and the AI tooling layered on top of them, optimize for what is easy to measure (lines of code, coverage percentages, regression pass rates) rather than what is hard to measure (whether the verification activity is grounded in design intent). The result is verification that scales in volume but not in confidence (Fig. [1.2](#)).

Closing the gap requires a verification process in which design intent is made explicit, captured in a form that both humans and tools can reason against, and used as the governing reference for every downstream artifact: testbench structure, test scenarios, coverage definition, debug, and sign-off. Without such a reference, automation amplifies the gap. With it, automation becomes a genuine accelerator.

This is the problem the VEGA framework is designed to address. The remainder of this book develops the methodology, infrastructure, and governance practices required to close the verification gap systematically, from specification through sign-off, and to measure how much of it remains when AI is involved in producing the verification artifacts themselves.

1.5 The VEGA Framework

VEGA, Verification Engineering through Guided AI, is introduced in this book as the practical implementation of Cognitive Verification Architecture. VEGA is a human-centered, structured verification framework for hardware interfaces, subsystems, and protocols.

VEGA is not a replacement for existing verification methodologies or tools. It does not replace UVM, SystemVerilog, formal analysis, or any existing EDA tooling. Instead, it operates as an architectural governance layer above them, augmenting established practice with structured reasoning, explicit decision points, and semantic continuity from specification through sign-off.

Its primary objective is to reduce verification time variability and late-stage churn by applying a consistent, layered strategy across all verification phases. VEGA preserves design intent as a governing constraint while enabling modern tooling, including AI assistants, to act as accelerators rather than autonomous decision-makers.

The principles and methods in this book apply to any hardware design. The VEGA framework is not tied to a specific protocol, interface, or design class. Where worked examples are needed, this book uses the **Educational Bridge Model** (Appendix [A](#)), an educational AHB-to-APB bus bridge that converts transactions from the Advanced High-performance Bus (AHB) to the Advanced Peripheral Bus (APB). This design is compact, fully specified, and representative of the protocol bridging challenges found across real SoC integration. Every concept illustrated with this example applies equally to AXI interconnects, PCIe controllers, DMA engines, coherency fabrics, or any other IP block or subsystem.

The choice of a small bridge is deliberate. A production protocol specification can run to many hundreds of pages, and a book built on one would spend most of its length teaching that specification rather than the method. The Educational Bridge Model is sized to avoid that: small enough to hold in your head, so no chapter is lost to spec reading, yet complete enough to exercise every layer of the framework, from intent extraction through debug and coverage, including a real bug that has to be found. The design is the vehicle. The method is the subject.

Despite its size, the bridge contains the elements that make real verification hard—two distinct bus protocols and the translation between them, a register file, a control FSM, and multiple error paths (protocol error, address-decode error, and timeout). It condenses the structure of an industrial specification into a form a reader can hold in their head, then scale and apply to any production IP or subsystem (Fig. 1.3).

DESIGN INTENT

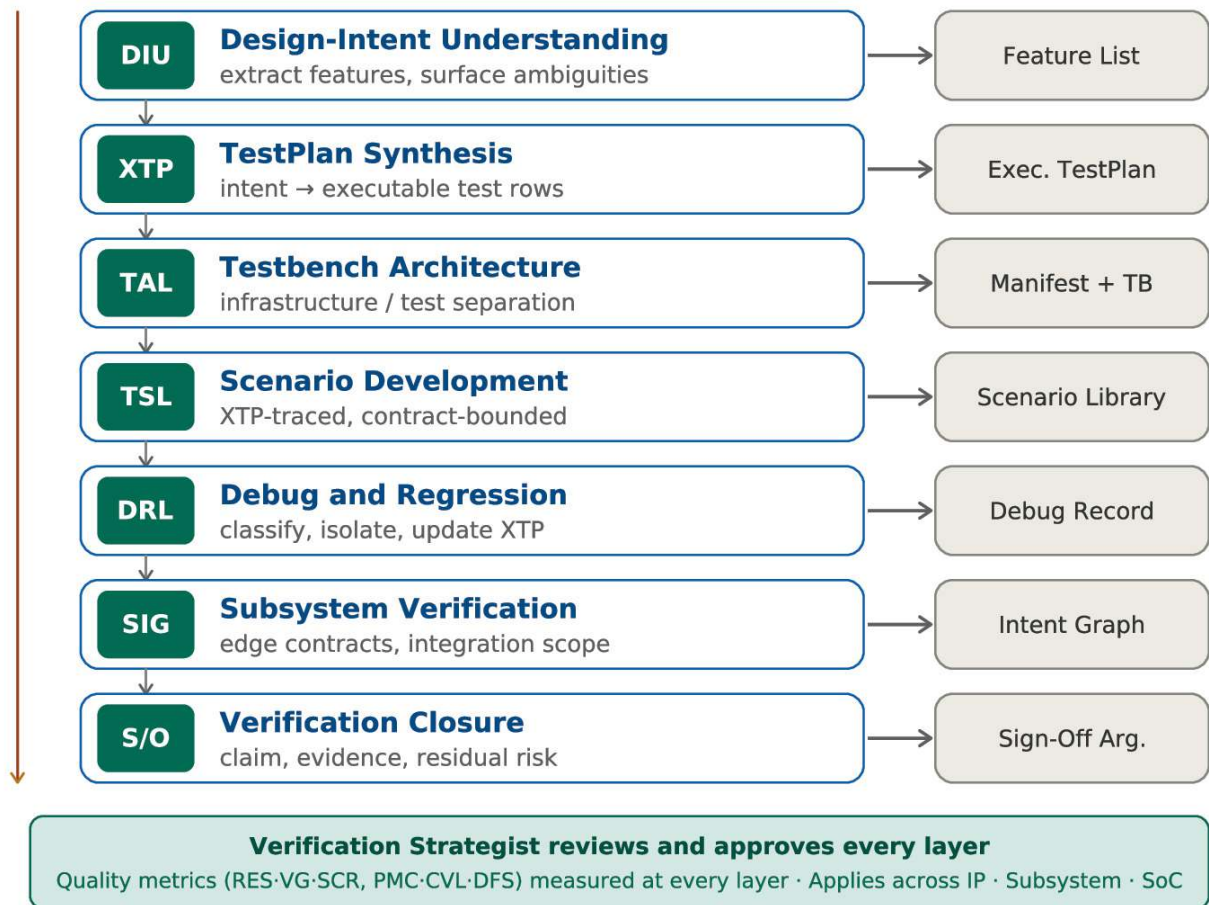


Fig. 1.3 The VEGA framework as a sequence of layers governed by the Verification Strategist and measured by quality metrics at every layer. Layer acronyms: *DIU* Design-Intent Understanding; *XTP* Executable TestPlan; *TAL* Testbench Architecture Layer; *TSL* Test Scenario Layer; *DRL* Debug and Regression Layer; *SIG* Subsystem Intent Graph

VEGA is guided by three foundational principles:

- **Cognitive Amplification:** Modern tooling, including AI assistants, is used to extend the engineer's ability to reason over large

specification, state, and scenario spaces. The goal is to amplify human judgment, not to obscure or replace it.

- **Operational Transparency:** Every automated inference and generated artifact must be explainable, traceable, and subject to human review before it influences downstream verification decisions.
- **Design-Intent Fidelity:** Verification objectives must originate directly from authoritative specifications and architectural contracts. Intent is the governing constraint, not coverage metrics.

Together, these principles establish verification as a governed engineering process, not merely a computational one.

To make this concrete, consider what VEGA produced when applied to the bridge verification. Starting from a PDF specification, the VEGA workflow generated a complete UVM testbench scaffold, the agents, sequences, scoreboard, environment, test, interfaces, and compile scripts, in a fraction of the time required by conventional manual construction. The verification engineer then collaborated with the tooling iteratively, using waveform analysis and protocol-level reasoning to resolve timing issues and validate behavior.

The result was not just a passing simulation. The process found **real RTL bugs** in the bridge's logic. For example, the FSM was generating a phantom APB transfer for every five AHB transactions, because a registered signal cleared one cycle too late. This class of bug, at the boundary between handshake protocol and FSM state management, is precisely the kind that manual test development frequently misses and that structured, intent-driven verification is designed to expose.

This example illustrates the core VEGA value proposition: faster environment construction, structured test planning from the specification, and more systematic bug discovery, all with the engineer in control at every step. The same workflow applies to any design, at any level of complexity.

1.6 Why We Start with Protocols

Hardware interface protocols provide an ideal domain for illustrating structured verification methodology. They are architecturally central, widely reused, and defined by publicly available specifications. Errors in protocol behavior can propagate across entire systems, affecting performance, correctness, security, and power efficiency.

Protocols such as AHB, AXI, CHI, PCI Express, and UCIe exhibit verification characteristics that mirror those of full systems:

- High degrees of concurrency and non-determinism;
- Complex ordering and dependency rules;
- Interactions across power, reset, and clock domains;
- Emergent behavior arising from compliant but interacting agents.

Because protocol specifications are public and stable, they provide a transparent and reproducible foundation for illustrating verification methodology. The lessons derived from protocol verification apply directly to custom IP blocks, subsystem integration, and full SoC verification.

1.7 From One Block to a Subsystem

The VEGA framework applies uniformly across individual IP blocks, subsystems, and full SoC designs. Rather than targeting a specific class of interfaces or components, VEGA addresses verification challenges that arise whenever correctness depends on the interaction of multiple independently developed elements.

In subsystem contexts, correctness emerges from the composition of IPs, shared resources, and architectural contracts rather than from any single block or interface in isolation. Independently correct components can still produce incorrect system-level behavior when assumptions, ordering constraints, or integration policies are misaligned.

Subsystem verification represents the earliest stage at which SoC-level behavior can be reasoned about meaningfully. By structuring verification intent and governance at the IP and subsystem levels, VEGA

enables shift-left discovery of integration defects before full-system assembly, where changes are most costly, least predictable, and hardest to localize.

1.8 From Verification Engineer to Verification Strategist

This brings us back to where the chapter started, now with a name for it. The shift is not a projection about the future of the profession. It is a description of what is already happening, driven by a specific and present change: the mechanical layer of verification has become automatable.

Verification work has always had two distinct layers. The first is mechanical: writing UVM infrastructure, generating boilerplate agents and sequences, connecting analysis ports, implementing straightforward directed tests, running regressions. This work requires skill and experience. It has also, until recently, consumed the majority of a verification engineer's time.

The second layer is intellectual: deciding what to verify, resolving ambiguous specification language, classifying whether a failing test found a DUT bug or a testbench bug, judging whether coverage closure is genuine or coincidental, making the sign-off argument to management. This work also requires skill and experience, and it cannot be automated, because it requires accountability to an outcome, not just execution of a procedure.

Within the VEGA framework, engineers therefore assume the role of *Verification Strategists*. This is not a change in job title. It is a change in where engineering judgment is applied. The Verification Strategist writes the intent that governs what AI generates. They review the generated artifacts at each quality gate. They interpret coverage gaps, classify failures, and make the sign-off decision. AI provides leverage. Strategic authority remains explicitly human.

The engineers who will matter most in this environment are not those who generate tests fastest. They are those who can define what must be tested, judge whether the tests are actually testing it, and defend the sign-off argument when it matters. Those skills compound with experience in a way that code generation never will.

For someone learning the field, this is the opportunity. The skill that used to take years of boilerplate to even begin building, judgment about what matters, is now the skill you start building on day one, because the AI clears the boilerplate out of your way.

1.9 How This Book Is Organized

At a high level, VEGA is organized as a sequence of governed layers that span the full verification lifecycle. Early layers focus on understanding, structuring, and preserving design intent. Later layers address testbench architecture, scenario execution, debug, and regression management. A measurement layer then establishes whether the AI-assisted work actually verified anything, and the final layer assembles that evidence into a defensible sign-off. Each stage builds on validated artifacts produced in the previous stage, and no stage proceeds without Verification Strategist review and approval.

The remainder of this book develops the VEGA framework progressively, with each chapter corresponding to a distinct layer in this architecture:

2. **Design-Intent Understanding and TestPlan Synthesis:** extracting and structuring specification intent into a machine-consumable test plan;
3. **Testbench Architecture and Environment Development:** governed construction of a scenario-ready UVM environment;
4. **Test Scenario Design and Implementation:** structured development of functional, corner, stress, and error scenarios;
5. **Debug, Failure Analysis, and Regression Management:** intent-guided failure classification and knowledge accumulation;
6. **Subsystem Verification Through Intent Composition :** scaling the framework from individual IP blocks to composed subsystems;
7. **Quality Metrics for AI-Assisted Verification :** a structured suite of metrics, grouped into families that separate generation-side failures from specification-side failures, so the quality of AI-assisted verification is measured rather than asserted, before anything is signed off;

8.

Verification Closure: From Coverage Evidence to Sign-Off

Argument: intent-driven coverage interpretation and a structured, defensible sign-off argument that records what was verified, what was not, and who accepted the risk.

An **Afterword** follows the final chapter, addressing the evolving role of the Verification Strategist in a landscape of increasingly capable AI tooling, and the directions in which the VEGA framework extends naturally: post-silicon bring-up grounded in the residual risk register, and formal verification as a first-class input to the XTP.

Figure [1.4](#) shows the whole journey on one page: seven stages, one per chapter, each producing an artifact the next stage depends on. You can return to it whenever you want to see where the current chapter sits in the larger program.

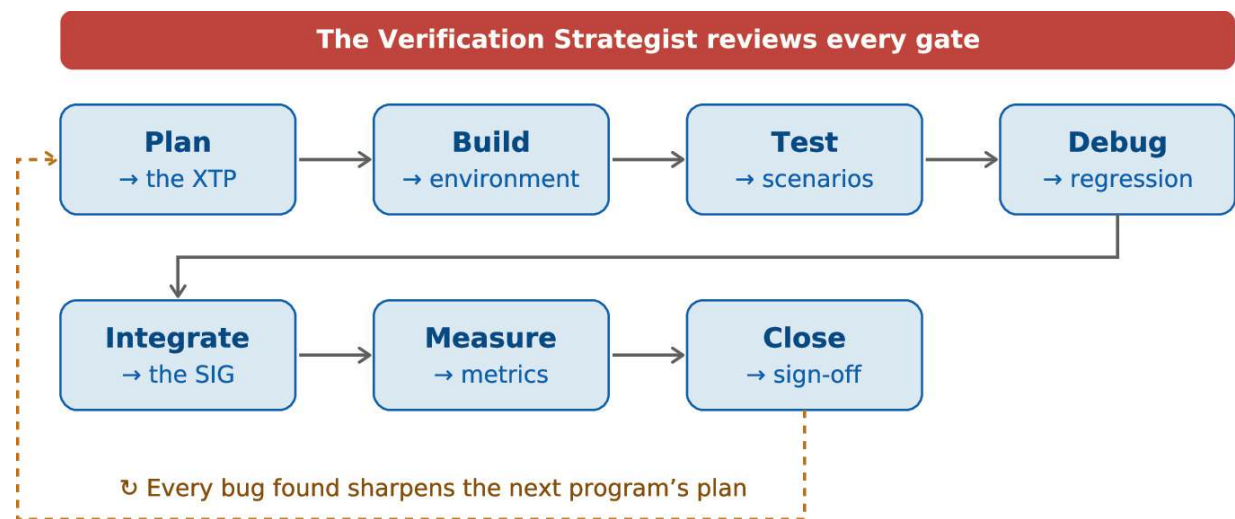


Fig. 1.4 One program, end to end. Each chapter is one stage; each builds on the artifact the last one produced, the Verification Strategist reviews every gate, and the plan learns from every bug found

2. Design-Intent Understanding and TestPlan Synthesis

Vikash Kumar¹ 

(¹) Leander, TX, USA

 **Vikash Kumar**

Email: vikash.singh261@gmail.com

Verification programs most often fail not in tooling but in planning: bugs escape to silicon because a feature was misunderstood, an ambiguity was silently resolved, or a cross-feature interaction was never identified as a risk. The remedy is to treat the test plan as the verification strategy and to produce it first. This chapter develops the Executable TestPlan (XTP), a structured, versioned, fully traceable artifact that drives implementation, regression planning, and sign-off. It presents a governed four-step flow from specification to XTP: exhaustive feature extraction paired with an explicit ambiguity register; review of the feature list against the specification to catch missed, hallucinated, and assumption-as-requirement defects; identification of cross-feature interactions before any test case is written; and generation and validation of the XTP against executable-detail and traceability checks. At every step an AI produces the draft and a Verification Strategist owns the gate where judgment catches what automation cannot. The same flow can be run end to end with the VEGA generator when risk is low, collapsing the gates into a single review. A

complete worked test case illustrates the signal-level precision the XTP format requires.

2.1 The TestPlan Is the Verification Strategy

You have just received a design. Your job is to verify it. Where do you start?

The instinct in most verification programs is to start building: write the testbench, set up the environment, get something running. This instinct is understandable. It feels productive. But it is the wrong place to start, and it is the reason so many programs end with bugs found late, schedules blown, and a question nobody wants to ask out loud: *did we actually find everything?*

The primary goal of verification is to find bugs. Not to build an elegant testbench. Not to achieve a coverage percentage. Not to demonstrate that the design passes a suite of tests. Those are instruments. The goal is bugs: finding every bug that matters before the design reaches silicon, where fixing it costs ten to a hundred times more.

A testbench built without a rigorous plan will exercise the paths the engineer thought of. The bugs that escape to silicon are almost always in the paths nobody thought to test, because a feature was misunderstood, an ambiguity was silently resolved the wrong way, or a cross-feature interaction was never identified as a risk. These are not tooling failures. They are planning failures.

The test plan is the verification strategy. It defines exactly what must be verified, why, and how success will be measured. A well-constructed test plan is the single greatest determinant of verification completeness. Everything else (the testbench, the scenarios, the regressions, the coverage closures) is the realization of what the test plan defined.

This is why the test plan must come first. The process requires four things:

- Extracting every verifiable feature from the specification, exhaustively, not selectively
- Surfacing every ambiguity and getting it resolved by the architect before it silently becomes a coverage gap
- Identifying every cross-feature interaction that could hide a bug
- Producing a traceable, governed artifact that drives every downstream verification decision.

Conventional verification skips or shortens this process because it is hard to do manually at scale. With the right structure, AI can extract features exhaustively, detect ambiguities systematically, and reason across the full feature space for cross-feature interactions, in a fraction of the time manual analysis requires and with greater consistency. The Verification Strategist governs every output. AI generates the drafts.

The outcome of this chapter is a complete, versioned Executable TestPlan (XTP), a structured artifact that drives implementation, regression planning, and sign-off tracking across IP, subsystem, and SoC verification efforts.

2.2 The XTP Generation Flow

VEGA produces the XTP through a governed flow of four steps. The VEGA tool runs all four internally and generates the final XTP in a single pass; this chapter teaches the four steps one at a time so you understand what the tool does and can govern its result. Conceptually, each step produces a reviewable artifact behind a gate. AI generates the drafts at every step; the human owns the gates. The gates are not bureaucracy. They are the checkpoints where judgment catches what automation cannot.

[*Table*](#)

The four steps are realized as the staged prompts inside the VEGA generator, `ch01_vega_tools/vega_xtp_gen.py`. The sections that follow walk each step in turn: what it decides, and why the gate

before the next step exists. Once you have seen the four steps and the judgment each gate protects, Sect. [2.8](#) shows how to run all four at once with the VEGA tool when the risk is low enough to collapse the gates into a single review (Fig. [2.1](#)).

Working with the AI at Each Gate

The four steps differ, but your method does not. At every step the AI drafts, you interrogate, and you decide. Hold each draft to four moves:

- **Ask:** give the AI the full specification, not a summary, and state exactly which artifact you want.
- **Distrust:** know the one place the AI is weak at this step, and look there first.
- **Interrogate:** test the draft against the specification, not against how convincing it reads.
- **Decide:** resolve what you can, and send what you cannot to the architect.

The *Verification Strategist* action note at each step is the **Decide** move made specific. The skill this chapter teaches is the interrogation: a draft that reads well is the AI's strength, and your trap.

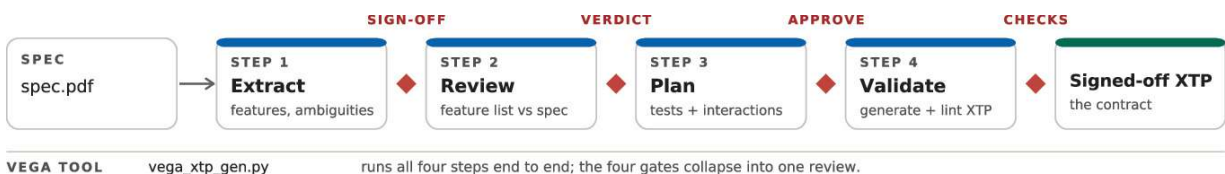


Fig. 2.1 The XTP generation flow: four governed steps from specification to Executable TestPlan, with a Verification Strategist gate between each. When the design is familiar and low-risk, the same four steps run end-to-end with the VEGA tool and the gates collapse into a single review of the final XTP

2.3 Step 1: Feature Extraction and Clarification

This step answers a single question: what does this design actually do, and what about it is unclear?

The extraction prompt reads the full specification and produces two outputs simultaneously: a structured feature list with source references, and an ambiguity register with a clarification question for each item. The two outputs are generated together because the extraction and the ambiguity detection feed the same reading pass, and separating them into sequential prompts risks missing ambiguities that only become visible when features are compared against each other.

The failure mode this step prevents is silent assumption. Without an explicit ambiguity register, an engineer reading an underspecified section will resolve the ambiguity in whatever way seems natural and proceed. That resolution becomes an invisible assumption in the test plan. Three months later, when the DUT behaves differently, no record exists of the decision that was made or who made it. The ambiguity register converts every such decision into an explicit architect question before any test is written.

Verification Strategist action: Review the feature list and ambiguity register. Send every blocking ambiguity to the design architect. Update the feature list with architect responses before proceeding to Step 2. Do not proceed with unresolved blocking ambiguities.

2.4 Step 2: Feature List Review

Before any tests are written, the extracted feature list must be validated against the original specification. This step is a quality gate, not a formality.

A missed feature at this stage costs one correction. The same missed feature discovered after the testbench is built costs an order of magnitude more: the scenario must be written, the coverage model

updated, the regression re-run, and the root cause of the miss investigated. The economics strongly favor finding the gap here.

Step 2 catches three categories of problem that consistently escape Step 1. **Missed features** are sections of the spec that were read but not extracted, particularly appendices, timing diagrams, and error-handling clauses, which LLMs deprioritize when the main datapath is more prominent. **Hallucinated features** are behaviors that appear in the feature list but cannot be traced to a specific line in the specification. The model inferred them from protocol training data rather than extracting them from the document. **Ambiguities masquerading as requirements** are the most dangerous category: assumptions that were stated as facts rather than flagged for architect review. These pass every structural check and only reveal themselves when the DUT behaves differently from what the test assumes (Fig. 2.2).

*Verification Strategist action: Review the verdict. If **NEEDS FIXES**, correct the feature list and re-run Step 2. If **BLOCKED**, resolve the flagged ambiguities with the architect before proceeding. Only a **READY FOR STEP 3** verdict unlocks Step 3.*



Fig. 2.2 Reviewing the feature list against the spec. Three failure classes hide in the gap: features in the spec but missing from the list, features in the list with no spec line, and assumptions stated as requirements. Only a clean reconciliation earns **READY FOR STEP 3**

2.5 Step 3: Cross-Feature Interactions and the Test Plan

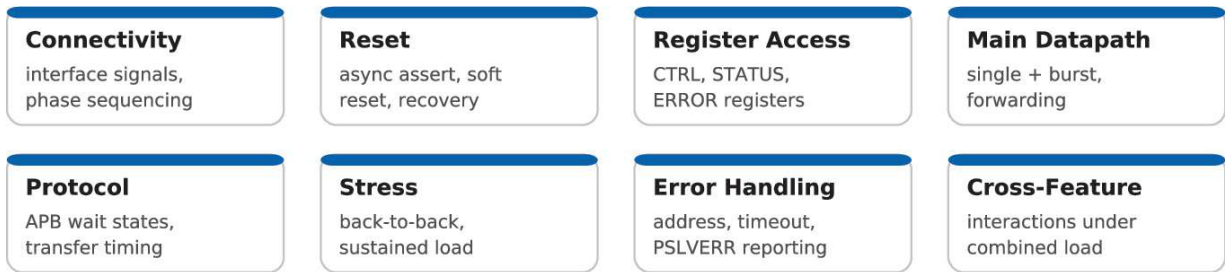
With a validated feature list in hand, this step generates the test plan. But it does something before writing a single test case that most verification programs skip entirely: it identifies cross-feature interactions.

A cross-feature interaction is a situation where two or more features, each correctly specified in isolation, produce unexpected behavior when they operate together. These are not edge cases. They are the dominant category of integration bugs. A reset that is asserted mid-transfer. A burst that starts while an error recovery is in progress. A clock-domain crossing that is correct under single-transfer conditions but fails under back-to-back load. None of these bugs exist within any single feature. All of them are invisible to a test plan organized around individual feature verification.

The Step 3 prompt identifies cross-feature interactions before writing test cases, because the test case generation must know which interactions exist to allocate scenarios for them. An interaction identified at this stage costs one scenario. An interaction discovered during silicon debug costs a respin.

The output is a test plan table with every row traced to either a Feature_ID or an Interaction_ID. Priority is explicit: P0 tests are must-pass for sign-off, P1 tests are important, P2 tests are nice-to-have. The Verification Strategist reviews the P0 set before proceeding. Confirming that it is both necessary and sufficient for sign-off is a judgment call that AI cannot make (Fig. [2.3](#)).

Verification Strategist action: Review the test plan table. Confirm all P0 tests are necessary and sufficient for sign-off. Verify that every cross-feature interaction has at least one test. Approve before proceeding to Step 4.



An empty category is not a clean result. It is a class of behavior no test will reach.

Fig. 2.3 The eight categories a complete XTP must span. Every feature classifies into one; a category left empty is a class of behavior no test will reach

2.6 Step 4: Generate and Validate the XTP

The final step converts the approved test plan into a machine-consumable XTP and validates it. This is the last gate before the XTP is handed to the testbench engineer as a contract. Validation happens in two layers.

The first layer checks that every test is specified precisely enough to execute. LLM-generated plans have a predictable failure mode: tests that are structurally correct but verification-useless, steps that say “verify the transfer completes” rather than “sample PREADY at T2, confirm PREADY=1 and PSLVERR=0.” The quality validator runs twelve checks to enforce executable detail. The ones that matter most:

- **Signal-level detail:** every step must use signal=value format
- **Hex addresses:** every register access must use hex notation
- **Timing references:** every step must reference a clock edge, phase, or cycle number
- **Verification step:** every test must include a read-back or observable pass condition
- **Step count minimums:** reset tests require 10 steps, error tests require 15. The counts enforce that these categories receive the depth they need
- **Rule ID coverage:** spec rule references must appear at 70% or better across the requirement set

- **Register address consistency:** the same register must use the same hex address every time it appears
- **Traceability:** every test must reference a requirement; every requirement must reference a test.

The second layer checks the plan against the specification: every requirement maps to at least one test, every test maps to at least one requirement, every test has a stimulus and a pass condition, no test rests on an undocumented assumption, and no test references infrastructure that does not exist. Any failing check requires a return to an earlier step as appropriate.

A plan that passes both layers can be handed to the testbench engineer as a contract. One that fails is a documentation artifact, not a verification specification. From this point forward, every testbench decision traces to a requirement in this document.

Verification Strategist action: Review the validation report. Resolve every FAIL finding before signing off. The signed-off XTP is the input to every subsequent chapter.

2.7 XTP Reference: One Test Case in Full

A single test case from the bridge's XTP illustrates what the format requires. Every field is mandatory. Every step uses the signal-level detail the quality validator enforces.

The test shown here is `TEST_MAIN_DATAPATH_001`, the single write conversion, the simplest path through the bridge. It is the right starting point: if this test is specified incorrectly, every more complex scenario built on top of it inherits the error.

Reading this case carefully before writing the testbench is not optional preparation; it is the first act of verification.

```

<testplan name="ahb2apb_verification" version="2.0">
  <features>
    <feature req_id="REQ_001"
      name="Single Transfer Conversion"
      category="main_datapath">
      <description>
        Each AHB beat converts to exactly one APB transfer.
      </description>
      <signals>HADDR, HWDATA, HRDATA, PADDR, PSEL, PENABLE</signals>
      <requirements>
        <requirement
          text="CONV-1: Each AHB beat shall produce one APB transfer"
          page="section 3.2"
          test_ref="TEST_MAIN_DATAPATH_001"/>
        </requirement>
      </requirements>
    </feature>
  </features>
  <test_suites>
    <test_suite name="main_datapath_tests">
      <test_case test_id="TEST_MAIN_DATAPATH_001"
        name="single_write_conversion">
        <req_ref>REQ_001</req_ref>
        <objective>
          Verify single AHB write produces exactly one APB transfer
          with correct address, data, and response.
        </objective>
        <preconditions>
          <condition>Bridge out of reset, CTRL ENABLE=1</condition>
          <condition>HCLK and PCLK free-running (single domain)</condition>
        </preconditions>
        <steps>
          <step num="1">
            <action>At T1: HSEL=1, HADDR=0x0000_1000,
              HTRANS=2'b10, HWRITE=1,
              HWDATA=0xDEAD_BEEF</action>
            <expected>APB SETUP: PSEL=1, PENABLE=0,
              PADDR=0x0000_1000, PWRITE=1</expected>
          </step>
          <step num="2">
            <action>At T2: PENABLE=1, PREADY=1, PSLVERR=0</action>
            <expected>HREADY_OUT=1, HRESP=0,
              transfer complete</expected>
          </step>
          <step num="3">
            <action>Read back: HADDR=0x0000_1000,
              HWRITE=0, HTRANS=2'b10</action>
            <expected>PRDATA=0xDEAD_BEEF, HRESP=0</expected>
          </step>
        </steps>
        <pass_criteria>
          One APB transfer per AHB beat.
          Read-back matches written data.
          No protocol errors.
        </pass_criteria>
      </test_case>
    </test_suite>
  </test_suites>
</testplan>

```

The signal values are not illustrative. They are the test. If you are still learning the protocol, do not worry about decoding every signal name here; the point is the precision, not the mnemonics. A step that says “write test data to the bridge” has no pass condition. A step that says “At T1: HADDR=0x0000_1000, HWDATA=0xDEAD_BEEF” can be automated, reviewed, and traced. That precision is what separates a verification contract from a verification intention.

Figure 2.4 lays the same test case out visually, with callouts naming what each field contributes to a verifiable test.

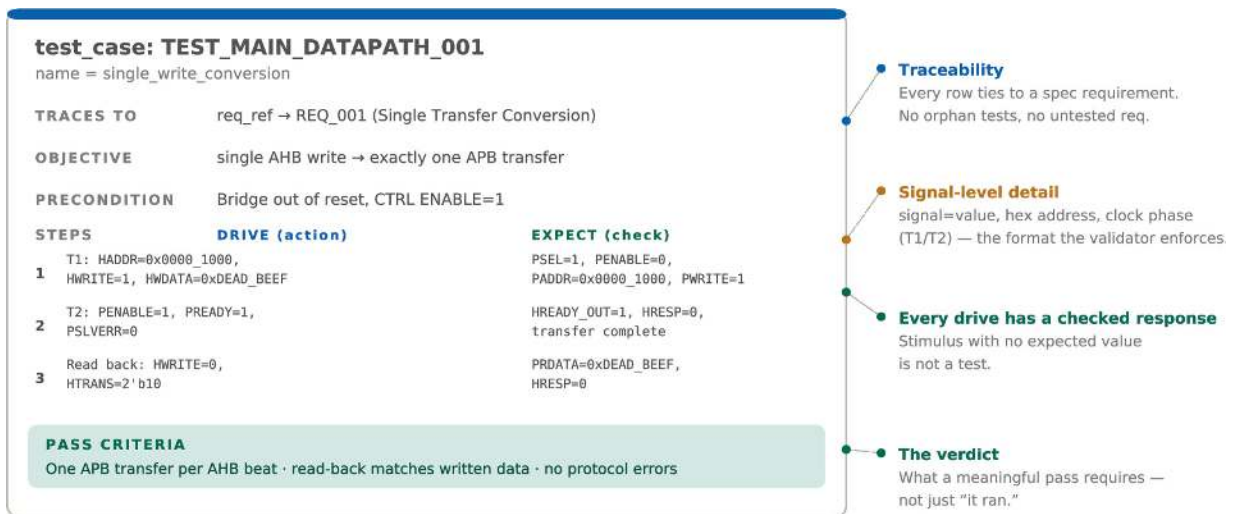


Fig. 2.4 Anatomy of one XTP row, TEST_MAIN_DATAPATH_001. The mandatory fields, traceability to a requirement (REQ_001), signal-level DRIVE and EXPECT steps, and explicit pass criteria, with callouts for what each part contributes. A drive without a checked response is not a test; a row without a requirement is an untraceable claim

2.8 Running the Whole Flow with the VEGA Tool

You have now seen the four steps and the judgment each gate protects. Once you understand them, you do not have to walk them one at a time. When the protocol is familiar, the specification is well-formed, and the cost of a missed feature is low, the VEGA tool runs all four steps in a single command, and you review the final test plan once, at the end:

```
pip install pymupdf anthropic

set ANTHROPIC_API_KEY=your-key           (Windows)
export ANTHROPIC_API_KEY=your-key       (Linux/Mac)

python ch01_vega_tools/vega_xtp_gen.py spec.pdf \
    --output ch02_testplan
```

The tool sends the full specification to the model once and caches it, then works in stages: it discovers features (standard, then design-specific, then cross-feature), audits the list for completeness, and generates the test cases for each feature in parallel, applying the same quality checks described in Step 4. A feature whose tests fall below the 80% quality bar is retried; one that still falls short is written as a flagged stub for human review rather than dropped silently. A deterministic lint validates the XML before the XTP is written:

```
ch02_testplan/spec_testplan.xtp
```

Running the flow end-to-end collapses the per-step gates into a single review at the end. That trade is only safe when the risk is low. Run it this way when the protocol is familiar, the specification is well-structured, and you are prototyping or iterating on a known design. Stop at every gate instead when the domain is new, the specification leans on tables and timing diagrams the model may misread, an architect must sign off, or the design is safety-critical and a missed feature is expensive.

One caveat governs the fast path. The quality score measures test detail, not design coverage. A score of 92% means the tests that were generated are well-specified. It does not mean that every feature in the specification was found. So when you run the flow end-to-end, your one review is not a glance at the score. It is a deliberate check that the feature list is complete, which is the judgment the per-step gates would otherwise have forced.

2.9 Conclusion

The goal of this chapter was not to produce documentation. It was to maximize the probability of finding every bug in the design before it reaches silicon.

The XTP that emerges from this flow, whether you walked the four gates yourself or ran them in one command and reviewed the result, is not a planning artifact. It is a contract. Every requirement in it has a source in the specification. Every test in it has a falsifiable claim. Every cross-feature interaction in it was identified before a single scenario was written.

AI accelerates every stage of this process: exhaustive feature extraction, systematic ambiguity detection, cross-feature reasoning at scale. The Verification Strategist governs every output. The result is not a smarter testbench. It is a more complete understanding of the design, captured in a form that every downstream activity can honor.

The bugs that escape to silicon are the ones that were never described. This chapter is about making sure every behavior worth testing gets described, precisely, unambiguously, and before a single line of testbench code is written.

With a finalized XTP in place, Chap. [3](#) shows how VEGA realizes this plan through governed testbench architecture and environment development.

3. Testbench Architecture and Environment Development

Vikash Kumar¹ 

(¹) Leander, TX, USA

 **Vikash Kumar**

Email: vikash.singh261@gmail.com

A testbench is the program that surrounds a design, drives stimulus, observes outputs, and checks them against the specification across thousands of transactions. Built without a prior architectural decision, it collapses: drivers that know about scenarios, monitors that make test decisions, infrastructure bound to one test sequence. This chapter establishes the structural rule that prevents that collapse, namely that infrastructure and test logic are separate, with intent flowing only one way, and it turns a signed-off Executable TestPlan into a UVM environment that compiles, simulates to a passing run, and produces inspectable waveforms. It introduces the VEGA LLM Testbench Generator, which builds UVM scaffolding from four structured inputs: an IP-XACT port list, a manifest, a plain-English verification-intent file, and an optional specification. An engineer writes only two, neither requiring UVM expertise. Generation, fixing, and recording are split across three tools: a batch generator, an interactive agentic fixer, and version control as the experiment record. A worked example takes the Educational Bridge Model from generated scaffolding to a passing run, and a nine-point quality gate establishes

when the environment is scenario-ready. The framework is simulator- and platform-neutral.

3.1 From a Plan to a Bench That Runs

First, what a testbench is, since the rest of the chapter assumes it. A hardware design is a description, not a machine. The RTL you write says what the logic should be, but on its own it does nothing. You cannot press run on a chip and watch it work the way you run a program, and once a design is manufactured into silicon, a bug cannot be patched the way software can. Fixing it means a new chip, with the weeks and cost that implies. So before anything is manufactured, the design is exercised in simulation, where its behavior can be observed and corrected cheaply. The catch is that a real design has far more input combinations, orderings, and timing situations than anyone can check by hand, and reading the code to convince yourself it is right is exactly how bugs survive to silicon.

A testbench is the program that does this checking for you. It surrounds the design, generates stimulus (the inputs the design would see in a real system), drives those inputs under controlled clocking and reset, observes the outputs the design produces, and compares them against what the specification says should happen. When the two disagree, it flags the exact transaction that failed. It does this automatically, across thousands of transactions, so you are testing the design's behavior rather than trusting it. Building that harness, and getting it to compile, run a real simulation, and check something meaningful, is most of the work of verification, and it is what this chapter produces.

You have a signed-off Executable TestPlan (XTP). You know exactly what needs to be verified. The natural next step seems obvious: open a file and start writing the testbench.

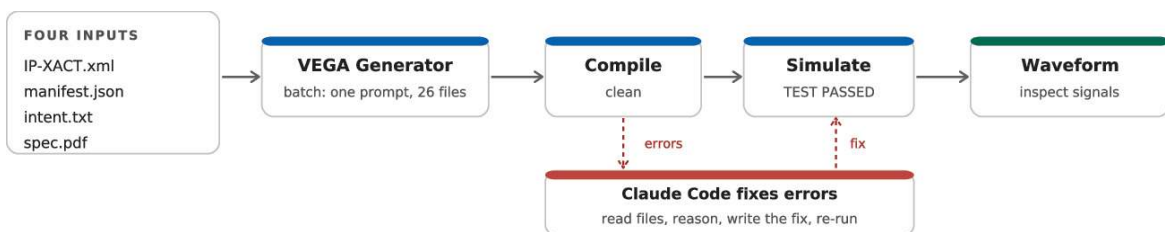
Most engineers do exactly that. Six weeks later the environment is structurally compromised. Drivers that know about scenarios. Monitors that make test decisions. A scoreboard coupled to a specific test

sequence. The environment works for the tests it was built for and nothing else. Every new scenario requires an infrastructure change. Every new engineer who joins spends a week decoding what is supposed to happen where.

This is not a skill problem. It is a process problem. There was no architectural decision made before the coding started, only implementation decisions made under schedule pressure.

This chapter gives you a process with one concrete outcome:

From four input files to a UVM testbench that compiles clean, runs a real simulation to TEST PASSED, and produces waveforms you can inspect. AI generates the scaffolding and fixes the errors. You govern every step (Fig. 3.1).



GIT records every iteration: iteration-0 → iteration-N. The diff between commits is the data.

Fig. 3.1 The chapter in one line. Four inputs become a generated testbench; it compiles clean and runs a real simulation; Claude Code fixes whatever broke; the result is a waveform you can read, with every iteration recorded in git

The process applies to any design: an AHB-to-APB bridge, PCIe controller, AXI interconnect, custom DMA engine, CHI fabric. You describe your DUT in structured files. An LLM generates the scaffolding. Claude Code brings it to TEST PASSED. Git records every step.

By the end of this chapter you will have:

- All tools installed, configured, and verified
- A working UVM testbench generated from four input files
- A first simulation that reaches TEST PASSED
- An experiment record in git showing exactly what changed and why

The framework runs on any SystemVerilog simulator, on Windows or Linux. The worked example in this chapter uses Questa Intel FPGA Starter Edition (FSE) because it is free, but nothing in the workflow is specific to it. The manifest selects your simulator and the generator emits the matching compile, simulation, and waveform scripts for your platform.

The worked example throughout this book is the *Educational Bridge Model* (Appendix [A](#)), an AHB-to-APB bridge whose implementation files use the name `ahb2apb_bridge`. In this chapter we refer to it simply as *the bridge*. Every command, file, and output in this chapter was produced by running the actual workflow, not synthesized after the fact.

3.2 The Core Principle: Infrastructure and Test Logic Are Separate

One rule governs everything in this chapter:

The testbench is infrastructure. Scenarios are separate. Nothing in the environment knows about test intent. Nothing in a scenario reaches into infrastructure internals.

This is not a style preference. It is the structural decision that determines whether your testbench scales or collapses under its own weight.

- Drivers drive signals according to protocol. Nothing else.
- Monitors observe and report transactions. Nothing else.
- Scoreboards compare observed transactions against expected behavior. Expected behavior comes from the test, not from the scoreboard itself.
- Assertions check protocol properties. They do not make test decisions.

- Coverage collectors sample transactions. Coverage goals are defined by the test plan, not by the collector.

When this boundary holds, the environment is reusable across designs, integration levels, and engineers. When it breaks, you rebuild from scratch for every new project.

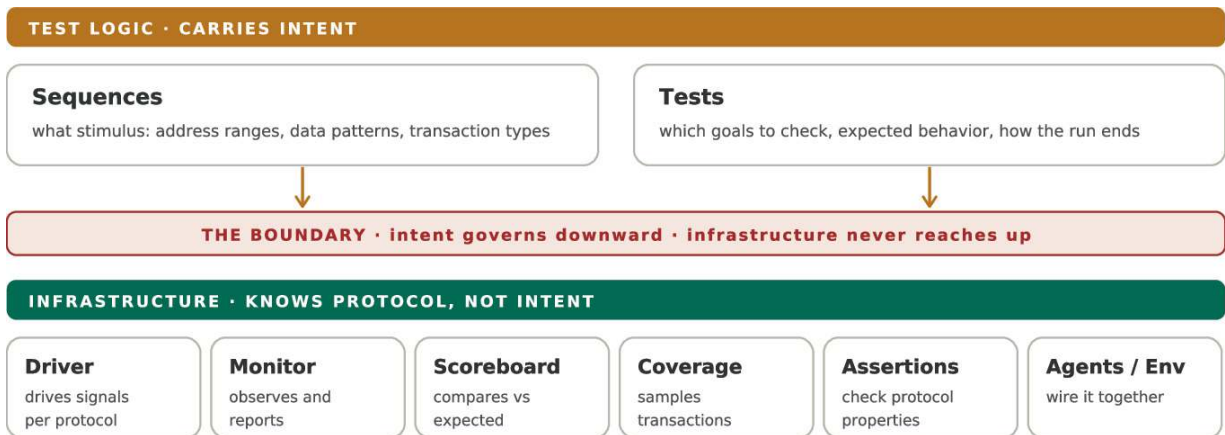


Fig. 3.2 Intent flows one way. Test logic governs what the infrastructure does; the infrastructure never reaches back up into test intent. That one-way boundary is what makes the environment reusable

3.3 The VEGA LLM Testbench Generator

The VEGA LLM Testbench Generator (`vega_llm_tbgen.py`) automates the creation of UVM testbench scaffolding from four structured inputs. It separates the work into three tools, each doing exactly one job:

Table

VEGA Generator BATCH python + API key One prompt produces the 26-file testbench. Runs once.	Claude Code INTERACTIVE node 20 + subscription Reads, runs, fixes compile and sim errors. Loops.	Git RECORD any recent version Commits every iteration. The log is the record.
SIMULATOR · ANY SYSTEMVERILOG SIMULATOR · SELECTED IN THE MANIFEST		

Fig. 3.3 Three tools, one job each. The generator is a batch tool, Claude Code is interactive, git records. The simulator, whichever you use, sits underneath them all

The script runs once. It calls an LLM to generate the behavioral files, interfaces, drivers, monitors, and scoreboard, then generates structural files deterministically: package include order, compile filelist, batch scripts, and a `CLAUDE.md` context file that Claude Code reads at the start of every fix session. It then runs `git init` and makes the initial commit.

The key insight this workflow encodes: Generation is a batch problem. One prompt, one response, done. Fixing is an interactive problem. Read a file, run a command, see what broke, decide what to change. These are different problems requiring different tools. Earlier approaches that tried to build agentic compile-sim-fix loops inside the generation script failed because they mixed both problems into one tool. Separating them made both work.

3.4 Setup

Set up three things: the generator, the fixer, and a simulator with git. The framework runs identically on Windows and Linux; install each tool with your platform's standard mechanism, and verify each install before moving on. Open a fresh terminal after each install so the updated PATH takes effect.

3.4.1 The Generator: Python and an API Key

The generator script runs on Python 3.10 or later, calls the Anthropic API, and therefore needs an API key. Install Python from `python.org` or your package manager, then:

```
pip install anthropic           :: verify: python -c "import anthropic"
pip install pdfplumber          :: optional, only if the spec is a PDF
```

If `python` or `pip` is not found after install, the install directory is not on your PATH. Add it and reopen the terminal.

Obtain an API key at <https://console.anthropic.com/settings/keys>, then set it as the environment variable `ANTHROPIC_API_KEY` (the value begins `sk-ant-...`) using your platform's standard mechanism, and make it permanent so it survives a new terminal. Verify that printing the variable from a fresh terminal shows your key, not a blank.

The key is billed per token. Generating one complete testbench with Claude Opus costs roughly \$1 to \$2.

3.4.2 The Fixer: Node and Claude Code

Claude Code is Anthropic's agentic coding tool. It runs in the terminal, reads and writes files, runs shell commands, and reads `CLAUDE.md` at the start of every session. Install Node.js 20 LTS from nodejs.org or your package manager, then:

```
npm install -g @anthropic-ai/claude-code      :: verify: claude --version
claude                                         :: authenticate once
```

Node must be 18.18 or later. An older release (for example v18.15.0) produces a `TypeError: Object not disposable` crash at startup, because Claude Code uses the `using` keyword introduced in 18.18. Node 20 LTS is recommended. On first launch, `claude` prints an authentication URL; log in with your Anthropic account and the authentication persists for every later session.

Subscription Required

Claude Code requires a Claude.ai subscription. The Pro plan (\$20/month) covers all fix sessions with no per-token billing. The Max plan (\$100/month) provides 5× higher usage limits and access to Opus models in Claude Code.

3.4.3 The Simulator and Git

Any SystemVerilog simulator works. The manifest names your simulator and the generator emits matching `compile`, `sim`, and `waves` scripts. This chapter uses Questa FSE, which is free and ships with the Quartus Prime Lite package from Intel. During installation, ensure the **Questa Intel FPGA Starter Edition** component is checked. Verify the simulator from its install location and install git with your package manager:

```
<install_dir>/questa_fse/<platform>/vlog -version      :: or your simulator
git --version                                           :: any recent git
```

Pre-simulator Constraints

Some simulators impose constraints the framework must respect. Questa FSE, for example, does not include the `svverification` license, so `.randomize()` cannot be called on user-defined objects. The generator and `CLAUDE.md` encode this rule so Claude Code never introduces a `.randomize()` call during fixes. A full simulator without that restriction lifts the constraint automatically. The point is general: simulator-specific quirks live in `CLAUDE.md`, not in your head.

Total Cost (Questa FSE Example)
Table

Claude Code fix sessions are covered by the subscription with no additional per-token billing. Only the `vega_llm_tbgen.py` generation call uses API credits.

3.5 The Four Inputs

Before running the generator, prepare four files. Together they contain everything the LLM needs to generate a correct UVM testbench. The engineer writes only two of them, `manifest.json` and `verification_intent.txt`, and neither requires SystemVerilog knowledge, UVM expertise, or signal-level understanding.

[*Table*](#)

3.5.1 IP-XACT XML

IP-XACT is IEEE standard 1685 for IP component description. IP-XACT files are frequently available from EDA tools, synthesis flows, or IP vendor deliverables. In modern design flows, the IP-XACT file is often one of the first deliverables a design team can provide. It describes the interface contract of the block before the RTL implementation is complete. For the generator it is the source of truth for the DUT interface: it gives the AI the exact signal names, widths, and directions, so the testbench interfaces and the DUT binding are built from the real port list rather than guessed or hallucinated signals.

Verification teams that receive the IP-XACT file early can begin testbench infrastructure development in parallel with RTL development, compressing the overall schedule. The design team should treat IP-XACT delivery as a verification dependency, not an afterthought: every week the file is delayed is a week of testbench development that cannot start.

If the IP-XACT file is not available, use the `--rtl` argument instead and the VEGA generator will parse the RTL module declaration directly.

```

<component xmlns="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1685-2009">
  <name>your_design</name>
  <model>
    <ports>
      <port>
        <name>CLK</name>
        <wire><direction>in</direction></wire>
      </port>
      <port>
        <name>DATA_IN</name>
        <wire>
          <direction>in</direction>
          <vector><left>31</left><right>0</right></vector>
        </wire>
      </port>
      <!-- one <port> block per DUT port -->
    </ports>
  </model>
</component>

```

3.5.2 Manifest.json

The manifest is the only file requiring deliberate engineering judgment. It defines how the testbench is configured, not how the hardware works. Each field maps to a concrete generation decision: clocks and resets set how `tb_top` drives timing, the interface list and roles decide which agents are built and whether each drives or only monitors, and the verification goals become scoreboard checks, so a wrong field here yields a structurally wrong testbench, not a cosmetic one. All fields use plain names and values; no signal names or UVM constructs appear here.

This separation is intentional: the manifest captures verification intent at a level that a verification lead can review without reading generated code, and that a new team member can understand without knowing the testbench internals.

```

{
  "project": {
    "design":      "<top_module_name>",
    "engineer":   "<your name>",
    "date":       "YYYY-MM-DD"
  },
  "tool": {
    "simulator":  "questa_fse",
    "questa_path": "<path/to/questa_fse>",
    "uvm_version": "uvm-1.1d"
  },
  "dut": {
    "top_module": "<top_module_name>",
    "rtl_files":  ["<relative/path/to/design.sv>"]
  },
  "clocks": [
    { "name": "<clock_port_name>", "period_ns": 10 }
  ],
  "resets": [
    { "name": "<reset_port_name>", "active_low": true, "assert_ns": 100 }
  ],
  "interfaces": [
    { "name": "<agent_name>", "protocol": "<ahb|apb|axi|generic>",
      "role": "active" },
    { "name": "<agent_name>", "protocol": "<ahb|apb|axi|generic>",
      "role": "passive" }
  ],
  "stimulus": {
    "num_txns":      5,
    "drain_time_ns": 200
  },
  "verification_goals": [
    { "id": "G1", "priority": "must",
      "goal": "<plain English description of what must be true>" },
    { "id": "G2", "priority": "must",
      "goal": "<plain English description of what must be true>" }
  ]
}

```

The simulator field selects your toolchain (questa_fse, vcs, xcelium, and so on); the generator emits the matching scripts. The role field is the most consequential entry. "active" means the testbench initiates transactions on this interface, so the agent's driver generates stimulus. "passive" means the agent only monitors DUT-initiated transactions.

3.5.3 Verification_intent.txt

The verification intent file captures the strategy in plain English using four sections. It takes under ten minutes to fill in for any new design. That ten minutes is not overhead. It is the most valuable ten minutes in the verification program.

An engineer who can fill in this file has understood the design well enough to verify it. An engineer who cannot is not ready to write tests, regardless of what tools are available. The verification intent file makes that readiness explicit and reviewable before a single line of testbench is generated.

DRIVE:

<What stimulus to generate. How each active interface behaves.
Transaction types, address ranges, data patterns.>

OBSERVE:

<What to monitor. When to capture a transaction.
Which signals mark the capture point.>

CHECK:

<What constitutes correct behavior. One rule per verification goal.
Written as "Gn: <condition that must hold true>".>

OUT OF SCOPE:

<What this testbench intentionally does NOT verify.
Explicitly limiting scope prevents scope creep.>

The LLM implements this intent directly in the driver, monitor, scoreboard, and sequences. The quality of the generated testbench is bounded by the quality of this file. Vague intent produces a testbench that compiles and simulates but checks nothing meaningful.

3.6 Running the Generator

This section shows the complete VEGA workflow for any design. Apply it to any DUT by substituting your design name, files, and output directory.

3.6.1 Project Directory Layout

Keep input files alongside the generator script in `ch01_vega_tools/`. The script writes the output testbench to the directory you specify with `--output`.

```
vega_framework/  
  ch01_vega_tools/  
    vega_llm_tbgen.py      <- generator script  
    manifest.json         <- you write this  
    verification_intent.txt <- you write this  
    <your_design>.xml      <- IP-XACT (from design team or --rtl)  
    <your_design>_spec.pdf <- optional  
  <output_dir>/          <- created by script  
    <your_design>_uvmtb/  
    ...
```

3.6.2 Preview Before Spending API Credits

Always run `--dry-run` first. It validates all inputs, builds the master prompt, prints the first 4000 characters, and reports token count without calling the API.

```
cd <path>/vega_framework  
  
python ch01_vega_tools/vega_llm_tbgen.py \  
  --ipxact    ch01_vega_tools/<your_design>.xml \  
  --manifest  ch01_vega_tools/manifest.json \  
  --intent    ch01_vega_tools/verification_intent.txt \  
  --dry-run
```

If the pre-flight check fails (missing sections, design name mismatch, no interfaces defined), fix the manifest before proceeding.

3.6.3 Generate the Testbench

```
python ch01_vega_tools/vega_llm_tbgen.py \  
  --ipxact    ch01_vega_tools/<your_design>.xml \  
  --manifest  ch01_vega_tools/manifest.json \  
  --intent    ch01_vega_tools/verification_intent.txt \  
  --spec      ch01_vega_tools/<your_design>_spec.pdf \  
  --output    <output_dir>/<your_design>_uvmtb
```

Optional arguments:

- `--spec <file>`: PDF, Markdown, or text design specification. Improves protocol timing accuracy in drivers and monitors. Skipped if not provided.
- `--rtl <file>`: SystemVerilog RTL file. When provided, the script parses the module declaration directly to produce `dut_bind.sv`, and cross-checks the port list against IP-XACT. Use this instead of `--ipxact` if the IP-XACT file is not available, or use both to validate consistency.
- `--no-git`: Skip `git init` and the initial commit. Not recommended.
- `--dry-run`: Preview prompt without API call.

The generator streams the response, printing each file name as it is generated. Allow 3–5 min for a complete 21-file testbench. Once it finishes, the generated scripts handle compile and simulate, and a third opens the waveform viewer (shown here as the Windows batch names the worked example produced; the generator emits the matching scripts for your platform):

<code>compile.bat</code>	<code><- compile</code>
<code>sim.bat</code>	<code><- bringup test (default)</code>
<code>sim.bat <design>_sanity_test</code>	<code><- sanity test</code>
<code>waves.bat</code>	<code><- open waveform viewer</code>

To bring the generated scaffolding to `TEST PASSED`, open Claude Code in the output directory and work through the errors (the full fix workflow is in Sect. [3.10](#)):

```
cd <output_dir>/<your_design>_uvmtb
claude
```

3.7 Worked Example: The Educational Bridge Model

The bridge converts AHB transactions into APB transactions in a single clock domain. It presents an AHB slave port to a processor and an APB master port to a peripheral bus, so the testbench drives it with the complementary roles: an AHB master agent and an APB slave agent. Both sides run on the same clock ($HCLK = PCLK$). The bridge includes an internal register file, error propagation, and a configurable timeout.

AI Output Varies Between Runs

The file counts, the specific bug, and the exact fix shown in this section come from one real run. Large language models are not deterministic and will not reproduce the same output twice, so your run will differ in the details. What does not change is the process: the generator produces a complete, compiling testbench that still contains bugs an engineer must review and fix, and the two-test loop surfaces them the same way every time. Read this section as a representative example, not a result to reproduce line for line.

This section is the framework applied to the bridge for real. Every command, file, and result here came from the actual run. The four inputs follow the templates introduced earlier in this chapter; the engineer writes the manifest and the intent file, and the bridge's six verification goals state what a correct conversion must guarantee:

- **G1:** every AHB address appears unchanged on the APB interface
- **G2:** write data reaches APB without corruption
- **G3:** read data returns to the AHB master unchanged
- **G4:** direction never flips, a write stays a write and a read stays a read
- **G5:** an APB error response becomes an error on the AHB side
- **G6:** exactly one APB transaction per AHB transaction, none lost or duplicated.

Companion Repository

The complete, runnable inputs for the bridge (ahb2apb_bridge.xml, manifest.json, verification_intent.txt, the spec) and the full generation log are in the companion repository at

https://github.com/vksabnima/vega_framework

under ch01_vega_tools/. The generated testbench, including the complete 26-file tree, lives under ch03_testbench/ahb2apb_bridge_uvmtb/. Clone the repository and run the python ch01_vega_tools/ ... commands shown in this chapter to reproduce this run end to end.

3.7.1 What the Generator Produced

One run produced 26 files from those four inputs: 21 from the LLM and 5 deterministic script files. The structure mirrors the infrastructure and test-logic boundary from Fig. 3.2:

```
ahb2apb_bridge_uvmtb\  
  ahb2apb_bridge_pkg.sv  compile.bat  sim.bat  waves.bat  CLAUDE.md  [SCRIPT]  
  env\  
    ahb_mst_agent\  
      seq_item, driver, monitor, sequencer, agent  [LLM]  
    apb_slv_agent\  
      seq_item, driver, monitor, sequencer, agent  [LLM]  
    ahb2apb_bridge_scoreboard.sv  ahb2apb_bridge_env.sv  [LLM]  
  sequences\  
    ahb_mst_bringup_seq.sv  apb_slv_bringup_seq.sv  [LLM]  
  tests\  
    ahb2apb_bridge_bringup_test.sv  ..._sanity_test.sv  [LLM]  
  top\  
    tb_top.sv  ahb2apb_bridge_dut_bind.sv  [LLM]
```

Files marked [LLM] were generated by the language model; those marked [SCRIPT] were generated deterministically by the Python script. dut_bind.sv is LLM-generated from the IP-XACT port list; with --rtl the script overwrites it from the RTL module declaration for more reliable connections. The full tree is in the repository (Fig. 3.3).

Figure 3.4 shows how those files compose into the running UVM environment, which parts are reusable infrastructure versus per-scenario test logic, and the path a transaction follows from sequence to scoreboard.

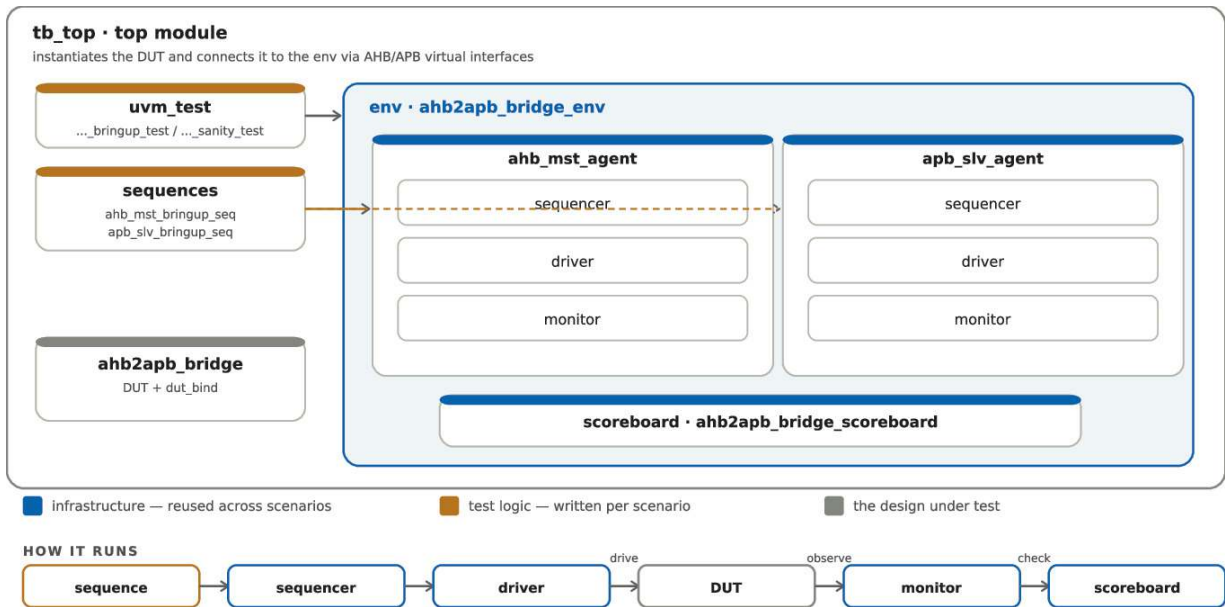


Fig. 3.4 The generated UVM environment for the bridge. `tb_top` instantiates the DUT and connects it through the AHB/APB virtual interfaces to the `env`, which holds the two agents (`ahb_mst_agent`, `apb_slv_agent`, each with a sequencer, driver, and monitor) and the scoreboard. Infrastructure (blue) is reused across scenarios; test logic (the test and sequences) is written per scenario. The bottom strip traces how a run flows from sequence to sequencer to driver to the DUT, then back through the monitor to the scoreboard

3.7.2 Fix Session: From Scaffolding to TEST PASSED

Open Claude Code in the generated directory (`cd` into the `..._uvmtb` folder, then run `claude`). It reads `CLAUDE.md` automatically. The entire fix session for the bridge was a single pass:

Fix Session: The Bridge, Iteration 1

```
> read the directory structure and files and prepare yourself
  for compile and sim error fixes
[Claude Code reads all 26 TB files in parallel]
> run compile.bat and fix any errors
[Compile passed: 0 errors, 0 warnings on first run]
> run sim.bat and fix any UVM_FATAL or UVM_ERROR
[Bringup test passed: 1 txn, 0 errors]
> run sim.bat ahb2apb_bridge_sanity_test
[3 UVM_ERRORS: AHB=4 txns, APB=2 txns : G6 FAIL, G1 FAIL]
[Claude Code reads ahb_mst_driver.sv, both monitors,
scoreboard]
[22 minutes thinking : identifies AHB address/data phase collision]
[Writes three @(vif.drv_cb) sync point fixes to ahb_mst_driver.sv]
[Re-runs sanity test]
[AHB=5 APB=5, G1-G6 all PASS]
[*** TEST PASSED ***]
```

After TEST PASSED, commit the iteration in git with a message naming what changed (for example, iteration-1: TEST PASSED sanity, AHB driver timing fixed).

The diff between iteration-0 and iteration-1 is three lines in `ahb_mst_driver.sv`. That is the entire lesson of the worked example: the LLM produced a complete, compiling testbench, lacked one piece of AHB pipeline-timing knowledge, and the loop surfaced it as a failing transaction count. What the AI got wrong, and the minimal fix that resolved it, is the data, and git holds it.

3.8 Two Tests, Two Purposes

The generator always produces exactly two tests:

Bringup test (<design>_bringup_test):

- Drives exactly one transaction
- No scoreboard checks
- Passes if simulation completes without UVM_FATAL or UVM_ERROR
- If this fails, the TB infrastructure is broken, not the DUT

Sanity test (<design>_sanity_test):

- Drives `num_txns` transactions (from manifest)
- Full scoreboard: all verification goals checked
- Passes when all goals pass and `TEST PASSED` appears in log
- If bringup passes but sanity fails, DUT behavior is wrong

Always run bringup first. The two-test strategy separates TB infrastructure bugs from DUT bugs before debugging begins.

3.9 The CLAUDE.md Context File

Every generated testbench contains a `CLAUDE.md` file. Claude Code reads it automatically at the start of every session. It contains:

- Design name and verified simulator tool path
- All verification goals from the manifest
- Summary of the verification intent
- Both test names with their purpose
- All critical UVM and simulator rules (encoded directly into the file)
- Edit hint tag meanings
- Compile, sim, and wave commands
- Workflow commands for the fix-commit cycle.

`CLAUDE.md` is not an AI prompt. It is a project specification that is read by an AI tool. When the design changes, update it manually.

One important constraint: do not add expected DUT behavior to `CLAUDE.md`. If Claude Code knows how the DUT is supposed to behave, it may adjust monitor and scoreboard fixes to match the expected behavior rather than detect violations. Monitors observe what happens. Scoreboards compare what happened against the intent. `CLAUDE.md` contains rules and context, not answers.

3.10 Claude Code in Practice: Fixing, Recording, and Waveforms

Generation is the batch step. Everything after it is the interactive loop: run, see what broke, fix, re-run, and record. Figure 3.5 shows the cycle.

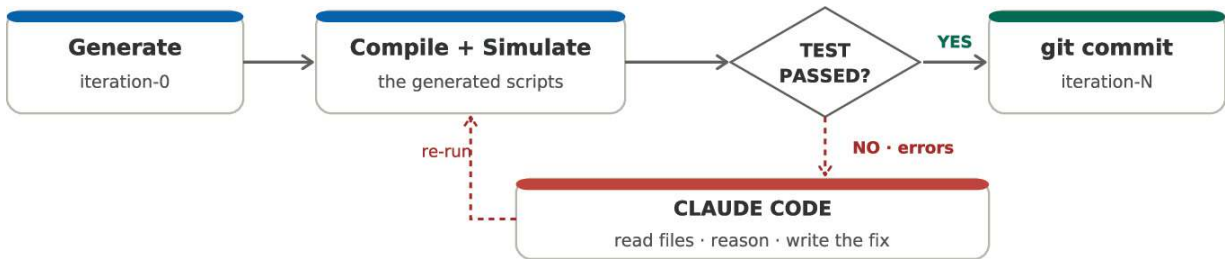


Fig. 3.5 The fix loop. Generate once, then compile and simulate; when a run fails, Claude Code reads the files, reasons over them, and writes the fix; when it passes, the iteration becomes a git commit

3.10.1 Three Operating Modes

Claude Code operates in three modes, cycled with Shift+Tab in the terminal:

- **Default mode:** proposes each file change and asks for approval before writing. Use when reviewing complex or high-risk changes.
- **Auto-accept mode:** writes changes immediately without asking. Use for routine fixes once you trust the direction.
- **Plan mode:** reads files and describes what it would do, but makes zero changes. Use when you want to review the strategy before committing to it.

The streaming token counter in the status bar shows tokens received in real time. For complex multi-file reasoning sessions, 15–25 min and 15,000–25,000 tokens is normal. Do not interrupt a session that is still receiving tokens.

3.10.2 What Claude Code Already Knows

Because `CLAUDE.md` is loaded at session start, Claude Code applies these rules automatically without the engineer stating them:

- Respect simulator constraints (for Questa FSE, never use `.randomize()`, since there is no `svverification` license)
- Use `null scope` in `config_db::set()`, not `this`

- No modport suffix in virtual interface type in `config_db`
- `timescale` only in `tb_top.sv`
- `uvm_analysis_imp_decl` at file scope before class definition
- Package include order is script-enforced; do not edit `<design>_pkg.sv`.

3.10.3 Git as the Experiment Record

`git init` runs automatically during generation, and the first commit (`iteration-0`) captures the LLM-generated scaffolding. Commit once per fix session with a message naming what changed, so the history reads as a sequence of iterations.

The diff between `iteration-0` and `iteration-1` is the primary research output of this chapter. It shows exactly what the LLM generated, where protocol knowledge was insufficient, and what minimal change produced a passing test.

3.10.4 Viewing Waveforms

Every simulator writes a waveform database, and the generated `sim` script always produces one. For Questa FSE the run is:

```
vsim -c tb_top_opt ... -wlf sim.wlf -do "log -r /*; run -all; quit -f"
```

The `log -r /*` command logs every signal in the design hierarchy. Open the viewer after any run with one command; it loads all signals and applies zoom-full automatically, so no manual wave configuration is required:

```
waves.bat
```

Useful signal paths for the bridge:

- `tb_top/u_dut/state` : FSM state (IDLE, APB_SETUP, APB_ACCESS, etc.)
- `tb_top/ahb_mst_if/*` : all AHB interface signals
- `tb_top/apb_slv_if/*` : all APB interface signals.

To confirm the AHB address/data phase timing bug in waveforms, zoom to the first two clock cycles after reset and look at `HTRANS`. If it shows `NONSEQ` followed immediately by `IDLE` in the same cycle (no gap), the bug is present. Count the number of times `PSEL` asserts to confirm the APB transaction count.

3.11 Quality Gate: Scenario-Ready Checklist

Before writing a single scenario, the testbench must pass all nine checks. Scenarios written against broken infrastructure produce misleading results that are difficult to attribute to the DUT.

1. **Bringup test passes.** One transaction. Zero errors. Zero fatals. Simulation exits cleanly.
2. **Sanity test passes.** All `num_txns` transactions. Scoreboard reports `Checked=N Pass=N Fail=0.*** TEST PASSED ***` appears in the sim log.
3. **Transaction counts match.** AHB monitor count equals APB monitor count. G6 passes by definition.
4. **All verification goals pass.** Every goal in the manifest has a corresponding scoreboard check that fires and passes.
5. **Reactive components run forever.** No passive sequence has a fixed loop count.
6. **Enum fields synced in monitors.** Every monitor calls `post_randomize()` after copying signal values.
7. **Clock and reset only in `tb_top`.**
8. **No test logic in infrastructure.** No hardcoded addresses or scenario-specific values in drivers, monitors, or scoreboard.
The commit history shows the story. `iteration-0` is the

9. baseline. Every fix has a commit with a meaningful message.

When all nine checks pass, the testbench is scenario-ready.

Chapter [4](#) introduces the first scenario against this infrastructure.

3.12 Conclusion

The VEGA LLM Testbench Generator separates three jobs that verification engineers have always done manually: generating the scaffolding, fixing the protocol timing, and recording what changed. Each job has a dedicated tool.

The generator runs once and produces 26 files from four inputs, two written in plain English with no UVM knowledge. Claude Code brings the generated scaffolding to `TEST PASSED` through iterative conversational fixing, guided by the `CLAUDE.md` context file the generator also produced. Git records every iteration automatically.

The bridge went from generated scaffolding to `TEST PASSED` in one Claude Code session and two git commits. The commit history is the experiment record, and the diff between any two commits is the data: what changed, which file, which line. When errors occur, Claude Code reads them, reasons over the relevant files, and writes the fix. The engineer commits the result and moves on.

The six to eight hours of boilerplate that the generator handled correctly is time that goes directly to understanding the DUT. The two-test strategy, bringup before sanity, localizes problems to infrastructure or DUT behavior before debugging begins. Both are meaningful separations that scale to any design, on any simulator.

The infrastructure layer is now complete and scenario-ready. Chapter [4](#) covers how VEGA realizes the XTP through governed test scenario design, with this infrastructure.

4. Test Scenario Design and Implementation

Vikash Kumar¹ 

(¹) Leander, TX, USA

 **Vikash Kumar**

Email: vikash.singh261@gmail.com

A simulation drives a design and records what happened; a test also checks the result against what the specification requires and produces a verdict. Stimulus without a checker is not a test. This chapter turns a signed-off Executable TestPlan into executable UVM scenarios that exercise the design and trace back to specification requirements. It names the four purposes tests serve and the four scenario classes, functional, corner-case, stress, and error, that determine how each plan row is written and what a meaningful pass requires. It separates directed stimulus, which confirms a known condition, from constrained-random stimulus, which explores a legal space, and names the over-constraint failure mode that defeats randomization. Each scenario is exactly one sequence and one test and never modifies the infrastructure layer. The chapter catalogs what AI gets wrong in test generation, defines the readiness checks a scenario must pass before regression, and applies a batch generator that produces a compiling sequence and test per plan row. The result is the entire plan rendered as compiling, traceable scenarios; running each scenario to a verdict, and the bugs that surface, is the next chapter.

4.1 A Simulation Is Not a Test

When the bringup test in Chap. 3 reached `TEST PASSED`, something important happened. But it is easy to misread what that meant.

The bringup test drove one transaction and confirmed the simulation completed without `UVM_FATAL` or `UVM_ERROR`. It confirmed the testbench infrastructure works. It did not confirm the bridge works. No checker compared the APB output against the AHB input. No assertion fired on a protocol violation. No scoreboard verified a single data value. The design under test could be a wire connecting input to output and the bringup test would still pass.

This is the distinction that governs everything in this chapter:

A simulation drives a design and records what happened. A test drives a design, checks what happened against what should have happened, and produces a verdict. Stimulus without a checker is not a test.

The sanity test that followed bringup was closer to a real test: it drove five transactions and the scoreboard checked that the AHB address, data, and direction appeared correctly on the APB side. But five transactions, single-transfer only, with no error injection and no boundary conditions, covers a small fraction of what the bridge spec requires. The rest is what this chapter addresses: turning the test plan into scenarios that actually exercise the design, and using AI to do it at the speed the plan demands.

4.2 Why Tests Exist: Four Purposes

Before writing a single sequence, it is worth being precise about what a test is supposed to accomplish. Tests serve four distinct purposes, and confusing them leads to test suites that are large but ineffective.

4.2.1 Find Bugs the Designer Did Not Anticipate

The most important purpose of verification is discovering behavior the designer did not intend. Designers test their own understanding of the spec. Verification tests the gap between that understanding and the actual specification.

Directed corner-case and error tests exist for exactly this purpose. They are not verifying normal behavior; they are probing the design at the edges of its specification.

4.2.2 Confirm Traceability to the Specification

Every feature in the specification must be verified. Every verification activity must trace to a specification requirement. Without this two-way traceability, sign-off is a judgment call, not a fact.

The XTP from Chap. 2 makes traceability explicit. Each test case carries a `req_ref` pointing to a specification requirement. When those tests pass, you can state with precision which requirements have been verified and which have not. This is not bureaucracy. It is the difference between a verification sign-off that stands up to review and one that collapses under the first question.

4.2.3 Build the Regression Net

RTL changes. Integration happens. Bugs get fixed and regressions get introduced. A regression suite is the net that catches new failures before they reach silicon.

A good regression suite runs in a predictable time, covers all previously verified functionality, and is sensitive enough to catch regressions without producing false failures. Tests that are too broad fail on unrelated changes. Tests that are too narrow miss regressions in adjacent logic. The XTP category structure (connectivity, register access, datapath, protocol, reset, error, cross-feature, and bringup) provides natural groupings that allow partial regression when only one subsystem changes.

4.2.4 Give Management Confidence to Tape Out

Tape-out decisions are made by people who do not read simulation logs. They read coverage numbers, test counts, and sign-off checklists.

The summary table the XTP flow produces, every test with an ID, a name, and a one-line objective, is the artifact that communicates verification status to a non-technical audience. When the suite passes, the project manager sees a complete table with green status. When some categories fail, they see exactly which ones are at risk. This is the management visibility layer that directed verification planning provides.

4.3 AI Augmentation: From Writing Tests to Reviewing Them

The four purposes above have not changed in thirty years of verification. What changes in this chapter is who does the first-level work.

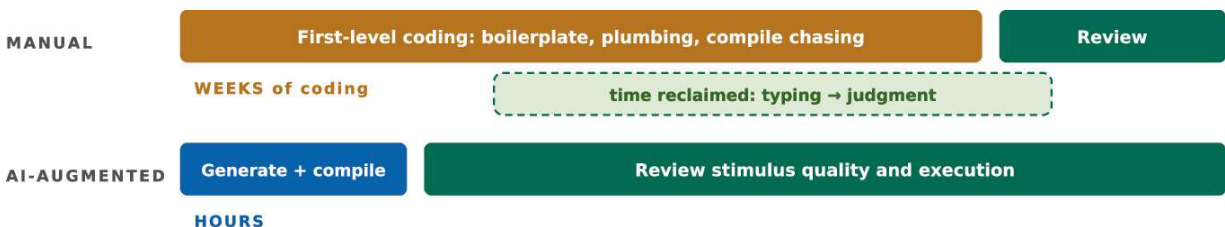
Consider the manual cost. A single scenario for this bridge is a sequence class and a test class: drive the right stimulus on the AHB side, configure the environment, start the sequence, manage the objection, and read the verdict. Written and compiled clean by an experienced engineer, that is typically thirty minutes to two hours of work per scenario, most of it mechanical: UVM boilerplate, protocol plumbing, factory registration, include ordering, and the compile-error chasing that follows. For a plan the size of the bridge XTP, that is on the order of one to three weeks of first-level coding before a single deliberate verification question has been asked.

That first-level work is mechanical, and it is exactly the kind of work a language model produces quickly. The generator reads the XTP and produces a compiling sequence and test for every row in a few hours rather than weeks. The boilerplate, the plumbing, and the compile loop no longer fall to the engineer.

AI does not replace the verification engineer. It removes the first-level work, the typing and the compiling, so the engineer's

time goes to the questions that require judgment: whether the stimulus is good, and whether the execution actually exercises the design.

The book follows this pattern at every stage. The Verification Strategist uses AI to collapse the mechanical first pass and spend the saved time on judgment. In Chap. 2 that first pass was extracting features and drafting the test plan; in Chap. 3 it was generating the testbench scaffolding; here it is writing the scenarios. The flow stays the same, the tool changes with the stage, and the human governs every output. The engineer can begin asking whether the stimulus exercises the design's decision points and whether the checks are meaningful from the start, instead of after weeks of plumbing (Fig. 4.1).



The same move recurs at every stage: AI does the first pass, the Verification Strategist governs the result.

Fig. 4.1 Where AI changes the economics of test writing. Manually, weeks of first-level coding stand between the plan and any real verification thinking. AI generates and compiles the suite in hours, so the engineer's time lands directly on stimulus quality and execution. The same strategist move recurs at every stage of the flow

4.4 The XTP as the Test Contract

The Executable Test Plan from Chapter 2 is not documentation. It is a contract between the verification architect and the design team.

Every test traces to an XTP row. Every XTP row traces to a specification requirement. Tests without XTP rows are uncontrolled scope creep. XTP rows without tests are unverified requirements.

The bridge XTP organizes its rows into eight categories. Table [4.1](#) shows what each category is responsible for verifying. (The exact per-category counts for this design are given later, in the Result section; here the point is the structure, not the numbers.) The categories instantiate the canonical set from Chap. [2](#) for this design: the bridge plan carries its stress-class rows inside Cross-Feature, where load combines with faults, and adds Bringup/Init as the readiness category every regression runs first.

Each row in the XTP provides a test ID, a test name, a one-line objective, the signals under test, the stimulus description, the expected result, and the specification page reference. This is exactly the information needed to generate an executable UVM scenario. The discipline is to use it, to start from the XTP row and not from a blank file.

Table 4.1 Bridge XTP categories and what each one verifies

4.5 Stimulus: What It Means to Drive a Design

Stimulus is the input the testbench delivers to the DUT in order to exercise a specific behavior. Understanding what good stimulus looks like, and what it does not, determines whether a test suite finds bugs or misses them.

4.5.1 The Four Scenario Classes

Every test in the bridge XTP belongs to one of four scenario classes. The class determines how the test is written, what kind of stimulus it uses, and what a meaningful pass looks like.

Functional scenarios verify that the design does what the spec describes under normal operating conditions. A single AHB write produces a single APB write at the correct address and with correct data. These are the baseline tests. If a functional test fails, something fundamental is broken. Functional tests must pass before any other category is meaningful.

Corner-case scenarios target boundary conditions: maximum address values, minimum burst lengths, the exact timeout boundary, the first and last valid register addresses. These are directed tests. You are not exploring; you are confirming that one specific boundary condition is handled correctly. The bridge’s AHB address range is 0x0000_0000 to 0x0000_FFFF. A corner-case test checks what happens at exactly 0x0000_FFFF and exactly 0x0001_0000.

Stress scenarios drive the design at maximum load: back-to-back transfers with no idle cycles, concurrent accesses, sustained bursts for thousands of transactions. Stress tests expose ordering failures, backpressure bugs, and counter rollover issues that never appear in low-volume functional tests. When a plan calls for them, a stress scenario drives a long run of consecutive back-to-back transfers, a condition that immediately exposes any state the design fails to clear between transactions.

Error scenarios inject conditions the design should reject: invalid addresses, illegal protocol sequences, fault injection via PSLVERR. These tests verify not just that the design handles errors, but that it handles them correctly: asserting HRESP, capturing ERROR_ADDR, clearing STATUS flags, and recovering to a functional state. An error test that passes because no error was detected has verified nothing.

Figure 4.2 summarizes the four classes, and Table 4.2 maps the bridge XTP categories onto them.

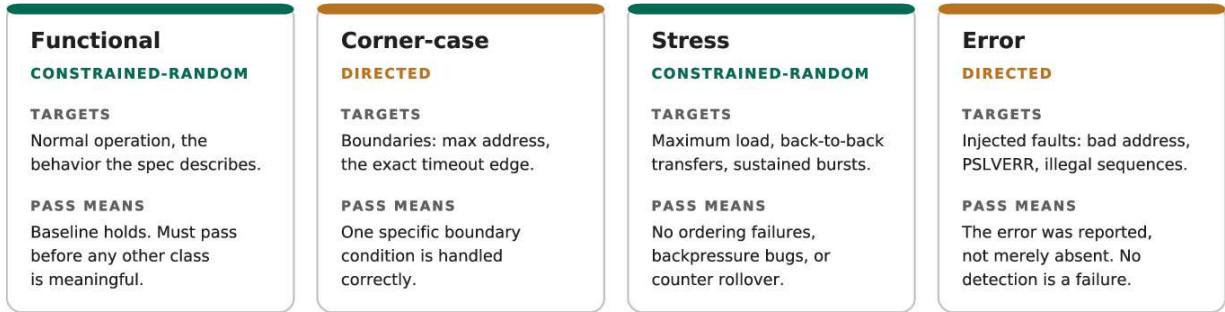


Fig. 4.2 The four scenario classes. The class determines the stimulus style and what a meaningful pass requires. An error scenario that detects no error has verified nothing

Table 4.2 XTP categories mapped to scenario classes

4.5.2 Stimulus Quality: Decision Points

Good stimulus exercises the design's decision points, the conditions where the RTL chooses between two behaviors. For the bridge, the key decision points are:

- HTRANS: IDLE, NONSEQ, SEQ, BUSY, each causes a different bridge response
- HWRITE: read vs write determines the APB direction
- PREADY: 0 extends the APB transfer; 1 completes it
- PSLVERR: 0 is normal; 1 triggers error propagation
- TIMEOUT_EN + TIMEOUT_VAL: determines when the bridge gives up on a stalled APB slave
- Address range: inside valid APB space vs outside

A test suite that never exercises BUSY, never sets PSLVERR, and always uses addresses in the middle of the valid range is not a verification suite. It is a collection of repeated confirmations of the same behavior. This is precisely the judgment AI augmentation frees the engineer to make: not whether the scenario compiles, but whether its stimulus reaches the decision points. Functional coverage measures this, and it is the subject of Chap. [8](#).

4.6 Randomization: Exploration Versus Confirmation

Two tools exist for generating stimulus: directed and constrained-random. They have different jobs. Using the wrong tool for a scenario produces tests that look correct but miss bugs. Figure [4.3](#) contrasts the two.

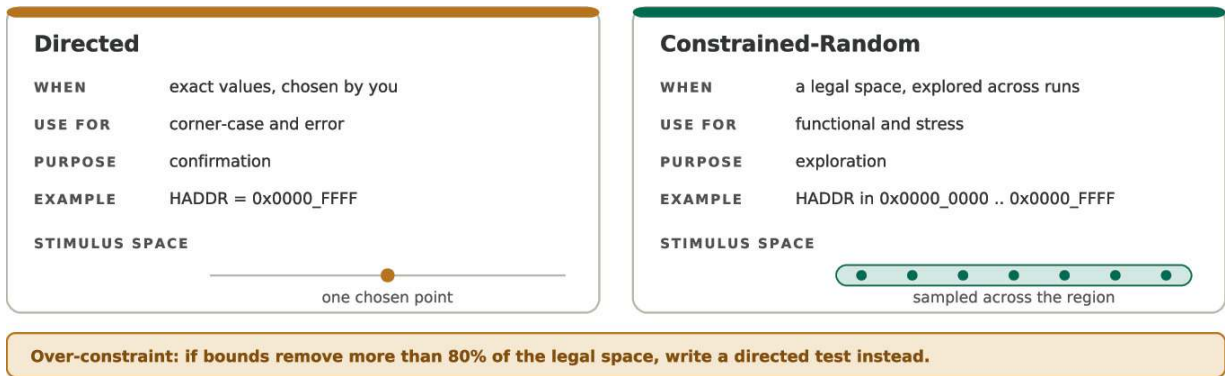


Fig. 4.3 Directed confirms a condition you already know; constrained-random explores a legal space the solver searches. Over-constraining a random test until it is effectively directed defeats the purpose

4.6.1 Directed Tests

A directed test specifies exact stimulus values. The address is 0x0000_FFFF. The transfer type is BUSY. PSLVERR asserts on the third APB transfer. These values are not chosen by the solver; they are chosen by the verification engineer because that specific combination exposes a specific behavior.

Corner-case and error scenarios are almost always directed. The engineer has identified a specific condition to exercise and wants to verify that one thing. Introducing randomization into a scenario that targets a specific boundary condition makes the test unreliable: sometimes it hits the boundary, sometimes it does not.

4.6.2 Constrained-Random Tests

A constrained-random test specifies a legal stimulus space and lets generation explore it. The address is anywhere in the valid APB range. The transfer type is any legal HTRANS value. The read data is any 32-bit value. Constraints enforce legality; randomization drives coverage.

Functional and stress scenarios benefit from this because the design must work correctly across all legal inputs, not just the specific ones the verification engineer thought of. A functional test that always uses address 0x0000_1000 and data 0xDEAD_BEEF confirms one specific combination. A randomized functional test covers thousands of address/data combinations across multiple runs.

4.6.3 The Over-Constraint Failure Mode

The most common randomization mistake is over-constraining a scenario until it is effectively directed, then wondering why functional coverage is not closing.

If your constraints eliminate more than 80% of the legal stimulus space, write a directed test instead. Randomized stimulus is only valuable when there is room to explore.

For the bridge, the correct boundaries are:

- **Address:** bound to the valid APB range `0x0000_0000` to `0x0000_FFFF`. This is a legality bound, not a coverage restriction.
- **HWRITE:** leave free. Both read and write are legal and both must be verified.
- **HTRANS:** leave free within legal values. Let generation produce IDLE, NONSEQ, and SEQ combinations.
- **Data values:** leave completely free. The scoreboard checks forwarding correctness regardless of data content.
- **Transaction count in stress:** configure as a test parameter, not a stimulus bound.

4.7 Anatomy of a UVM Test

The testbench from Chap. [3](#) provides two working tests, a bringup test and a sanity test. The structure they use is the template for every scenario in this chapter, and it rests on one rule: every VEGA scenario is exactly two components, always separate.

The **sequence** generates stimulus. It lives in `sequences/`, knows the protocol signals, and knows nothing about which test started it or what the verification objective is. The **test** configures the environment, starts sequences, and defines how the simulation ends. It lives in `tests/`, knows the verification objective, and knows nothing about protocol signal details.

This separation is the governing rule from Chap. 3 applied one level down. A sequence written for one test can be reused in another with different configuration. The scoreboard logic in the environment never changes regardless of which test is running. Infrastructure stays fixed while scenario logic varies freely.

The checking is already in place. The scoreboard generated in Chap. 3 carries six checkers, one per verification goal (G1 through G6), and fires on every transaction. A new scenario does not configure the scoreboard or duplicate its logic. It simply drives stimulus that exercises those checks. The infrastructure already knows what to check; the scenario only decides what to drive. This is why the generator can add scenarios safely: it writes only the two files that vary and never touches the checking layer.

4.8 What AI Gets Wrong in Test Generation

Test generation failures are less frequent than testbench generation failures, but they are more dangerous. A testbench compile error stops the simulation. A wrong test passes the simulation while verifying nothing. These are the patterns the engineer reviews for once the generator has removed the mechanical work.

Randomizing the error condition. Error scenarios must be directed. AI will sometimes generate a randomized sequence for a fault injection scenario, producing a test that sometimes injects the fault and sometimes does not. Diagnosis: the test passes on most runs but occasionally fails with no obvious cause. Every error scenario must have fixed fault injection values.

Reactive slave not in fork. The APB slave sequence is started with a direct `.start()` call before the AHB master sequence instead of in a concurrent fork branch. The slave runs one response, finishes, and exits. The bridge initiates a second APB access and waits forever for `READY`. Diagnosis: simulation reaches a fixed timestamp on any test with more than one transaction and stops making progress.

Expected values computed before stimulus. The sequence pre-calculates the expected APB address from the AHB address and stores it. The test reads this pre-calculated value to verify the scoreboard. This is circular: the sequence is checking its own calculation, not the DUT's forwarding. The scoreboard already checks G1 through G6. The test should not duplicate this logic or, worse, override it with a pre-calculated value that cannot detect forwarding errors.

Infrastructure modification. A generated scenario sometimes adds a new monitor output port, a new scoreboard checker, or a change to the APB slave driver to support fault injection. Any of these violates the infrastructure boundary from Chap. 3. If fault injection requires a change to the APB slave driver, the driver was not built correctly when the testbench was generated. The change must be flagged and stopped, not absorbed into the scenario layer: go back to Chap. 3, update the manifest and `verification_intent.txt` to include the missing capability, and regenerate.

Infrastructure bugs surface as test failures. When a clocking block samples on the same edge that a driver is asserting, every test fails with timing-related undefined values. This looks like a DUT bug; the first instinct is to check the RTL. The diagnostic signature is that multiple unrelated tests fail at the same point in the same way. A DUT bug produces failures that are specific to the stimulus that triggered them. An infrastructure bug produces failures that are consistent across all stimulus that reaches the affected interface.

When this pattern appears, stop. Do not modify the scenario. Do not add delays to the sequence. Return to the Chap 3 infrastructure and fix the clocking block, driver, or monitor. Regenerate if needed. Patching an infrastructure bug from the scenario layer hides the root cause and produces a test suite that is fragile in ways that will not become visible until a different test exercises the same path differently.

4.9 Scenario Readiness: Checks Before Regression

Before a scenario enters the regression suite, it must pass a set of checks. A scenario that fails any of them does not belong in regression. Two of these checks can be confirmed at generation time, when the scenario is written and compiled; the other two are confirmed as the scenario is run, which is the debug work of Chap. 5. The standard is stated here in full because the generator enforces what it can and flags the rest.

Check 1: XTP traceability (generation time). Does this scenario trace to an XTP row? Every scenario file must reference an XTP test ID in a comment at the top of the test class. If no XTP row exists for this scenario, either the scenario is scope creep, in which case remove it, or the test plan is incomplete, in which case add the row with architect sign-off before the scenario enters regression.

Check 2: Structural compliance (generation time). The scenario is exactly one sequence and one test, and it modifies no infrastructure file (driver, monitor, environment, interface, scoreboard, or `tb_top`). These are the boundaries from Chap. 3, and they are checkable the moment the files are written.

Check 3: Intent alignment (run time, Chap. 5). Does the scenario actually exercise what the XTP row says it should? Run the scenario, read the simulation log, find the scoreboard output, and confirm that the right checkers fired on the right transactions. A scenario that exercises a different behavior than its XTP row describes produces a false sense of coverage.

Check 4: Observable pass/fail oracle (run time, Chap. 5). Does the scenario produce a definitive PASS or FAIL verdict? A simulation that ends with `UVM_INFO: Test completed` and no scoreboard output is not a test. The log must contain at minimum how many transactions were checked, how many passed, and the final verdict. The sanity test log from Chap. 3 produces:

```
[SCOREBOARD] Checked=5 Pass=5 Fail=0  G1-G6 all PASS
[TEST] *** TEST PASSED ***
```

Every new scenario must produce an equivalent verdict line once it is run. Any FAIL, VIOLATION, or AMBIGUITY sends the scenario back to be fixed rather than forward into regression.

4.10 Generating the Suite with `vega_test_generator`

Everything to this point is the method: what a test is, what its stimulus must exercise, how it is structured, what goes wrong, and what makes it ready. The VEGA framework applies that method with one batch script, `vega_test_generator`, so the engineer does not hand-code each scenario. Figure 4.4 shows the path from an XTP row to a compiling, committed scenario.

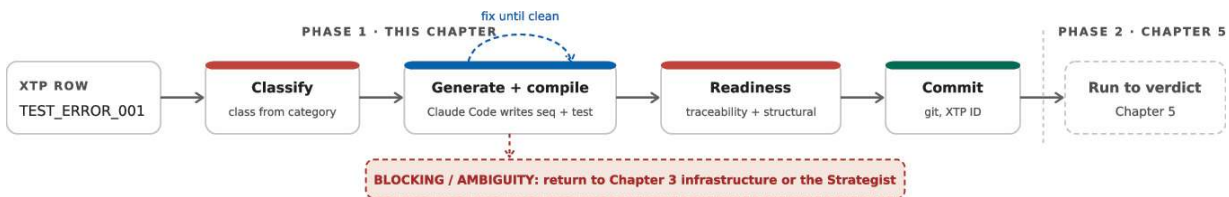


Fig. 4.4 From XTP row to committed scenario. The Verification Strategist owns classification and review; `vega_test_generator` drives generation and compilation. A BLOCKING flag returns the work to the Chapter 3 infrastructure rather than working around it. The chapter ends at compile-clean; running each scenario to a verdict is Chap. 5

4.10.1 The Generator

`vega_test_generator` is the Chap. 4 counterpart to the XTP generator from Chap. 2 and the testbench generator from Chap. 3: one governed tool for one stage. It reads the XTP and, for every row, drives Claude Code to write a sequence and a test and to compile them clean, generating the whole suite in a single batch run. This is Phase 1, coding. Running each scenario to a passing verdict, Phase 2, is the subject of Chap. 5.

The per-class instructions are encoded inside the tool and selected automatically from each row's category using the mapping in Table 4.2. Classification stays a Verification Strategist decision, expressed once in

the XTP, rather than an AI guess made at generation time. For each row the generator enforces a fixed set of rules:

- One sequence class and one test class per XTP row. The sequence carries stimulus only; the test configures the environment, starts sequences, and owns the objection. Neither crosses into the other's job.
- Generated files land in the testbench's `sequences/` and `tests/` directories. No infrastructure file (driver, monitor, scoreboard, env, interface, `tb_top`) is ever modified.
- Error and corner-case scenarios are directed, with fixed stimulus values. Functional and stress scenarios are randomized within the legal space, following Sect. [4.6](#).
- It runs the generation-time readiness checks from Sect. [4.9](#) and commits each scenario with its XTP ID.
- If a row needs an infrastructure change, it emits `BLOCKING` and stops. If a row's expected response is not stated clearly enough to check, it emits `AMBIGUITY` and skips the row. It works around neither.

A run is a single command. The XTP is the one from Chap. [2](#); the testbench is the one from Chap. [3](#):

```
python ch01_vega_tools/vega_test_generator.py \
    --xtp ch02_testplan/ahb2apb_testplan.xtp \
    --tb-dir ch03_testbench/ahb2apb_bridge_uvmtb \
    --category error
```

Omit `--category` to generate every row in the XTP. Per row the tool reports the XTP ID being traced, the goal checker (G1 to G6) that fires on its eventual pass, the files written, and any `BLOCKING` or `AMBIGUITY` flag. When the run finishes it prints a summary table, every row with its generation verdict, which is the management visibility artifact from Sect. [4.2](#).

4.10.2 From an XTP Row to a Scenario

To make the input concrete, here are three rows from the Error Handling category and what the generator must get right for each:

- TEST_ERROR_001, *address_error_detection*: verify HRESP=ERROR and error capture when HADDR is outside the valid range.
- TEST_ERROR_003, *timeout_error_detection*: verify timeout error detection when the APB slave holds PREADY=0 beyond the configured limit.
- TEST_ERROR_006, *protocol_error_detection*: verify the bridge detects PSLVERR and generates a protocol error response.

Each row becomes one sequence/test pair. The key elements the generator must get right:

For TEST_ERROR_001: the address must be set to a value **outside** 0x0000_FFFF, not inside. The test must then check that HRESP asserted and that the ERROR_ADDR register captured the failing address. This requires register reads after the error; the test class orchestrates this sequence.

For TEST_ERROR_003: the APB slave sequence must hold PREADY=0 for more cycles than TIMEOUT_VAL specifies. This is a directed stimulus condition. The test class reads the STATUS register after the timeout and confirms the error bit is set.

For TEST_ERROR_006: the APB slave sequence asserts PSLVERR=1 on a specific transfer. The test confirms HRESP=1 (ERROR) on the AHB side. This exercises G5 in the scoreboard directly.

4.11 Result: Generating the Bridge Suite

The sections above are the method. This is what a Phase 1 run produces against the bridge XTP and the Chap. [3](#) testbench.

AI Output Varies Between Runs

The numbers and the specific generation events below come from running this flow on the bridge. Large language models are not deterministic, so your run will differ in the details, which row surfaces an issue first, how many fix iterations a scenario takes. What does not change is the shape: one sequence and one test per XTP row, compiled clean, with the infrastructure left untouched.

The bridge XTP carries 96 rows across the eight categories of Table [4.1](#). Table [4.3](#) gives the per-category breakdown, the only place in this chapter the hard counts appear.

Table 4.3 Bridge XTP test counts by category (Phase 1 input)

The generator started with the Error Handling category and the first row, `TEST_ERROR_001`. It generated the sequence and test class and compiled them clean. The very first run of that scenario, however, surfaced something the bringup and sanity tests had not: the scoreboard read `PRDATA` as `0xFFFFFFFF`, the undefined-value signature of a clocking-block sampling race in the Chap. [3](#) infrastructure, where the monitor sampled on the same edge the driver asserted.

This is exactly the situation Sect. [4.8](#) warns about: an infrastructure bug surfacing as a scenario failure. The correct response was not to patch the scenario but to return to the Chap. [3](#) layer and fix the sampling point once, in the interface, so every scenario inherits the fix. With that single infrastructure correction in place, generation continued.

From there the generator was turned loose under one standing rule: any row that required a further infrastructure change would be flagged `BLOCKING` and stopped, not worked around. Working category by category, it produced one sequence and one test per XTP row and compiled each clean. The end state of Phase 1 is the whole plan

rendered as compiling scenarios: every row in Table [4.3](#) has a sequence/test pair, each traced to its XTP ID and committed.

Phase 1 endpoint. Every XTP row has a generated, compiling sequence/test pair, with the Chap. [3](#) infrastructure unmodified except for the one shared sampling fix. What none of these scenarios has done yet is run to a verdict. Driving them to `TEST PASSED`, and the real DUT bugs that surface when they do, is Phase 2, Chap. [5](#).

That endpoint is what the whole structure is built to reach quickly. The XTP provides the input. The generator provides the structure and the governance. The Verification Strategist provides the infrastructure decisions that AI correctly refuses to make unilaterally. In return, AI provides speed: a plan that would take weeks to write by hand becomes a compiling suite in hours, leaving the engineer free to judge the stimulus and the checks.

4.12 Conclusion

A simulation that runs without errors has not verified anything. A test drives a design, checks the result against a specification requirement, and produces a definitive verdict that is traceable, repeatable, and defensible.

This chapter turned the XTP into the scaffolding for that evidence, using AI to do the first-level work. The four scenario classes decide how each row is written; directed and constrained-random decide how its stimulus is generated; the readiness checks decide whether it may enter regression. `vega_test_generator` applies those decisions row by row without touching the Chap. [3](#) infrastructure, generating and compiling the whole suite far faster than hand-coding would allow. The engineer's time then goes to the quality of the stimulus and the meaning of the checks.

This is the method the book applies at every stage of verification: let AI do the mechanical first pass, and keep human judgment on the

decisions that determine whether the work is sound.

Phase 1 ends with the suite generated and compiling. Phase 2 begins when those scenarios run. Chapter [5](#) covers it: how to diagnose failures efficiently using waveforms and simulation logs, how to distinguish DUT bugs from testbench bugs, and how to drive every scenario to a verified verdict.

5. Debug, Failure Analysis, and Regression Management

Vikash Kumar¹ 

(¹) Leander, TX, USA

 **Vikash Kumar**

Email: vikash.singh261@gmail.com

Running a generated scenario suite produces failures, and that is the point: a regression that fails nothing has probably exercised nothing. What wastes time is not the failures but the unstructured response to them. This chapter gives debug a structure and uses AI to accelerate its mechanical steps. Its governing principle is to classify before investigating: every regression failure is a design bug, a test-logic issue, an infrastructure issue, or an environment artifact, and the class dictates the first action. Supporting disciplines follow: reading the structured log before the waveform, debugging only the first failure, and expressing every failure as an expected/observed/deviation triple. A representative eight-hour bridge debug session, in which four of five failure modes were testbench issues and only the last was a real RTL bug, is walked as a classification exercise. The chapter covers minimal reproduction, anchoring each failure to its plan row, the four deviation types, timeout debug reframed as a missing-observability problem, and regression triage. Its deeper claim is that every failure is feedback on the test plan, so a program that debugs this way does not just close tickets, it learns.

5.1 The Problem Regression Creates

Chapter [4](#) ended at a precise endpoint. Every XTP row had a generated sequence and test, each one compiled clean, and the whole suite sat in the testbench ready to run. That was Phase 1, coding. It produced scaffolding that compiles. It did not produce a single verdict.

This chapter is Phase 2. The scenarios run, and failures appear. This is expected. This is the point. A regression that produces no failures on its first full run has almost certainly not exercised the design, and the absence of failures is itself a finding worth investigating.

The problem is not that failures exist. The problem is what happens next in most verification programs. Each failure opens a new investigation with no shared structure. Engineers form hypotheses before they have classified the failure, open waveforms before they have read the log, and debug signal activity before they have verified the transaction count. By week three, the same root causes are being diagnosed multiple times under different symptoms. Senior engineers are spending the majority of their time on a minority of the actual problems.

This chapter gives debug structure, and it uses AI to accelerate every mechanical step inside that structure. It does not reduce the number of bugs in the design. It reduces the time spent finding them, and it ensures that every bug found improves the verification program rather than simply closing a ticket.

5.1.1 Where AI Accelerates Debug

Debug has two halves. One half is judgment: deciding which layer a failure belongs to, whether a deviation matters, whether the test plan itself was wrong. The other half is mechanical: reading a thousand-line log to find the first error, expressing a failure as a precise deviation, grouping twenty failures by symptom, running a fix and re-running the regression to confirm it. The mechanical half is most of the elapsed time, and it is exactly what AI does quickly.

AI does not decide what a failure means. It does the mechanical work that surrounds the decision: it reads the log and extracts the first error, drafts the deviation precisely, clusters a regression's failures by likely root cause, and drives the fix and re-run loop. The Verification Strategist keeps the judgment: classification, intent, and sign-off.

Because a model left to its own devices will drift, fabricate a cause, or declare victory on a wrong fix, the engineer's job is to bound it. The structures in this chapter are those bounds: each hands AI a narrow task with a fixed output and an explicit rule on what it may not do. The guardrails below apply to every AI step that follows.

Guardrails: keeping a deviation-prone AI on the goal

- Hand AI a bounded task with a fixed output contract. Never "debug this."
- Classify first; AI drafts only *within* the class. It may not cross a layer: no infrastructure edits while fixing a test.
- Never let AI propose the root cause and the fix in one step. A fix aimed at the wrong cause is worse than no fix.
- One change, then re-baseline. The model's confidence is not evidence; the regression is.
- The engineer owns classification, intent, and sign-off. AI owns the mechanical fill, nothing more.

This is the same division the book applies at every stage. AI removes the first-level work so the engineer's time goes to the decisions that require understanding the design. In debug, that means a session that took a senior engineer a full day of log-reading and waveform-chasing becomes a few hours of decisions, each one handed the evidence it needs.

A representative debug session for the bridge, the Educational Bridge Model specified in Appendix [A](#), serves as the worked example throughout this chapter. That session took eight hours and produced five distinct failure modes before a real RTL bug was found. Applied with the structure in this chapter, and with AI doing the mechanical work at each step, the same session compresses to under three hours. The difference is not cleverness. It is process.

A Representative Session

The eight-hour timeline, the five failure modes, and the `xfer_pending` bug that follow are drawn from a single debug session on the bridge. They show the classification framework sending each failure to the layer that owns it. Run the design again, or a different design, or the same one under new seeds, and the failures will differ in kind and in order. What stays constant is the structure: the failure classes, the sequence of investigation, and the rule that each defect is fixed at its own layer.

The worked session is one instance. What transfers to any design is the structure, not the particular bugs. Table [5.1](#) separates the two: the left column is what an engineer reuses on every design; the right column is whatever the run happens to throw (Table [5.1](#)).

Table 5.1 What is invariant across designs, and what each run changes

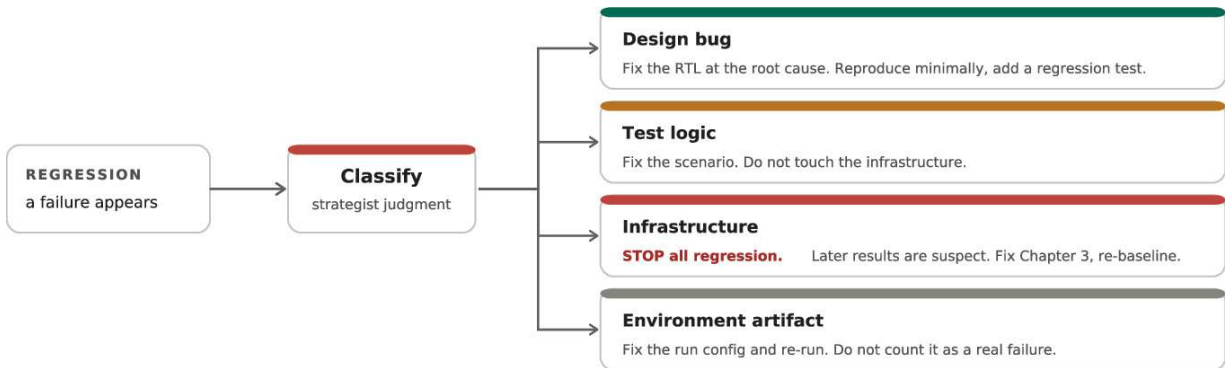


Fig. 5.1 Classify before you investigate. Every regression failure sorts into one of four categories, and the category dictates the first action. An infrastructure failure halts the regression; an environment artifact is not counted as a real failure. The classification, not the waveform, is the first step

5.2 The Governing Principle: Classify Before You Investigate

Before opening a waveform, before reading a log, before forming a hypothesis: classify the failure. The classification determines where you look. Looking before classifying wastes time.

Every regression failure belongs to exactly one of four categories. The category determines the next action completely. This classification is a Verification Strategist judgment call. It cannot be delegated to automated tooling, because the boundary between a test logic issue and an infrastructure issue requires understanding the intent behind both the scenario and the testbench. AI can propose a first-pass classification from the log, but the engineer owns the decision (Fig. [5.1](#) and Table [5.2](#)).

The stopping rule for infrastructure issues is not optional. Finding an infrastructure bug mid-regression means all test results from the broken environment may be invalid. Continuing to run scenarios against broken infrastructure is not debugging. It is generating noise. Stop. Fix. Re-baseline.

5.2.1 Flaky Failures Are Infrastructure Failures Until Proven Otherwise

A failure that appears on some random seeds and not others is almost never a design bug in protocol verification. It is almost always a race condition in the testbench: timing-dependent sampling, clocking block skew that manifests differently under different stimulus orderings, or a

reactive sequence that behaves correctly on average but fails under specific concurrency patterns.

The rule: before investigating a flaky failure as a design bug, run it with a fixed seed ten times. If it passes on some runs, the testbench has a race condition. Fix the infrastructure. Do not file a design bug against an RTL that is behaving consistently.

Table 5.2 Failure classification and first action

In the bridge session, a clocking-block timing defect threw intermittent read mismatches that could pass for timing-sensitive design behavior. They were nothing of the sort: an infrastructure failure, classified as such and fixed at that layer, with the RTL correct the whole time.

5.3 Log First, Waveform Second

Engineers open the waveform immediately. Waveforms are visually compelling and feel productive. They are also the slowest way to find a bug in a UVM environment.

UVM logs contain structured information that narrows the search space in minutes: transaction records from monitors, scoreboard comparison results, assertion fire messages, and UVM_ERROR messages with timestamps. A waveform contains all of that information buried in thousands of signal transitions. Reading the log first is also the step AI accelerates most directly: a model can scan the full log, surface the first UVM_ERROR with its timestamp, and extract the offending transaction in seconds (Fig. 5.2).

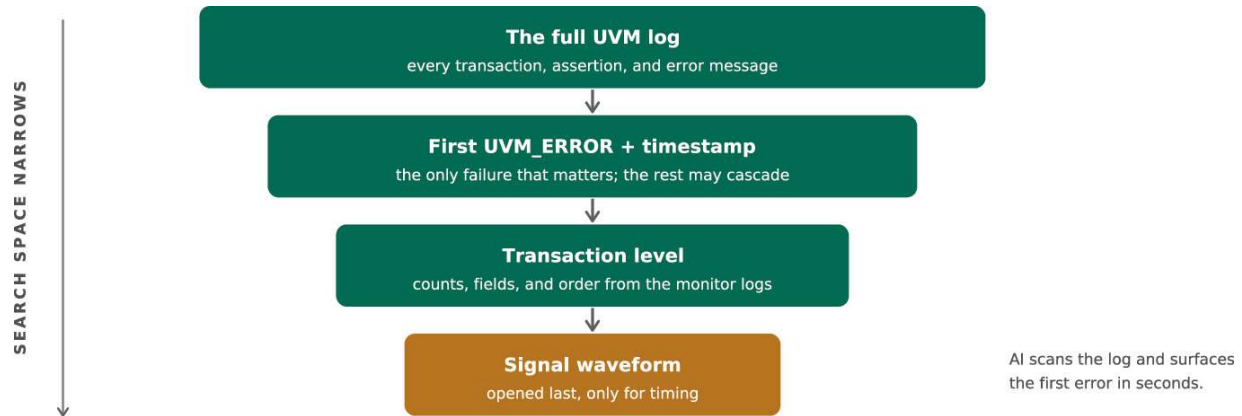


Fig. 5.2 The debug funnel. Start at the structured UVM log, narrow to the first error and the failing transaction, and only then open the waveform, at the failure timestamp and only for timing questions. Each stage shrinks the search space before the slower tool is reached

The discipline: read the full UVM log before opening the waveform. Identify the first UVM_ERROR and its timestamp. If the failure is a scoreboard mismatch, identify the transaction ID and the field that mismatched. If the failure is a simulation timeout, identify the last successful transaction before progress stopped, not what was happening at the timeout, but the last event that completed normally. Then open the waveform at that timestamp minus five clock cycles.

5.3.1 The First Failure Is the Only Failure

A simulation that produces eight UVM_ERROR messages has one failure that matters: the first one. Every subsequent error may be a consequence of the first. Scoreboard mismatches cascade. A single wrong transaction produces wrong expected values for every subsequent comparison. An assertion fire can corrupt state in a way that causes five later errors.

The discipline: find the first UVM_ERROR by timestamp. Debug only that error. Fix it. Re-run. Examine what survives.

In the bridge session, the first run that simulated to completion had the scoreboard reporting zero checks, not a mismatch but no checking at all. That was the one failure worth chasing, and the empty check count pointed straight at the analysis-port connection rather than at any protocol behavior.

5.3.2 Assertion-First Triage

When the testbench includes SVA assertions (placed in bind files per Chap. 3), assertion fires are more informative than scoreboard mismatches. Scoreboard mismatches tell you the end result was wrong. Assertion fires tell you exactly which clock cycle the protocol contract was broken.

Check assertion output before scoreboard output. The assertion gives you a timestamp and a condition. The scoreboard gives you a consequence. Debug from cause, not from consequence.

5.4 The Expected/Observed/Deviation Triple

Before any investigation begins, every failure must be expressed as three lines. This is not documentation overhead. It is the act of understanding the failure precisely enough to debug it. It is also a task AI does well from the log and the XTP row, drafting the triple for the engineer to confirm.

Expected: what the XTP row says should happen.

Observed: what the simulation log shows happened.
--

Deviation: the specific, measurable gap between them.
--

Engineers who cannot write these three lines cleanly do not yet understand their failure well enough to debug it. “The test failed” is not a deviation. “Expected 5 APB accesses, observed 6, deviation is one phantom access occurring after the last AHB transfer completed” is a deviation.

Take the bridge’s xfer_pending bug:

- **Expected:** 5 AHB writes yield exactly 5 APB accesses (cross-feature XTP row TEST_CROSS_FEATURE_004)

- **Observed:** 5 AHB writes yielded 6 APB accesses; the scoreboard flagged 1 unmatched APB transaction
- **Deviation:** a transaction-count error; one phantom APB access raised after the final AHB transfer had completed; the deviation type is count/ordering at the protocol boundary

Writing that triple takes two minutes, and it rules out every hypothesis that does not account for a count deviation at the AHB-to-APB boundary. The search narrows from “protocol timing is wrong somewhere” to “something is emitting one extra APB transaction.”

5.4.1 Transaction-Level Before Signal-Level

Once the deviation triple is written, begin investigation at the transaction level, not the signal level.

Transaction-level questions first:

1. Did the correct number of transactions occur on each interface?
2. Did each transaction carry the correct field values (address, data, control)?
3. Did transactions arrive in the correct order?

If any of these answers is no, the problem is structural. Open the monitor transaction log, not the waveform. The monitor observed the protocol behavior at the transaction level. The waveform will confirm what the monitor already told you.

Only after transaction-level analysis is exhausted should signal-level waveform investigation begin. Signal-level debug is the right tool for timing problems: clock edge alignment, setup and hold violations, registered signal clearing sequences. It is the wrong tool for protocol semantic problems.

For the `xfer_pending` bug the transaction count was wrong, six against five, which is structural. The APB monitor log showed six PSEL assertions, and the waveform confirmed the sixth landed one cycle after

the fifth transfer finished, with no matching HSEL on the AHB side. That was enough to file the report. The RTL root cause, `xfer_pending` clearing one cycle too late, came from reading the FSM always block, not from more waveform work.

5.5 The Bridge Debug Session as a Classification Exercise

Seen through the classification framework, the eight-hour bridge session shows one clear pattern: four of its five distinct failure modes were testbench issues, not design bugs, and the first genuine design bug did not surface until hour seven. Without the discipline of classifying first, an engineer could have burned hours probing the wrong layer on each of those first four failures (Fig. 5.3).

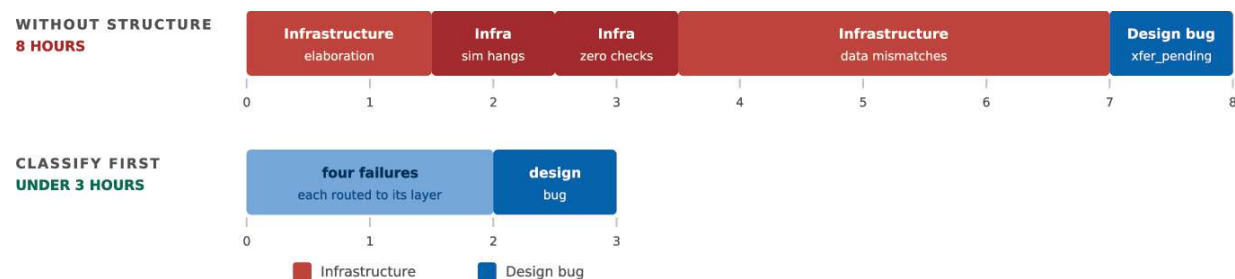


Fig. 5.3 The bridge debug session as a classification exercise. Four of the five failure modes were in the testbench; the only design bug appeared in the final hour. Classification-first routes each failure straight to its layer, compressing the session from eight hours to under three

Hours 1–2: Elaboration crashes and invented field names

Classification: **Infrastructure issue.**

The monitor reached `vif.mon_cb.HRESETn`, a port rather than a clocking-block member, which the compiler waved through because virtual-interface member resolution happens at runtime. On top of that, the generated sequence item named fields (`size_type`, `rw_type`) the package never declared.

Action: stop and repair the testbench. Do not simulate until the environment compiles and elaborates cleanly against the quality gate from Chap. 3. This is neither a scenario problem nor a design problem.

Hours 2–3: Simulation hangs after first transfer

Classification: **Infrastructure issue.** Two separate root causes, both in infrastructure.

Root cause 1: the APB slave sequence looped a fixed ten times and then stopped, where a reactive sequence should run `forever`.

Chapter [3](#) named this the most common infrastructure error, and here it was, exactly as predicted.

Root cause 2: the test used `fork...join`. Because the reactive slave never returns, `join` never completes. That is a test-architecture error, not a scenario error.

Action: set the reactive slave sequence to `forever` and switch the test to `fork...join_any` with `disable fork`. Re-run the Chap. [3](#) quality gate before going on.

Hours 3–4: Scoreboard reports zero checks

Classification: **Infrastructure issue.**

The scoreboard saw AHB transactions but no APB transactions: the APB monitor's analysis port was never connected in the environment. That is wiring inside an infrastructure file, not protocol behavior and not scenario code.

Action: connect the port in the environment, a one-line change that takes five minutes to locate and thirty seconds to apply. Because it is an infrastructure fix, re-run the Chap. [3](#) quality gate before trusting any later results.

Hours 4–7: Read and write data mismatches

Classification: **Infrastructure issue.**

The generated drivers and monitors got both clocking-block sampling timing and AHB pipeline data sequencing wrong. The giveaway was the signature: a mismatch value that stayed constant across all stimulus and tracked no particular address or data pattern, the hallmark of an infrastructure timing failure. Chapter [3](#) covers both issues in detail. The RTL was correct the whole time.

Action: fix the infrastructure and re-run the Chap. 3 quality gate before continuing. Do not paper over timing issues from the scenario layer.

Hours 7–8: Phantom APB transfer

Classification: **Design bug.**

The first real design finding in eight hours. Five AHB writes produced six APB accesses: `xfer_pending` cleared one cycle after the FSM had read it in the IDLE state, so the bridge re-entered `ST_APB_SETUP` spuriously.

Action: file the bug with its expected/observed/deviation triple, write a directed regression test for the five-transfer sequence, fix the RTL, and verify. Update the XTP to confirm `TEST_CROSS_FEATURE_004` is now covered.

The lesson from the timeline: classifying first would have sent each of the first four failure modes straight to the right layer, with no time lost guessing whether the hang was a design FSM issue or whether the data mismatches were a protocol-compliance problem. Each failure classified, fixed at its layer, re-baselined, and the session moved on.

5.6 From Failure to Minimal Reproduction

Before investing time in root cause analysis, reduce the failing scenario to the minimum stimulus that reliably reproduces the failure. This is the most underrated skill in verification.

A failure that appears in a stress scenario with 500 transactions, reproduced in a directed test with 5 transactions, debugs in a fraction of the time. The minimal case also prevents the fix from being entangled with unrelated stimulus behavior.

Process:

1. Start with the failing scenario. Run it with a fixed seed to confirm it fails deterministically.
2. Reduce the transaction count by half. Re-run. Does it still fail?

3. If yes, reduce by half again. If no, you have bracketed the failure: it requires between $N/2$ and N transactions.
4. Disable parallel sequences one at a time while keeping the core stimulus. Does removing the concurrent sequence eliminate the failure? If yes, the failure is in the interaction, not in the core stimulus.
5. Stop when you have the smallest scenario that fails reliably.

The one-change rule: when making fixes, change exactly one thing at a time. If three changes are required, make them sequentially and re-run after each. Engineers who make three simultaneous changes and see the test pass have no idea which change fixed the bug, whether all three were necessary, or whether one of the three introduced a new issue that is not yet visible.

Differential debugging: when a test passes with N transactions and fails with $N+1$, the failure is in the $(N+1)$ th transaction or in how the design transitions after completing the N th. Run the passing case and capture the final state of every relevant register and interface signal. Run the failing case and capture state at the same point. The difference between those two states is where the bug lives.

Drive minimal reproduction by handing the model the structure below, not a worded prompt. You supply the three inputs; the contract fixes what it may return and the bounds it may not cross. The engineer keeps the call on whether the reduced case is genuinely minimal.

Output contract: minimal reproduction

You provide: the failing scenario, the first UVM_ERROR with its timestamp, and the expected/observed/deviation triple.

It must return:

- the minimal stimulus that still reproduces the deviation;
- the stimulus elements that can be removed;

- a fixed-seed recommendation for deterministic reproduction;
- an estimated minimal transaction count.

Bounds (the rails): do not modify infrastructure files; do not alter the fundamental stimulus class; if the failure is non-deterministic, flag it as a likely infrastructure race and stop.

5.7 Anchoring the Failure to Intent

Once a failure is classified as a design bug and a minimal reproduction exists, anchor it to the XTP before filing or fixing. This step takes five minutes and prevents two classes of waste: fixing the wrong behavior (because the XTP had an error), and fixing a behavior that was not in scope (because no XTP row covers it).

5.7.1 The XTP Lookup

For the minimal reproduction scenario, identify the XTP row that governs the expected behavior. The row gives you:

- The feature or interaction being exercised
- The precise expected result (the oracle)
- The priority of the bug
- Whether this behavior was explicitly planned for

If no XTP row governs the failing behavior, stop. You have found an XTP gap. Do not fix the design bug yet. First, determine whether the behavior is within scope. If it is, add the XTP row with architect review before proceeding. If it is not within scope, document the finding and defer it. Fixing design behavior that is not in the XTP is scope creep, not verification.

5.7.2 Intent Deviation Classification

With the XTP row identified, classify the deviation type. Four types cover the vast majority of protocol verification failures:

- **Count deviation:** the wrong number of transactions, accesses, or events occurred. The xfer_pending bug is a count deviation.
- **Value deviation:** a transaction occurred with the wrong field value (wrong address, wrong data, wrong control signal). Clocking block skew producing wrong HRDATA is a value deviation.
- **Ordering deviation:** transactions occurred in the wrong sequence, or a required handshake was skipped or reordered. For the bridge: HWDATA driven in the same cycle as HADDR violates the AHB pipeline ordering requirement; PWDATA showed the next transfer's data because the ordering was wrong in the driver.
- **Liveness deviation:** a required event never occurred. The PREADY-never-asserts hang is a liveness deviation. Simulation timeouts are almost always liveness deviations. The reactive slave fixed-loop failure produced exactly this: PREADY never asserted after transaction 10, simulation reached timeout, no UVM_FATAL, just silence.

Deviation type determines where in the RTL to look. Count deviations point to FSM state transitions and registered control signals. Value deviations point to datapath assignments and pipeline timing. Ordering deviations point to handshake logic and protocol sequencing. Liveness deviations point to stuck conditions: signals that should transition and do not.

A structured deviation report follows the same idea: give the model the contract, not a prompt. It expresses the gap against the plan and nothing more, so it cannot wander into guessing a cause or proposing a fix.

Output contract: intent deviation report

You provide: the XTP row, the deviation triple, the UVM log excerpt, and the minimal reproduction.

It must return: the XTP row ID; the Feature and Interaction IDs involved; expected behaviour (from the XTP oracle); observed behaviour (from the log); the deviation type, one of {count, value, ordering, liveness}; severity (architectural/functional/secondary);

the earliest timestamp of the deviation; the minimal reproduction; and a suggested RTL search area (FSM/datapath/handshake/clock domain) as a hypothesis only.

Bounds: express the deviation against the XTP oracle only; do not guess root cause; do not propose RTL fixes; if no XTP row covers the observed behaviour, output `XTP_GAP` and stop.

5.8 Debugging Timeouts: The Liveness Deviation

Of the four deviation types, liveness deviations are the hardest to debug. A count, value, or ordering deviation produces evidence: a wrong number, a wrong value, a wrong sequence. A liveness deviation produces silence. The simulation runs, time advances, nothing happens. Eventually a global timeout fires and the regression reports a failure whose only diagnostic content is a timestamp.

The reframing that solves timeout debug: a timeout is not a simulation failure, it is a missing-observability failure. The simulation ran for its time budget and reported what it could. What failed is the testbench's ability to distinguish forward progress from a stall, at the boundary where the stall actually occurred.

A single global timeout cannot make that distinction. It cannot tell the Verification Strategist whether the design hung or the testbench did. The fix, at the bridge scale, is to replace the one outer timeout with a small set of **localized watchdogs** scoped to specific protocol boundaries. For the bridge, the natural watchdog is on the APB handshake: if `PREADY` does not assert within the protocol's setup-plus-access window after `PSEL`, the watchdog fires and reports the boundary, the last transaction that completed, and the cycle count at which the bound was exceeded.

This is exactly the hours 2–3 hang from the bridge session. With a global timeout, the symptom is “simulation ran out of time.” With an APB handshake watchdog, the symptom is “PREADY never asserted after transaction 10,” and the monitor log simultaneously shows the reactive slave sequence had exited normally. The classification, an infrastructure issue fixed in the reactive slave, becomes direct rather than inferred, and it no longer depends on an engineer’s experience reading silent logs. The same AI prompt pattern used for the deviation report turns the watchdog output and the last-progress record into a liveness deviation report automatically.

At scale, this idea grows. On a subsystem or SoC regression, a stall can originate in any of several VIPs and propagate across boundaries before the simulation goes quiet. There, localized watchdogs are extended with transaction-identity propagation, distributed progress telemetry, and temporal correlation across layers, so a silent multi-VIP timeout becomes a located, attributable event. That scaling is the subject of Chap. [6](#); at the IP level, a few protocol-scoped watchdogs are enough.

5.9 Regression Triage: Order Matters

Before diagnosing individual failures, triage the regression as a whole. Running 50 scenarios, seeing 20 failures, and debugging them in random order is the conventional approach. It is also the slowest.

P0 first. Run P0 scenarios before P1 or P2. If any P0 fails, stop running lower-priority scenarios. A regression full of P1 and P2 failures while P0 is broken produces noise. The signal is buried. P0 failures always represent the highest-risk behaviors defined in the XTP. Fix them first.

Cluster before diagnosing. Before investigating any individual failure, group all failures by symptom pattern. Failures that share the same first UVM_ERROR message, the same deviation type, and the same XTP row family likely share a root cause. Fix one representative failure

from the largest cluster. Re-run all failures in that cluster. If they all pass, one root cause produced many visible symptoms. If some persist, the cluster has multiple root causes and needs further splitting.

This is where AI assistance provides the most time savings in the entire debug process. Hand the model one-line summaries of all failures and the clustering contract below. A regression with 20 failures and three root causes becomes visible in the clustering output in under a minute, work that manually means reading twenty logs and holding their patterns in your head at once.

Output contract: regression failure clustering

You provide: one line per failure—`test_name | seed | first_UVM_ERROR | timestamp`.

Cluster on three keys only: the first UVM_ERROR message, the XTP row family, and the deviation type—never the test name alone.

Per cluster, it must return: a cluster ID; the member failures (test, seed); the shared symptom pattern; the XTP row family if identifiable; the likely deviation type; the representative failure to debug first; and the cluster size as a percentage of total failures.

Bounds: do not guess a per-failure root cause; any failure that fits no cluster is returned as `ISOLATE` for separate investigation.

The baseline rule. Before any RTL change is integrated into the regression environment, run a full regression against the current RTL and record every failure. This is the baseline. After the RTL change, run regression again. Compare new failures against the baseline. Only failures that did not exist in the baseline are attributable to the RTL change. Engineers who skip this step spend hours debugging failures that were pre-existing before the change they made.

5.10 Fix, Verify, Update

Every confirmed design bug requires three steps before it is closed. Completing only the first step, fixing the RTL, is the most common

cause of regressions reintroducing old bugs. AI accelerates this loop in practice: Claude Code applies the fix, re-runs the minimal reproduction and the P0 regression, and reports what changed, while the engineer decides whether the fix is at the root cause.

Step 1: Fix the RTL at the root cause, not the manifestation.

The question to ask before applying any fix: does this fix prevent the bad behavior from occurring, or does it prevent the bad behavior from being observable? A fix that hides the symptom without correcting the root cause is not a fix. It is technical debt that will manifest differently in the next design revision.

For the `xfer_pending` bug: the manifestation was a sixth APB access. A counter that blocked the sixth access would have been a manifestation fix. Changing the `always_ff` block so that `xfer_pending` clears every cycle unless actively latching is the root cause fix.

Step 2: Verify the fix with the minimal reproduction.

The minimal reproduction scenario must pass against the fixed RTL. Additionally, run the full P0 regression. A fix that resolves one failure while introducing another is not a fix, it is a trade.

Step 3: Add a regression test and update the XTP.

Write a directed test that specifically targets the fixed behavior. The test must fail against the unfixed RTL and pass against the fixed RTL. This test becomes a permanent regression guard. If the same issue is reintroduced during a future feature addition, this test catches it immediately.

Then update the XTP. Specifically:

- If the bug was caught by an existing XTP row: mark the row as verified. The test plan worked. Record the bug as evidence of coverage closure for that row.
- If the bug revealed a gap in the XTP (a behavior not covered by any row): add the row. The regression taught you something the test plan did not know. This is how the XTP improves over time.

- If the bug revealed that an XTP expected result was wrong (the specification was misunderstood during planning): correct the row and file a specification clarification with the design team.

Workarounds. When a fix requires an RTL change that cannot happen on the current schedule, a workaround may be accepted temporarily. A workaround is legitimate only when it is documented with all four of the following: the root cause, the workaround description, the JIRA for the proper fix, and the expiry condition (the milestone or event at which the workaround must be replaced). A workaround without a JIRA is not a workaround. It is a hidden debt that the next engineer will discover as an unexplained test exclusion with no memory of why it exists.

5.11 Debug as XTP Validation

This is the idea that separates a verification program that learns from one that merely accumulates passing tests.

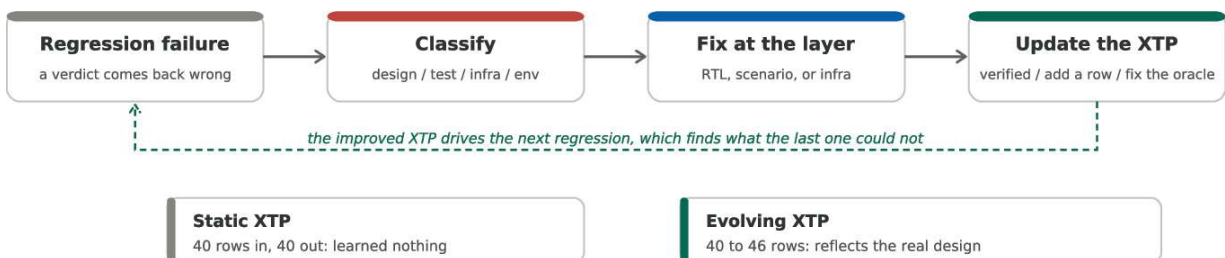


Fig. 5.4 Debug as XTP validation. Every failure feeds back into the test plan: a caught bug confirms a row, an uncovered bug adds a row, a wrong oracle corrects a row. The improved XTP drives the next regression, which finds what the last one could not. An XTP that ends a campaign unchanged is a plan that learned nothing

Every regression failure is a signal about the XTP. The failure is not just evidence of a bug in the design or testbench. It is evidence about the quality, completeness, and correctness of the test plan itself (Fig. [5.4](#) and Table [5.3](#)).

Table 5.3 What each failure type reveals about the XTP

The consequence of this framing is significant. An XTP that starts with 40 rows and ends with 40 rows after a full regression is a plan that learned nothing. An XTP that starts with 40 rows and ends with 46 rows, six new rows added because regression revealed behaviors that were not planned for, is a plan that reflects what the design actually does, not just what someone thought it would do when they wrote the spec.

By the time a verification program reaches sign-off, the XTP should be visibly different from what it was on day one. Every difference represents something the regression taught the team. That accumulated knowledge is the artifact that makes the next similar design faster to verify, not because the process was faster, but because the decisions embedded in the XTP do not need to be rediscovered.

5.12 Quality Gate: What Regression Clean Means

Regression clean is not zero failures. Regression clean is a defined state that the verification program can demonstrate and defend. The Verification Strategist signs off on this state, not the regression script, and not the coverage tool.

1. **All P0 scenarios pass.** No exceptions, no waivers for P0.
2. **All P1 scenarios pass, or have documented waivers.** Each waiver must reference a JIRA, a root cause, and a resolution plan. A waiver without these three elements is not a waiver. It is an unresolved failure with an alias.
3. **Every design bug found has been filed, fixed, and verified.** Verified means the minimal reproduction passes and P0 regression does not regress.
4. **Every XTP gap found during regression has been addressed.** Either a new XTP row was added and tested, or the gap was reviewed and explicitly deferred with documented justification

reviewed and explicitly deferred with documented justification.

5.

No new failures were introduced by any fix. Run full regression after every fix before closing the bug.

6.

Functional coverage is on track. Coverage closure criteria, covergroup definitions, and gap analysis are covered in Chap. [8](#).

7.

The XTP reflects what was learned. New rows added for gaps found. Incorrect rows corrected. Every change to the XTP reviewed and approved by the Verification Strategist.

When all seven checks pass, the IP-level regression is clean and the verification program is ready to scale. Chapter [6](#) covers subsystem verification, the level at which individually verified IPs are composed, and where a new class of failure appears that no IP-level regression can see: emergent behavior arising from interactions between components that are each independently correct.

5.13 Conclusion

The bridge debug session opened on an eight-hour timeline with five failure modes, four of them testbench issues and one a real design bug. Classifying first would have routed each of the first four straight to the right layer, with no hypothesis-spinning, no waveform chase of the wrong signal, and no time spent doubting RTL that was correct throughout. AI carried the mechanical load at every step, so the engineer's hours went to the decisions rather than the log-reading.

But the deeper finding is not about time. It is about what the regression left behind. The XTP that entered the bridge debug session had its full set of planned rows. The XTP that exits has more: rows added for behaviors the test plan did not anticipate, expected results corrected where the specification was ambiguous, a new regression guard for the xfer_pending boundary condition. The test plan is not the

same document it was on day one. It is better, because the design taught it something.

This is the verification program that learns. Every bug found either confirms the test plan worked or reveals a gap the test plan missed. Either outcome improves the XTP. The XTP improves the next regression. The next regression finds things the current one could not.

That compounding (intent structured in Chap. [2](#), testbench governed in Chap. [3](#), scenarios generated in Chap. [4](#), debug structured in Chap. [5](#)) is what the VEGA framework produces when it is applied end to end. Not a faster simulation run, but a verification program that gets demonstrably better as it executes.

Chapter [6](#) scales this to the subsystem level, where the governing artifact is no longer a single XTP but a composition of IP-level intents, and where correctness depends not on any individual component but on the contracts between them.

6. Subsystem Verification Through Intent Composition

Vikash Kumar¹ 

(¹) Leander, TX, USA

 **Vikash Kumar**

Email: vikash.singh261@gmail.com

A component that passes every IP-level test can still destroy a tapeout once integrated, because the bugs that matter at subsystem scope do not live inside components: they live in the contracts between them, the ordering assumptions, error-propagation chains, reset sequencing, and shared-resource arbitration that no single component's test plan owns. This chapter extends VEGA from IP to subsystem verification without introducing new tooling. It defines a protocol-agnostic ten-category taxonomy of integration intent, catalogs ten integration bug patterns IP-level VIPs structurally cannot see, and builds the Subsystem Intent Graph (SIG): a graph whose edges carry a guarantee from one component and an assumption from its neighbor, where every misalignment is a defect site visible before integration begins. An assumption harvest extracts those implicit contracts from existing IP-level artifacts, and a category-to-assertion mapping turns each into concrete SVA. Three new scenario classes follow, boundary handoff, failure-mode and reset propagation, and shared-resource contention, none writable at IP level, together with firmware-style stimulus and edge-following fault localization. A quantified case study, forty defects

caught at subsystem scope and a 3.6-fold per-ECO cost amplification avoided, makes the shift-left case concrete.

6.1 The Bridge Passed. What Can Still Fail?

Chapter [5](#) ended with a clean IP-level regression. The bridge passes all P0 and P1 scenarios. Every design bug found during regression has been fixed, verified, and added to the regression suite. The XTP reflects what was learned. By every IP-level measure, verification is complete.

And yet the final sentence of Chap. [5](#) named exactly what comes next: correctness at subsystem level depends not on any individual component but on the contracts between them.

Now the bridge is integrated into a subsystem. It becomes one node among several: a DMA controller drives transactions through an interconnect to the bridge, which fans out to a set of APB peripherals. An interrupt controller collects completion signals. A reset controller sequences power-on initialization across all components. Every component in this integration has passed its own IP-level regression.

And the subsystem will fail. Not because any component is wrong in isolation, but because the bugs that kill tapeouts do not live inside components. They live in the space between them, and that space is exactly what IP-level verification cannot see.

This chapter defines how VEGA extends from IP-level verification to subsystem verification by composing the artifacts built in earlier chapters, the XTPs, the manifests, the testbench agents, and the debug discipline, into a framework that covers the integration space that no IP-level test plan can see.

6.2 Where Subsystem Bugs Live

IP-level verification covers what a component does in isolation. It verifies that the component matches its specification when driven by a model stimulus that behaves correctly and a model environment that behaves correctly.

That model stimulus and that model environment do not exist in the subsystem. In the subsystem, the upstream component drives real transactions under real concurrency constraints. The downstream component has real latency and real error conditions. The reset controller sequences signals with real timing relationships. None of this was visible at IP level.

The result is a class of defects that cannot be found by IP-level verification, regardless of how thoroughly that verification was conducted (Fig. 6.1).

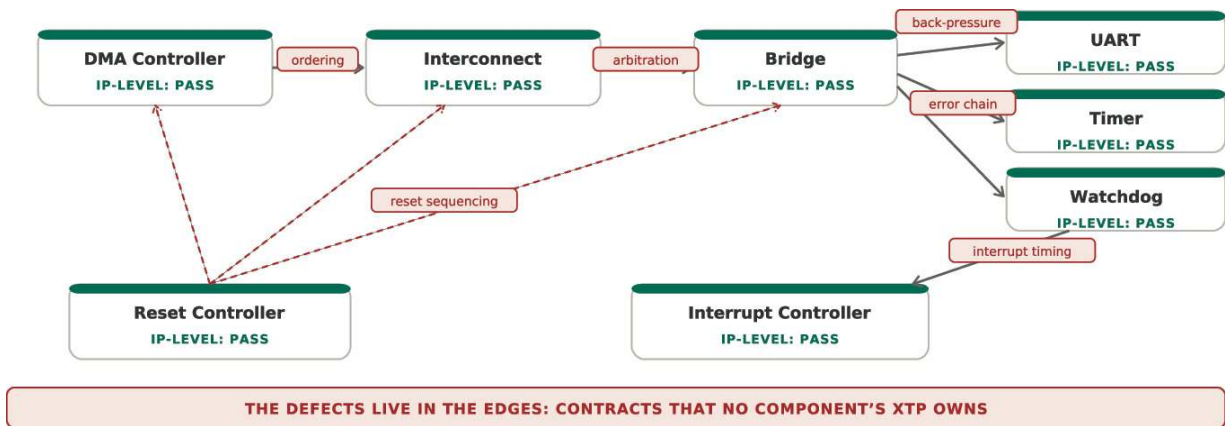


Fig. 6.1 The gap map. Each component passes IP-level verification independently. The subsystem defects live in the edges between components: the ordering contracts, error propagation chains, interrupt timing assumptions, and reset sequencing relationships that are not captured in any component's XTP

Four broad categories of defects live exclusively in these gaps. The following sections refine these into a more granular ten-category taxonomy that maps directly to assertion checks, but the broad categories establish the shape of the problem first.

Ordering assumption violations. Each component was verified with a model neighbor that behaved correctly. In the subsystem, real neighbors behave under real constraints. The DMA controller assumes the downstream slave will assert HREADY within a bounded number of wait states, completing the transfer within the DMA's programmed timeout. The bridge makes no such guarantee: slave response time is bounded only by the APB peripheral, which can exceed the DMA's assumption when the peripheral is under load or mid-state-machine.

Neither IP test plan covers this because neither team owned the interaction.

Error propagation chains. A peripheral returns an error response. The bridge propagates it upstream. The DMA controller must handle it and assert an interrupt. The interrupt controller must serialize it correctly. The end-to-end chain was never tested at IP level because no IP testbench modeled the full chain. Individual links were verified. The chain was not.

Reset and power sequencing. At IP level, reset is clean and isolated. At subsystem level, reset sequencing across multiple clock domains is a first-order correctness requirement. If a component deasserts reset while its downstream neighbor is still in reset, it can immediately drive transactions to a device that is not ready to receive them, hanging the entire bus. This class of defect is absent from every IP-level regression and routinely appears in first-pass subsystem integrations.

Shared resource contention. At IP level, each component operated with a dedicated path and no competition. At subsystem level, multiple components share interconnects, arbiters, and address spaces. Behaviors that are individually correct become incorrect when they occur simultaneously under real arbitration conditions.

6.3 The Ten Categories of Subsystem Intent

The four broad categories above describe *where* subsystem defects live. To verify against them systematically, a more granular taxonomy is required: one that is protocol-agnostic, applicable to any subsystem boundary, and direct enough to map to specific assertion checks.

The following ten categories cover the integration intent of any subsystem composed of digital components communicating across protocol, clock, reset, or power boundaries. Every meaningful contract between two components in a subsystem belongs to one or more of these categories. The taxonomy is the protocol-agnostic checklist that

the Verification Strategist applies to every edge of the Subsystem Intent Graph (introduced in the next two sections).

The taxonomy is exhaustive for digital subsystem boundaries. Every cross-component defect this book has discussed, and every integration bug pattern presented in the next section, maps to exactly one of these ten categories. Walking each SIG edge against the ten categories produces, by construction, the integration verification plan.

Categories 1-5 cover the crossing points, where different domains, protocols, and timing assumptions collide:

1. **Clock Domain Crossings (CDC).** How does data cross between clock domains? Subsumes synchronizer type and latency, multi-bit crossing strategy, metastability window, and reconvergence of synchronized signals. CDC defects produce intermittent failures that survive every same-domain regression.
2. **Reset Sequencing and Propagation.** Which reset deasserts first? What happens during the gap? Subsumes reset ordering across domains, minimum assertion width, FSM initialization delay after reset deassertion, and reset behavior during active traffic. The most common first-pass subsystem failure mode and the most expensive to debug post-silicon.
3. **Protocol Translation.** How does the source protocol map to the target? Subsumes pipeline-to-non-pipeline mapping (the AHB-to-APB core problem), wait-state insertion, response code translation, and attribute propagation across the boundary. Every protocol bridge in a subsystem has a translation contract; every contract has corner cases that IP-level verification does not see.
4. **Back-Pressure and Flow Control.** What happens when the downstream is slow or full? Subsumes credit and ready deassertion timing, maximum permitted wait states, internal buffer depth

timing, maximum permitted wait states, internal buffer depth

limits, and timeout behavior. Back-pressure failures often cascade across multiple boundaries before becoming observable.

5. **Burst and Packet Decomposition.** How are multi-beat transfers decomposed at the boundary? Subsumes per-beat address calculation, wrap and increment boundary handling, narrow transfer decomposition, and beat interleaving. Address arithmetic at decomposition boundaries is a frequent source of silent data corruption.

Categories 6-10 cover the hidden contracts: assumptions that live in architects' heads but rarely appear in any IP test plan:

6. **Error Propagation and Mapping.** How do errors cross subsystem boundaries? Subsumes error response timing and code mapping, error handling during bursts (abort versus continue), poison and tainted-data propagation, and interrupt generation from error responses. The error path is consistently the least-tested path at every level of verification.
7. **Ordering and Exclusivity.** Can responses overtake requests? What about exclusive accesses? Subsumes write buffer bypass rules, exclusive monitor scope across bridges, ordering guarantees across protocol translations, and in-order completion requirements. Read-after-write hazards through internal buffers are a common defect site that produces silent software inconsistency.
8. **Address Decode and Routing.** What happens on access to unmapped or reserved addresses? Subsumes default slave behavior, address hole handling, decode boundary crossing for spanning transfers, and device-select one-hot enforcement. Address holes are a particularly dangerous defect site: their failure mode is identical in symptom to a hung peripheral but originates from a different subsystem layer.

originates from a different subsystem layer.

9.

Data Integrity Across Domains. Does every byte arrive intact across the boundary? Subsumes byte-lane mapping on narrow transfers, endianness conversion, data corruption during CDC, and ECC propagation across protocol boundaries. Byte-level data integrity is rarely visible in protocol-level VIPs.

10.

Power State and Isolation. What if one domain powers down while the other is active? Subsumes isolation cell behavior, retention state, wake-up handshake sequencing, in-flight transaction handling at power transition, and clamp values. Power-aware verification at IP level and protocol-aware verification at subsystem level rarely intersect; defects live in that intersection.

The ten categories are protocol-agnostic by construction. They apply equally to AHB-to-APB, AXI-to-CHI, PCIe-to-CXL, or UCIe-die-to-die boundaries. The specific sub-aspects of each category vary by protocol; the category itself does not.

This is the vocabulary the Subsystem Intent Graph uses. Every SIG edge introduced in the next section carries a contract that belongs to one or more of these ten categories. The assumption harvest extracts implicit contracts; the categories give those contracts a shared structure.

6.4 Integration Bug Patterns That IP-Level Verification Misses

The ten categories define the structural taxonomy. The following ten bug patterns are the failure modes that have been observed across production subsystem verification programs, with one or more representative defects for each category. Each pattern is real in the sense that it has occurred in real designs, has been silently missed by

IP-level VIPs, and has been caught only when the subsystem verification process explicitly walked the relevant intent category.

The pattern catalogue serves two purposes. First, it gives the Verification Strategist a concrete checklist to apply when reviewing subsystem scenarios for completeness. Second, it makes the abstract ten-category taxonomy actionable: each pattern names what failure looks like, where it originates, and why IP-level VIPs are structurally unable to detect it.

#1 Data Corruption on Back-to-Back Transfers Across CDC

Severity: Critical Category: CDC

A synchronizer misses the second of two transfers when both arrive within the synchronization window. The first transfer completes correctly; the second is silently dropped. Downstream computations use the missing data and produce wrong results without any error signal.

Why IP-level VIPs miss it: Each side of the CDC has its own protocol VIP. Neither has visibility into the synchronizer timing or the back-to-back arrival pattern at the boundary.

#2 Transaction Accepted Before Downstream FSM Initializes

Severity: Critical Category: Reset Sequencing

The upstream component's reset deasserts on schedule. The downstream component's reset also deasserts on schedule, but the downstream FSM requires additional cycles to initialize before it can correctly process incoming transactions. An early transaction arrives during this initialization window and enters an undefined state.

Why IP-level VIPs miss it: The IP-level VIP models reset per the protocol specification, not per the implementation's actual FSM initialization latency. The defect is in the gap between specified and implemented timing.

#3 Bus Lockup on Cross-Domain Wait-State Timeout

Severity: High Category: Protocol Translation

A slow downstream component holds its ready signal low indefinitely. The bridge propagates this back-pressure upstream, which

locks the entire upstream bus fabric. No timeout mechanism exists at the boundary to detect or recover from this condition.

Why IP-level VIPs miss it: Each VIP sees a valid protocol sequence on its side. Neither checks for a cross-domain timeout because neither owns the cross-domain timing contract.

#4 Buffer Overflow Drops Transactions Silently *Severity: High*
Category: Back-Pressure / Flow Control

An internal buffer fills during a burst with a slow downstream. New entries overwrite pending entries before the pending ones complete. No back-pressure signal is generated upstream. Transactions are lost without error indication.

Why IP-level VIPs miss it: Buffer depth is an implementation detail not visible at the protocol level. The VIPs validate protocol correctness, not implementation queue capacity.

#5 Address Calculation Error on Wrapping Burst
Decomposition *Severity: High* *Category: Burst Decomposition*

A wrapping burst from an unaligned start address decomposes with the wrong wrap-boundary mask. The downstream component receives transactions at incorrect addresses. Data is written to or read from the wrong locations.

Why IP-level VIPs miss it: The protocol VIP checks burst legality at its own interface. It does not re-derive the addresses produced after the bridge has decomposed the burst.

#6 Error During Multi-Beat Transfer Leaves Bus Undefined
Severity: High *Category: Error Propagation*

The downstream signals an error on beat N of a decomposed burst. The bridge correctly signals the error upstream but continues issuing the remaining beats. The state of the bus during those remaining beats is undefined; data on the bus is stale or invalid.

Why IP-level VIPs miss it: Each VIP handles error per its own protocol's rules. Neither validates the cross-boundary contract for what happens to the unfinished beats of an in-flight burst when an error occurs mid-burst.

#7 Read Returns Stale Data from Internal Write Buffer *Severity: Medium Category: Ordering and Exclusivity*

A pending write sits in the bridge's internal write buffer. An immediate read to the same address bypasses the write buffer and returns the old downstream value. Software observes inconsistency between the write it issued and the read it just performed.

Why IP-level VIPs miss it: Both VIPs see protocol-correct behavior on their respective sides. The ordering contract that governs read-after-write through internal buffers is an implementation contract, not a protocol contract.

#8 Access to Address Hole Hangs the Subsystem *Severity: Medium Category: Address Decode*

A master accesses an address that is not mapped to any device in the subsystem. No downstream device is selected. The bridge waits forever for a response that will never arrive. The upstream bus locks up.

Why IP-level VIPs miss it: The upstream VIP does not know the subsystem-level address map. The downstream VIP never sees the transaction because no slave is selected. Neither side has the context needed to detect the failure.

#9 Byte-Lane Corruption on Narrow Transfer Across CDC *Severity: Critical Category: Data Integrity*

A narrow write transaction crosses a CDC boundary. The synchronizer captures the full data bus including stale lanes from a previous unrelated transaction. Downstream observes corrupted byte-select information.

Why IP-level VIPs miss it: The protocol VIP validates byte strobe correctness at its interface. CDC analysis tools check the synchronizer's structural correctness. Neither validates the byte-level correctness of the data *after* crossing the boundary.

#10 In-Flight Transaction Lost During Power-Domain Transition *Severity: High Category: Power State and Isolation*

The power manager isolates the downstream domain while the bridge has an active transaction in flight. The isolation cells clamp the

ready signal high, causing the bridge to observe a fake completion. The transaction is reported as successful upstream but the data never reaches the downstream destination.

Why IP-level VIPs miss it: The power-aware VIP verifies isolation cell behavior at its own boundary. The protocol VIP has no power domain awareness. The interaction between protocol activity and power transitions belongs to neither.

These ten patterns are not exhaustive; additional patterns exist in protocol-specific subsystems and emerging interconnect domains such as chiplets and coherent fabrics. They are, however, sufficient as a starting checklist for any subsystem verification program. A SIG that covers all ten patterns plus its protocol-specific extensions has addressed the integration defect classes that most commonly survive IP-level verification.

6.5 The Subsystem Intent Graph

The **Subsystem Intent Graph (SIG)** is the artifact VEGA uses to make integration-space defects visible before integration begins.

The SIG is not a block diagram. A block diagram shows components and connections. The SIG shows **contracts between components**: what each component guarantees to its neighbors, and what each component assumes from its neighbors. The gap between a guarantee and an assumption is not an architectural risk. It is a defect that exists before integration starts.

6.5.1 Building the SIG: A Four-Step Process

The SIG is constructed using a repeatable four-step process. Each step consumes artifacts that the verification team and design team already produce. The process introduces no new tools and no new specification formats; it organizes existing artifacts to expose the contracts between them.

1. **Identify Nodes.** List every component that participates in the subsystem: IP blocks, interfaces, clock domains, reset domains, and

infrastructure components (reset controller, interrupt controller, power management unit). Infrastructure components are first-class nodes even when they are not data-path participants.

2. **Map Edges.** Identify every dependency between nodes: data flows, control signals, domain crossings, reset relationships, interrupt connections. Each edge represents a potential defect site. An edge between two components in the same clock domain is a different verification problem than an edge that crosses a CDC boundary; the edge type informs the categories that apply.
3. **Capture Intent.** Walk each edge against the ten intent categories from Sect. [6.3](#). For each applicable category, add the contract that the edge must satisfy: the upstream guarantee, the downstream assumption, and the risk level if they do not align. This is the assumption harvest step; the prompt that automates the harvest is shown below.
4. **Generate Assertions.** Translate each captured intent into a concrete SVA property or coverage point. Section [6.6](#) provides the category-to-assertion mapping that drives this translation. The output is the executable verification plan that the SIG produces by construction.

6.5.2 SIG Construction Inputs

Three categories of input feed the four-step process. None of them is unique to VEGA; every mature design organization produces them already.

Architecture specifications. Timing requirements between components, reset sequence diagrams, protocol bridge behavior descriptions, and integration notes. The specification level is the authoritative source for guarantees and assumptions; the assumption harvest extracts these into a structured form.

RTL and IP-XACT. Module hierarchy, clock and reset domain definitions, interface connections, and parameterization. RTL provides

the actual structure; IP-XACT (where available) provides the structural metadata in a form that automates much of step 1 (node identification) and step 2 (edge mapping).

CDC and RDC reports. Domain crossings, synchronizer types, known violations, and waiver rationale from existing CDC and RDC analysis flows. These reports already enumerate the boundaries that require Category 1 (CDC) and Category 2 (Reset) coverage in the SIG. SIG construction reuses this analysis rather than duplicating it.

The four-step process produces the SIG by walking each edge against the categories that apply. No new tooling is required. The Verification Strategist's role is judgment: which assumptions are genuinely misaligned in the target integration, which risk levels are accurate, and which SIG edges therefore demand mandatory subsystem scenarios.

6.5.3 Building the SIG: The Assumption Harvest

The primary input to the third step (Capture Intent) is the set of implicit assumptions each component's verification team made during IP-level verification. These assumptions are embedded in the IP-level testbench: in how the model stimulus was constrained, in how the model environment was configured, in what error conditions were and were not simulated.

The assumption harvest extracts these implicit assumptions explicitly and converts them into SIG edges. Running the harvest is a Verification Strategist activity: the output requires review against the actual integration context before any assumption is promoted to a HIGH-risk edge. AI generates the candidate list. The Verification Strategist validates which assumptions are genuinely misaligned in the target subsystem.

Prompt: Assumption Harvest, Component Integration Assumptions

You are assisting a verification strategist building a Subsystem Intent Graph.

Task: Extract the implicit integration assumptions embedded in the IP-level verification of a single component.

Rules:

- Focus on assumptions about upstream components (what must be true before this component receives a transaction).
- Focus on assumptions about downstream components (what must be true for this component's outputs to be processed correctly).
- Focus on assumptions about reset, clock, and power sequencing.
- Do NOT extract verified behaviors -- extract assumed behaviors that were modeled but not owned by this component's XTP.
- Flag each assumption with a risk level (High/Medium/Low) based on how likely the real integration environment is to violate it.

For each assumption, output:

- (1) Assumption_ID
- (2) Component_Name
- (3) Neighbor_Component (upstream or downstream)
- (4) Assumption_Text (what was assumed to be true)
- (5) How_Modeled (how the IP testbench enforced this assumption)
- (6) Violation_Risk (High/Medium/Low)
- (7) Violation_Consequence (what fails if the assumption is wrong)
- (8) Intent_Category (one or more of the 10 intent categories)

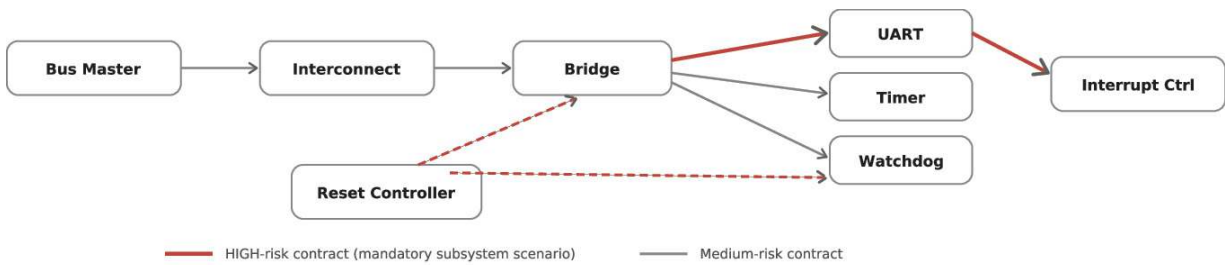
Inputs:

- Component spec excerpt: <<< INSERT >>>
- IP-level XTP: <<< INSERT >>>
- IP-level testbench manifest: <<< INSERT from Chapter 3 >>>

Run this prompt for each component in the subsystem. The output is a table of assumptions. Each assumption becomes a candidate SIG edge. Assumptions with High violation risk become mandatory subsystem scenarios.

6.5.4 SIG Structure: Nodes and Edges

The SIG has two types of nodes and two types of edges (Fig. [6.2](#)).



ANATOMY OF ONE EDGE



Every misaligned edge becomes a mandatory subsystem scenario before integration begins.

Fig. 6.2 The subsystem intent graph (SIG) for a representative AHB/APB subsystem. Each edge carries a contract: a guarantee from the upstream component and an assumption from the downstream. The gap between them defines the subsystem verification scope

Nodes represent either data-path components (bus masters, interconnects, bridges, peripherals) or infrastructure components (reset controllers, interrupt controllers, power management units). Infrastructure nodes are often omitted from block diagrams but are first-class nodes in the SIG because reset and interrupt sequencing are the most common sources of subsystem-level defects.

Edges represent contracts. Each edge has two halves: the *guarantee* from the upstream component (what it promises to provide) and the *assumption* from the downstream component (what it requires in order to function correctly). When a guarantee does not fully satisfy an assumption, the edge is a defect site. The SIG makes all defect sites visible before integration begins.

Table 6.1 shows the SIG edge table for the example subsystem. Every row where the guarantee and assumption do not align is a mandatory subsystem verification scenario.

Table 6.1 SIG edge contracts for the AHB/APB subsystem. Rows marked HIGH represent known defect sites that must be addressed by subsystem scenarios.

6.6 From Intent to Assertions

The ten intent categories define what to verify. The SIG enumerates where to verify it. The remaining link is how to verify it: the translation from each intent category into concrete SVA properties and coverage points.

Table 6.2 provides the canonical mapping. Each row gives one or more assertion patterns that exercise the intent of the corresponding category. The patterns are protocol-agnostic templates; the specific signals and timing values are filled in from the SIG edge contract, but the structure of what must be checked is fixed by the category.

Table 6.2 Mapping from intent category to assertion or coverage check. Each row is a protocol-agnostic template; the specific signal names and timing bounds are filled in from the SIG edge contract.

The intent-to-assertion mapping closes the gap between the abstract ten-category taxonomy and the concrete artifacts the simulator consumes. A SIG edge whose contract belongs to Category 2 (Reset) generates, by construction, the three assertion patterns in row 2 of the table, specialized to the specific reset signals, FSM initialization signals, and timing values from the edge contract.

This is what the fourth step of the SIG construction process (? Generate Assertions?) produces. The output is not a discretionary collection of checks based on engineer experience; it is a deterministic mapping from edge contract to verifiable property. Two engineers applying the same SIG to the same subsystem produce, in principle, the same assertion set. The reproducibility is the point: the verification plan becomes an artifact that can be reviewed and audited rather than a body of tribal knowledge.

Prompt: SIG Edge to SVA Generation

You are assisting a verification strategist.

Task: Generate concrete SVA assertions and coverage points for a SIG edge, using the intent-category-to-check mapping.

Rules:

- For each intent category that applies to the edge, generate the assertion patterns from the intent-category-to-check mapping table.
- Specialize each pattern with the specific signals, timing bounds, and protocol fields from the edge contract.
- Generate one coverage point per assertion to confirm the property was actually exercised, not merely held vacuously.
- Place generated assertions in a 'bind' file (per Chapter 3), not in design RTL.
- Do NOT fabricate timing bounds not present in the edge contract or component specs. Flag missing bounds as CONTRACT_GAP.

For each assertion, output:

- (1) Assertion_ID (trace to EDGE_ID and intent category)
- (2) Property name
- (3) SVA property body
- (4) Companion coverage point
- (5) Severity on failure (UVM_ERROR or UVM_FATAL)

Inputs:

- SIG edge contract: <<< INSERT >>>
- Component signal lists: <<< INSERT >>>
- Protocol specifications: <<< INSERT relevant timing sections >>>

6.7 Extending the Manifest to Subsystem Scope

The manifest introduced in Chap. 3 described a single IP: its interfaces, protocols, and the status of each VIP component. The `verification_intent.txt` alongside it captured the verification goals in plain language. At subsystem level, both artifacts extend naturally to cover the full integration context. The manifest gains component entries and SIG edges. The `verification_intent.txt` gains subsystem-level goals: which SIG contracts must be verified, which failure modes must be exercised, and which interaction

behaviors the subsystem must demonstrate before sign-off. Nothing in the process changes. The artifacts grow; the method does not.

This consistency is intentional and load-bearing. An engineer who has worked through Chaps. [2](#) and [3](#) picks up a subsystem manifest and recognizes it immediately: the same fields, the same structure, the same boundary between infrastructure and intent. The only new elements are the SIG edges and the subsystem-level goals that reference them. There is no new tooling to learn, no new process to onboard. The cognitive cost of moving from IP verification to subsystem verification is the cost of understanding the integration context, not the cost of learning a new framework.

That is the design principle behind the extension. Subsystem verification is harder than IP verification because the design space is larger and the failure modes cross boundaries that no single component owns. The framework should not make it harder still by introducing new artifacts, new vocabulary, or new governance structures at the point where complexity is already increasing. VEGA extends upward by adding to what already exists, not by replacing it. The Verification Strategist who reaches Chap. 6 already knows how to work with these artifacts. This chapter teaches them what to add.

The key additions to the manifest are two new fields: `components` (listing all RTL components that are present as existing blocks rather than generated models) and `sig_edges` (encoding the SIG contracts from Table [6.1](#) as structured manifest entries).

Subsystem manifest: extending Chapter 3 format

```
{
  "design": "ahb_apb_subsystem",
  "components": [
    { "name": "ahb2apb_bridge", "vip_status": "existing_rtl",
      "ip_xtp": "bridge_xtp_v2.yaml" },
    { "name": "dma_controller", "vip_status": "existing_rtl",
      "ip_xtp": "dma_xtp_v1.yaml" },
    { "name": "uart", "vip_status": "existing_rtl",
      "ip_xtp": "uart_xtp_v3.yaml" },
    { "name": "timer", "vip_status": "existing_rtl",
      "ip_xtp": "timer_xtp_v2.yaml" },
    { "name": "watchdog", "vip_status": "existing_rtl",
      "ip_xtp": "wdog_xtp_v1.yaml" },
    { "name": "reset_ctrl", "vip_status": "existing_rtl" }
  ],
  "interfaces": [
    { "name": "ahb_master_stim", "vip_status": "adapted",
      "note": "firmware-style register sequences; no direct AHB drive" },
    { "name": "apb_monitor", "vip_status": "existing" },
    { "name": "interrupt_monitor", "vip_status": "generated" },
    { "name": "reset_injection", "vip_status": "generated" }
  ],
  "sig_edges": [
    { "id": "EDGE_001",
      "from": "reset_ctrl", "to": "all_apb_peripherals",
      "contract": "preset_stable_before_hreset_deassert",
      "intent_category": "reset_sequencing",
      "note": "applies to: bridge, uart, timer, watchdog",
      "risk": "HIGH" },
    { "id": "EDGE_002",
      "from": "bridge", "to": "uart",
      "contract": "pready_bounded_response",
      "intent_category": "back_pressure",
      "risk": "HIGH" },
    { "id": "EDGE_003",
      "from": "bridge", "to": "uart",
      "contract": "pslverr_propagated_as_hresp_error",
      "intent_category": "error_propagation",
      "risk": "HIGH" },
    { "id": "EDGE_004",
      "from": "uart", "to": "intc",
      "contract": "interrupt_pulse_minimum_width",
      "intent_category": "ordering",
      "risk": "HIGH" },
    { "id": "EDGE_005",
      "from": "bridge", "to": "all_peripherals",
      "contract": "unmapped_address_returns_hresp_error",
      "intent_category": "address_decode",
      "risk": "HIGH" }
  ]
}
```

Four structural principles govern the subsystem manifest and distinguish it from its IP-level counterpart.

Existing IP agents are unchanged. The AHB master and APB slave agents built in Chap. 3 for the bridge are reused without modification. New agents are added only for new interfaces: interrupt monitoring, reset injection, and the firmware-style stimulus model. The infrastructure boundary from Chap. 3 holds at subsystem level.

VIP status for RTL components. Instead of `generated` or `adapted`, RTL components carry the status `existing_rtl`. This signals to the testbench generator that these components are instantiated as RTL, not as BFM models. Every component that appears in the SIG diagram must appear in the manifest.

IP-level XTP references. Each component's XTP is included in the manifest. The subsystem test plan references these to establish what is already verified and to identify what falls outside every component's existing scope. Components without an available XTP are flagged for IP-level verification before subsystem integration begins.

SIG edges as first-class manifest fields. The `sig_edges` array encodes the contracts from the SIG directly in the manifest. Each edge carries the risk level from the assumption harvest and the intent category from Sect. 6.3 that governs it. EDGE_001 covers reset sequencing across all APB peripherals. EDGE_005, added after reviewing the full peripheral address map, covers unmapped address handling, a gap that only became visible when all three peripherals were specified together. High-risk edges drive mandatory scenario generation. The manifest is the single artifact that connects the SIG analysis to the testbench generation to the test plan.

6.8 Three Scenario Classes That Do Not Exist at IP Level

The subsystem XTP extends each component's IP-level XTP with scenarios that cover the SIG edges. These scenarios fall into three

classes. None of them can be written at IP level because none of them is visible from within a single component.

Each scenario in the subsystem XTP carries two fields that IP-level rows do not: `SIG_Edge_ID` (which manifest edge the scenario exercises) and `Neighbor_Components` (which components participate beyond the DUT). For the example subsystem: `EDGE_001` drives the reset injection scenarios, `EDGE_002` drives the hung peripheral scenario, `EDGE_003` drives the `PSLVERR` propagation chain, and `EDGE_004` drives the interrupt pulse scenario. Every HIGH-risk edge in the manifest must map to at least one scenario in the subsystem XTP before the test plan is considered complete.

6.8.1 Boundary Handoff Scenarios

Boundary handoff scenarios exercise what happens at the moment one component transfers ownership or control to the next. The IP-level testbench always modeled a clean handoff. The real subsystem has real timing.

For the AHB/APB subsystem, the highest-risk boundary handoff is `HTRANS` sequencing across an arbitration boundary. The IP-level bridge testbench never changed AHB mastership mid-sequence. In the real subsystem, the interconnect can transfer `HGRANT` to a different master while an APB access is still in flight through the bridge. The bridge must handle this correctly: it must complete the in-flight APB access, then assert `HREADY` when that access completes. Only at that point does ownership transfer to the incoming master. ?Completing on the next cycle? is not a guarantee the bridge can make; the APB access may require multiple `PCLK` cycles to resolve.

Prompt: Boundary Handoff Scenario Generation

You are assisting a verification strategist.

Task: Generate boundary handoff scenarios for a SIG edge.

Rules:

- Generate one nominal scenario and at least two boundary conditions (early handoff, late handoff, handoff under load).
- All scenarios must exercise the contract on the specified edge.
- Do NOT modify infrastructure files.
- Each scenario must have an observable oracle traceable to a SIG edge contract.
- If a scenario requires a new monitor or checker not present in the subsystem manifest, flag it as INFRA_GAP.

For each scenario, output:

- (1) Scenario_ID (trace to EDGE_ID)
- (2) Scenario_Class: boundary_handoff
- (3) Precondition (system state before stimulus)
- (4) Stimulus (sequence of transactions or register operations)
- (5) Observable oracle (what must be true at completion)
- (6) SIG_Edge_Contract exercised
- (7) Priority (P0/P1/P2)

SIG edge: <<< INSERT EDGE_ID and contract text >>>

Component specs: <<< INSERT relevant sections >>>

IP XTPs involved: <<< INSERT >>>

6.8.2 Failure Mode and Error Propagation Scenarios

Failure mode scenarios test what happens when something goes wrong in one component and that failure must propagate correctly through the rest of the subsystem. These scenarios are absent from IP-level verification because IP-level testbenches model well-behaved neighbors. The real integration has neighbors that fail.

Three failure patterns are universally present in AHB/APB and similar bus-based subsystems.

Hung peripheral. A peripheral's PREADY never asserts. This models a peripheral that is deadlocked internally, or has been reset while an access is in flight, or is waiting for a condition that will never occur. The question is: does the bridge propagate this as a bus hang (HREADY held low indefinitely, freezing the entire AHB bus) or does it

time out and return HRESP=ERROR? The correct behavior is specified in the bridge spec; the scenario verifies it against the real peripheral.

PSLVERR propagation chain. A peripheral returns PSLVERR. The bridge must propagate HRESP=ERROR. The DMA controller must handle HRESP=ERROR by aborting its transfer and asserting an interrupt. The interrupt controller must correctly serialize the interrupt. The scenario exercises the complete chain from peripheral error to interrupt generation. At IP level, each link was verified in isolation. The chain was not.

Reset and power scenarios, integrated. This scenario class is included here rather than in a separate section because it is most correctly understood as a failure propagation scenario: what happens when reset propagates asynchronously through the system.

The highest-risk reset scenario in any AHB/APB subsystem is: HRESETn deasserts before PRESETn. This is EDGE_001 in the SIG, the HIGH-risk reset sequencing edge. In this condition, the bridge comes out of reset and is immediately available to forward AHB transactions, while every APB peripheral is still in reset and cannot respond. The next AHB transaction from the DMA controller will produce a PREADY that never asserts.

The scenario: configure the reset controller to release HRESETn two cycles before PRESETn. Initiate a DMA transfer that will arrive at the bridge within those two cycles. Observe whether the bridge correctly handles the race.

A second high-value reset scenario: assert HRESETn asynchronously while an APB access is in flight through the bridge. The bridge must cleanly terminate the APB access (or correctly abandon it) without leaving the APB bus in an undefined state for the next transaction after reset deasserts.

Prompt: Failure Mode and Reset Scenario Generation

You are assisting a verification strategist.

Task: Generate failure mode and reset injection scenarios for HIGH-risk SIG edges.

Rules:

- Generate one hung-resource scenario, one error-propagation chain scenario, and one reset injection scenario for each HIGH-risk edge in the SIG.
- Reset scenarios must specify: which reset signal, at which point in the transaction sequence it is injected, and the expected system state after reset deasserts.
- All scenarios must be observable: specify the monitor or checker that will detect a failure.
- Do NOT assume any behavior not in the SIG edge contracts or component specs.

For each scenario, output:

- (1) Scenario_ID (trace to EDGE_ID)
- (2) Failure_Class: hung_resource | error_chain | reset_injection
- (3) Stimulus (what is driven and when)
- (4) Injection_Point (for reset: which cycle relative to the in-flight transaction)
- (5) Expected behavior per SIG contract
- (6) Observable oracle
- (7) Consequence if contract is violated

HIGH-risk SIG edges: <<< INSERT from manifest sig_edges >>>

Component specs: <<< INSERT >>>

Reset controller spec:<<< INSERT reset sequencing section >>>

6.8.3 Shared Resource Contention Scenarios

At IP level, every component had exclusive access to its interfaces. At subsystem level, multiple components share the interconnect, the address space, and the arbitration logic. Behaviors that are individually correct become incorrect under real concurrency.

The key shared resource scenarios for an AHB/APB subsystem:

Concurrent access to different peripherals. The DMA controller accesses the UART while a CPU simultaneously accesses the Timer. Both are valid APB targets. The bridge must serialize the two AHB transactions correctly and not corrupt either peripheral's access.

Concurrent access to the same peripheral. Two bus masters attempt to access the UART simultaneously. Arbitration must ensure that both transactions complete correctly and in order, with no data corruption.

DMA transfer spanning a peripheral boundary. A DMA transfer is programmed with a size that would span across two APB peripheral address regions. The bridge must correctly split, map, or reject this transfer per the address decoding specification.

Address decode completeness. This scenario class is distinct from contention: it tests what happens when an AHB address falls exactly at the boundary between two peripheral regions, or maps to a hole in the address space (no peripheral responds). These conditions only become visible when the full peripheral address map is assembled. At IP level, the bridge was tested with a model that responded to all addresses. In the real subsystem, a hole address produces no PREADY response: a liveness failure identical in symptom to the hung peripheral scenario but with a different root cause. The bridge must return HRESP=ERROR for unmapped addresses, not hang the bus.

Prompt: Shared Resource Contention Scenario Generation

You are assisting a verification strategist.

Task: Generate shared resource contention scenarios for the subsystem.

Rules:

- Identify all shared resources in the SIG (interconnects, address regions, arbiters).
- Generate at minimum: one concurrent-different-target scenario, one concurrent-same-target scenario, and one address-boundary scenario.
- Each scenario must specify concurrency timing: how many cycles apart are the two initiating transactions.
- Do NOT generate scenarios that duplicate single-master scenarios already in any component's IP XTP.

For each scenario, output:

- (1) Scenario_ID
- (2) Shared_Resource
- (3) Initiating components (which masters, in what order)
- (4) Concurrency timing
- (5) Expected arbitration outcome (per interconnect spec)
- (6) Observable oracle (scoreboard or monitor check)
- (7) Priority

SIG: <<< INSERT edge table >>>

Interconnect spec:<<< INSERT arbitration rules >>>

IP XTPs: <<< INSERT to identify what NOT to duplicate >>>

6.9 How the Testbench Model Changes at Subsystem Level

The structural change from IP-level to subsystem-level verification is not in the infrastructure. That is unchanged and extended via the manifest. The change is in how stimulus is generated.

At IP level, stimulus was direct: a UVM sequence drove AHB transactions directly onto the DUT interface. The testbench owned the master role.

At subsystem level, the real bus master is an RTL component: a DMA controller, a CPU complex, a hardware accelerator. The testbench does

not drive AHB transactions directly. It drives the bus master by programming its configuration registers.

This is **firmware-style stimulus**: a sequence of register writes and reads over AHB that configure the DMA controller's source address, destination address, transfer size, and enable bit. The testbench still uses an AHB master agent, but it drives transactions to the DMA's register interface, not directly to the bridge. Once programmed, the DMA controller initiates the downstream AHB transactions autonomously. The testbench then observes the outcome: the transactions visible on the APB monitor, the interrupt assertion, the DMA status register after completion.

The difference matters for three reasons.

Observability shifts. At IP level, the testbench knew exactly what transactions it drove. At subsystem level, the downstream AHB transactions are generated by the RTL DMA controller. The testbench observes them through monitors but does not generate them. The oracle must check what the DMA produced against what the firmware sequence programmed it to produce.

Non-determinism increases. The DMA controller introduces latency, retry behavior, and priority arbitration that are not present in a direct-drive model. Stimulus timing is no longer cycle-accurate from the testbench's perspective.

Scenario abstraction level rises. Subsystem scenarios are expressed as: "program the DMA to transfer N bytes from address X to address Y" rather than "send HTRANS=NONSEQ, HADDR=X, HSIZE=2 on cycle 10." The scenario describes architectural behavior, not protocol transactions.

One practical consequence: verify the DMA register programming sequence in isolation before using it as subsystem stimulus. If the DMA is incorrectly programmed and the subsystem transfer fails, the failure could be attributed to the bridge or the peripheral rather than the DMA setup sequence. A brief directed test that programs the DMA and reads back its configuration registers, confirming source address, destination

address, and size are correctly latched, eliminates this ambiguity before subsystem scenarios run.

Prompt: Firmware-Style Stimulus Sequence Generation

You are assisting a verification strategist.

Task: Generate a firmware-style UVM stimulus sequence for a subsystem scenario. The sequence programs a bus master RTL component via register writes rather than driving transactions directly.

Rules:

- Express all stimulus as register-level operations (address, data, operation type: read/write).
- Include polling loops where the real firmware would poll for completion.
- Include timeout detection: if an expected status bit does not assert within N cycles, the sequence must raise a UVM_ERROR.
- Do NOT drive AHB transactions directly.
- The sequence class must NOT modify any infrastructure file.

Output:

- UVM sequence class skeleton with:
 - Register write steps (address, data)
 - Completion polling loop with timeout
 - Pass/fail check against XTP oracle
 - UVM_INFO logging at each step

Scenario description: <<< INSERT from subsystem XTP row >>>

DMA register map: <<< INSERT relevant registers >>>

Expected outcome: <<< INSERT XTP oracle >>>

Timeout cycles: <<< INSERT from SIG edge contract >>>

6.10 Debug at Subsystem Level: Edge-Following Fault Localization

Chapter [5](#) established that the first debug question at IP level is: *what category is this failure?*

At subsystem level, the first question is different: *which component owns this failure?*

A transaction mismatch at subsystem level could originate in the DMA controller, the interconnect, the bridge, the peripheral, or the

interaction between any two of them. A timeout could come from a hung peripheral, a stuck bridge, a reset sequencing error, or a DMA programming mistake. Opening a waveform before answering this question produces hours of undirected searching.

The SIG provides a systematic answer. Every observable failure has a location in the SIG. The debug process is: observe the symptom, identify which SIG edge it belongs to, check the monitor at both ends of that edge, and compare observed behavior against the edge contract. Before any of this begins, express the failure as an expected/observed/deviation triple (Sect. [5.4](#)). The triple identifies which edge the deviation belongs to. The SIG tells you which monitors to read.

Edge-following fault localization:

1. Identify the observable symptom from the UVM log. Express it as an expected/observed/deviation triple (Sect. [5.4](#)).
2. Identify the SIG edge that the deviation belongs to. Which component boundary was crossed incorrectly?
3. Check the monitor at the upstream end of that edge. Is the upstream component producing the correct output per its IP-level XTP?
 - If yes: the fault is in the downstream component or the edge contract itself. Advance one step downstream.
 - If no: the fault is in the upstream component. The component is violating its own IP-level XTP under subsystem conditions. This is a regression of IP-level behavior, likely triggered by a subsystem stimulus that was not covered at IP level.
4. Repeat until the component boundary where behavior first deviates from the IP-level XTP is identified. That component owns the failure.

For the AHB/APB example: the APB monitor shows PADDR=0x4000_0020 but the DMA was programmed for address 0x4000_0000. Check the AHB monitor at the bridge input. If HADDR=0x4000_0020 at the bridge input, the bridge is correctly forwarding a wrong address, so the fault is upstream of the bridge, in the DMA or interconnect. Check the AHB monitor between DMA and interconnect. If HADDR is correct there, the interconnect has a routing bug. If HADDR is wrong there, the DMA has an address calculation defect.

This process is faster than any waveform-first approach because the SIG tells you exactly which interface pairs to compare before any waveform is opened.

Prompt: Subsystem Failure Localization via SIG Edge Traversal

You are assisting a verification strategist.

Task: Localize a subsystem failure to a specific component using SIG edge traversal.

Rules:

- Work from the observable symptom inward along SIG edges.
- At each step, state whether the upstream monitor output is consistent with the upstream component's IP-level XTP.
- Do NOT guess root cause until the component boundary is identified.
- If a required monitor does not exist, output MONITOR_GAP and stop.

Output:

- (1) Starting edge (symptom location in SIG)
- (2) Traversal steps (edge ID, upstream check, result)
- (3) Component boundary where deviation first appears
- (4) IP-level XTP row that the observed behavior violates
- (5) Recommended minimal reproduction (from Chapter 5)
- (6) Classification: design_bug | integration_assumption | xtp_gap | infra_gap

Observed symptom (deviation triple): <<< INSERT >>>

SIG edge table: <<< INSERT >>>

Monitor outputs available: <<< INSERT >>>

IP-level XTPs: <<< INSERT >>>

The localization prompt produces a classification for every failure. Four classifications are possible: **design_bug** (RTL defect in the identified component), **integration_assumption** (the SIG contract was violated because the assumption harvest missed a misalignment), **xtp_gap** (the behavior is correct but was not planned for in any XTP), or **infra_gap** (the testbench cannot observe the failure correctly).

The most important distinction: subsystem failures that classify as **integration_assumption** are not design bugs in either component. They are gaps in the SIG, places where the assumption harvest did not identify that the guarantee and assumption were misaligned. The resolution requires a governance decision by the Verification Strategist with input from both component teams. Three resolutions are possible: the upstream component strengthens its guarantee (requires an RTL change to that component), the downstream component relaxes its assumption (requires an RTL change to that component), or the integration architecture changes (requires a specification update and potentially both). One of these three must be selected, documented as an architectural decision record, and reflected in an updated SIG edge. The edge is not closed until the chosen resolution is verified by a passing scenario.

6.11 The Subsystem Quality Gate

The IP-level quality gate from Chap. 5 asked: does every XTP row pass? That question is necessary at subsystem level but not sufficient. A subsystem can achieve 100% XTP pass rate and still have unexercised SIG edges that represent undetected integration defects.

The subsystem quality gate adds one structural question to the IP-level gate: **has every HIGH-risk SIG edge been exercised in both the nominal case and at least one failure mode?** The Verification Strategist signs off on this gate, not the regression script and not the coverage tool, for the same reason as at IP level: the boundary between a justified SIG edge waiver and an unresolved integration gap requires human judgment against the architectural context.

A complete subsystem quality gate:

1. All P0 subsystem scenarios pass.
2. All P1 subsystem scenarios pass or have documented waivers with JIRA and root cause.
3. All HIGH-risk SIG edges are covered by at least one passing scenario in the boundary handoff and failure mode classes. Shared resource scenarios are additionally required for any HIGH-risk edge involving a shared arbiter, shared address space, or shared interrupt line; they are not required for point-to-point or reset-only edges.
4. Every **integration_assumption** classification from Sect. [6.10](#) has been resolved: either the SIG contract was updated and re-verified, or the architectural gap was reviewed and explicitly accepted with a documented architectural decision record.
5. No subsystem scenario causes a regression in any component's IP-level XTP. A component that passes its IP-level regression but fails under subsystem stimulus has a latent defect that was not covered at IP level.
6. The subsystem XTP reflects what was learned. New edges and scenarios added during regression are reviewed and approved.
7. Debug artifacts are retained: the SIG edge table, the assumption harvest outputs, and the failure localization traversals. These become the primary inputs for SoC-level integration and for verifying the next revision of any component.

Check 5 deserves emphasis. A component that passes its IP-level regression but fails under subsystem conditions is the most dangerous class of subsystem defect. It means the IP-level XTP had a gap: the subsystem stimulus exposed a behavior that IP-level scenarios did not

cover. The resolution is to add the missing scenario to the IP-level XTP and re-run IP-level regression before proceeding. This is a feedback from subsystem verification back to component verification, the same XTP learning loop from Chap. 5, now operating across two levels of the hierarchy.

6.12 Quantified Impact at Subsystem Scope

The argument for subsystem-level verification is ultimately quantitative: the same defects discovered at the subsystem boundary cost dramatically less to resolve than the same defects discovered at full SoC integration or in silicon. This section presents the measurement that supports the argument.

The data below summarizes a representative subsystem verification program structured around the SIG, the ten-category taxonomy, and the scenario classes defined earlier in this chapter. Forty distinct defects were identified at subsystem scope across eight bug classes. Each was projected against the SoC integration phase at which the same defect would otherwise have been discovered, and the schedule impact of late-phase discovery was estimated.

Table 6.3 Subsystem-found defects by bug class, projected SoC discovery phase, and estimated schedule impact if discovery had been delayed to that phase.

The headline number is the defect count, but the more important observation is in the right two columns. Forty defects discovered at subsystem scope produce localized, bounded fixes: a defect in the DMA? s wrapping-burst decomposition is fixed in the DMA, re-verified against the subsystem regression, and closed within a sprint. The same defect discovered during SoC regression triggers a multi-week debug cycle: the failure manifests downstream of the DMA, requires cross-team coordination to localize, and re-validation requires the SoC environment which has its own setup and runtime cost.

The phase of discovery is the variable that compounds schedule risk, not the absolute defect count.

6.12.1 Per-ECO Validation Cost Amplification

A second measurement captures the per-ECO cost of validation at each verification scope. An ECO, an engineering change order, represents a fix to an existing defect that requires re-verification. The validation cost of an ECO is dominated by the time required to re-run regression and re-establish confidence in the affected behavior. Validation costs at the subsystem and full-SoC levels are not the same (Table [6.4](#)).

Table 6.4 Per-ECO validation effort: subsystem-scope fixes versus full-SoC fixes. Subsystem validation runs faster because the regression scope is bounded, the testbench setup is lighter, and the affected behavior is more directly observable.

A subsystem ECO costs approximately 2.5 days from fix to signoff. The same ECO at full-SoC scope costs approximately 9 days. The amplification factor of approximately $3.6\times$ per ECO is the cost of late-stage discovery, paid every time a fix arrives after the subsystem-to-SoC handoff.

The compounded effect across an entire program. A program that finds 40 defects at subsystem scope rather than at SoC scope saves approximately 6.5 days per ECO at the validation phase, plus the multi-week schedule impact per defect from Table [6.3](#). Across a representative SoC program, this compounds to a projected 48-week schedule recovery. The recovery comes not because subsystem verification finds bugs IP-level verification cannot find, but because the bugs it finds are resolved at a phase where validation is fast, scope is bounded, and ownership attribution is unambiguous.

6.12.2 Late-Stage Bugs Are Bounded, Not Eliminated

Subsystem verification does not eliminate late-stage bugs. Some classes of defects, such as system software interactions, multi-subsystem emergent behavior, and performance under realistic workloads, only

become visible at SoC scope and beyond. The argument for subsystem verification is not that it makes SoC verification unnecessary. It is that subsystem verification *bounds* the per-ECO cost of late-stage fixes, by ensuring that the integration-space defects that *can* be found earlier *are* found earlier. The SoC phase then handles the residual class of defects on a manageable schedule rather than absorbing the full forty-defect class on top of its own workload.

This is the practical implementation of shift-left at the subsystem boundary. The defects are not new. The phase at which they are discovered is.

6.13 Alignment with Industry Standards and Specifications

The SIG, the ten-category taxonomy, and the scenario classes in this chapter are not isolated framework constructs. They align with, and in several cases extend, active industry standards and emerging specifications in the verification and integration domain.

This alignment matters because it situates the subsystem verification methodology within the existing tooling and standards ecosystem rather than asking the Verification Strategist to adopt a parallel infrastructure. Each of the following standards already produces artifacts that the SIG can consume directly, or that can be augmented with SIG metadata at low integration cost.

Accellera CDC/RDC Standard. The Accellera Clock Domain Crossing and Reset Domain Crossing standard (2024?2025) defines a common format for reporting and exchanging CDC and RDC analysis results across tools. Category 1 (CDC) and Category 2 (Reset) of the intent taxonomy in Sect. [6.3](#) align directly with this standard's artifact set. CDC and RDC reports become first-class inputs to step 3 of SIG construction (Capture Intent), with intent timing predicates encoding the standard's CDC constraints as verification requirements that the assertion mapping in Sect. [6.6](#) then materializes. The SIG does not

replace CDC/RDC analysis; it extends it from a structural-checking tool into an integration-verification artifact.

UCIe (Universal Chiplet Interconnect Express). UCIe (2024 and subsequent revisions) defines the protocol stack for die-to-die communication in chiplet-based designs. Multi-die integration is fundamentally a subsystem verification problem at the die boundary: two independently designed and verified chiplets, each with its own internal contracts, integrated across a UCIe link with its own protocol, electrical, and link-management contracts. The SIG scales naturally to this scope. UCIe link nodes become first-class SIG nodes; UCIe-specific intent (link training, power management, sideband control) extends the ten-category taxonomy with chiplet-specific sub-categories. The bug patterns in Sect. [6.4](#) generalize to UCIe-specific failure modes such as link training corner cases and cross-die power state transitions.

IP-XACT (IEEE 1685-2022). IP-XACT defines structural metadata for IP component descriptions: interfaces, ports, parameters, configurations, and component connectivity. SIG construction reuses IP-XACT metadata for steps 1 and 2 (Identify Nodes, Map Edges). The SIG extends IP-XACT in one specific dimension: it adds *integration-time constraints* that are not captured in the component-level descriptions IP-XACT was designed for. Where IP-XACT describes what a component looks like, the SIG describes what contracts it must satisfy in a specific integration. The two artifacts are complementary; the SIG is a natural extension of an IP-XACT-described subsystem with the integration semantics that IP-XACT does not specify.

IEEE P3164 (Security Annotations for Electronic Design Integration). IEEE P3164 is an active standardization effort addressing security asset annotations for electronic design integration. The standard defines a vocabulary for marking security-sensitive assets at component boundaries: secrets, key material, security-domain crossings, and isolation requirements. SIG nodes naturally accommodate these annotations. A SIG node that represents a security-domain boundary carries P3164 security asset properties; the SIG edge across that boundary carries an additional contract category, security

isolation, alongside the ten functional categories. The framework's structure is unchanged; the category set expands when security-aware verification is in scope.

The pattern across all four standards is consistent: SIG is the integration-verification artifact that consumes existing standards' outputs and produces an executable subsystem verification plan. It does not require any of these standards to change. It does not require new EDA tooling. It organizes the artifacts the standards already produce into a graph whose edges are verifiable contracts and whose construction process is the four-step procedure described in Sect. [6.5](#).

6.14 VEGA Between IP and SoC Verification

VEGA is not a replacement for SoC-level verification. A full SoC integration introduces additional complexity, including software stacks, multiple subsystems interacting, chip-to-chip interfaces, and security boundaries, that is beyond the scope of this chapter. VEGA at the subsystem level occupies a specific and critical role: it is the earliest point in the design lifecycle where integration-space defects are visible and cheapest to fix.

The cost of finding a bug is not constant through the design lifecycle. A bug found during IP-level verification requires one engineer to fix one component. A bug found during subsystem integration requires coordination across component teams and may require architectural changes to the integration contract. A bug found during SoC integration may require RTL changes, testbench changes, and specification updates across multiple teams. A bug found in silicon requires a re-spin.

Subsystem verification performed before SoC assembly is the practical implementation of shift-left. The assumption harvest, the SIG construction, and the three scenario classes in this chapter are all activities that can begin before the subsystem RTL is complete. The SIG can be built from specifications alone, and the scenarios can be planned before integration begins.

The defects found this way are not new defects. They exist as soon as two components with misaligned assumptions are integrated into the same system. The only variable is when they are found. The data in Sect. [6.12](#) gives that variable a number: approximately $3.6\times$ per-ECO amplification when discovery shifts from subsystem to full-SoC scope.

6.15 Conclusion

Subsystem verification is a different activity from IP verification, with a different scope, different scenarios, and a different debug strategy. But it is the same framework: the same manifest, the same infrastructure discipline, the same XTP structure, and the same debug habits from Chap. [5](#).

What is new are the SIG, the ten-category intent taxonomy, and the assumption harvest that builds the SIG from existing IP-level XTPs and manifests, together making integration-space defects visible before integration begins. What is also new are the three scenario classes that cannot be written at IP level: boundary handoff, failure mode and reset propagation, and shared resource contention. These find the defects that kill tapeouts, not because any component is wrong, but because the space between components was never covered.

Forty defects caught at subsystem scope rather than SoC scope, a $3.6\times$ per-ECO amplification factor avoided, and a projected 48-week schedule recovery make the quantitative case. The qualitative case is simpler: defects do not become new at SoC scope, only more expensive. Chapter [7](#) turns to measuring the quality of the AI-assisted verification itself, the evidence on which a defensible sign-off ultimately depends.

7. Quality Metrics for AI-Assisted Verification

Vikash Kumar¹ 

(¹) Leander, TX, USA

 **Vikash Kumar**

Email: vikash.singh261@gmail.com

Pass rates and coverage report how well a testbench executed; they say nothing about whether it was asked to check the right things. Once a language model writes an entire UVM environment that a program signs off against, three questions open: is the generated code sound, does it verify the design's obligations, and can the generation be trusted? This chapter answers them with ten measurements in three families. The generation-side family, Repair Efficiency Score, Verification Gap, Specification Coverage Ratio, and Hallucination Rate, judges whether artifacts build, run, exercise the protocol, and name only real symbols; it rests on a taxonomy of eight failure modes of which compilation reveals one. The specification-side family, Programming Model Coverage, Contract Violation Latency, Descriptor Fidelity Score, and Observability Score, judges whether a testbench checks, and can even observe, obligations left only in prose, the gap a DMA controller exposed when its firmware interrupt-acknowledge step went unchecked as every test passed. The process-side family, Autonomy Index and Regeneration Stability, judges how much of the AI's output survived review and how repeatable its quality is. The

through-line: the strongest return is a more rigorous specification, not a stronger model. Together the ten numbers turn verification quality into a measurement the next chapter's sign-off can record.

7.1 Why Measurement Matters

The program has accumulated a great deal by now. Chapter [3](#) stood up a governed UVM environment; Chapter [5](#) ran it to a clean IP-level regression and a test plan that learned as it went; Chapter [6](#) folded the bridge into a subsystem and covered the seams between components. Throughout, AI helped in bounded ways, drafting scaffolds, proposing test-plan rows, flagging coverage holes, while the artifacts that carry judgment stayed in human hands: an architect wrote the XTP, a verification lead reviewed the manifest, engineers authored the scenarios.

What is left is to gather all of it into a defensible sign-off, the subject of Chap. [8](#). Before that argument can be made, one question has to be answered that none of the earlier chapters had to ask: what changes when AI does more than scaffolding? Once a language model writes a complete UVM testbench from a specification and that testbench is what the program signs off against, three questions must be settled before the Chap. [8](#) sign-off can hold:

First: is the generated testbench mechanically sound, that is, does it compile, elaborate, run, and drive the protocol behavior it was asked to drive?

Second: does that testbench check the things that matter, the firmware contracts, the boundary cases, the error paths, and the per-field completion checks the specification actually calls for?

Third: can the way it was produced be trusted, that is, how much of the AI's output survived review, and would a regeneration deliver the same quality again?

These are not the same question. They fail differently, they have different roots, and they need different metrics.

The usual verification dashboard cannot tell them apart. A regression that reports a 100% pass rate on five tests at full coverage closure says nothing about whether the testbench checks the right things: if the right things were never asserted, full coverage of what was asserted proves nothing about what was left out. Pass rates and coverage percentages grade *whether the testbench did its job*; none of them grades *whether that job was defined correctly when the AI generated it*.

This chapter adds ten metrics that close the gap, grouped into the three families the three questions imply. The first, Repair Efficiency Score (RES), Verification Gap (VG), Specification Coverage Ratio (SCR), and Hallucination Rate (HR), grades generation quality. The second, Programming Model Coverage (PMC), Contract Violation Latency (CVL), Descriptor Fidelity Score (DFS), and Observability Score (OBS), grades whether the testbench checks, and can even observe, the specification's obligations. The third, Autonomy Index (AUI) and Regeneration Stability (RGS), grades the generation process itself, how much of the output was trusted and how repeatable its quality is.

Between them the ten turn AI-assisted verification quality into a measurement rather than an assertion. A program that can state all ten has a defensible account of what the AI built, what got verified, and how far the generation can be trusted, and it carries that account into the Chap. [8](#) sign-off. A program that cannot has a green dashboard and an open question.

Figure [7.1](#) lays out how the three families fit: once the AI emits a testbench, one family asks whether it runs, one asks whether it checks

the right things, and one asks whether the way it was produced can be trusted, and all three feed forward into the Chapter 8 sign-off.



Fig. 7.1 The three metric families and the question each answers. After the AI emits the testbench, the generation-side family (RES, VG, SCR, HR) grades whether it runs, the specification-side family (PMC, CVL, DFS, OBS) grades whether it checks and can observe the right things, and the process-side family (AUI, RGS) grades how much of the output was trusted and how repeatable it is. All three feed the sign-off argument of Chap. 8

7.2 Two Failure Modes: Generation vs. Specification

Before the metrics, the two ways a generated artifact fails to verify a design. (A third concern, the trustworthiness of the generation process itself, is graded by the process-side family introduced later; it is not a failure mode of the artifact and so is not among the two here.)

When a language model writes a UVM testbench from a specification and the result does not verify the design, the failure has one of two roots. Each calls for a different repair, and confusing them spends engineering effort on the wrong layer.

7.2.1 Generation-Side Failures

A generation-side failure is a mechanical one: the AI emitted an artifact that does not work as code. It fails to compile or elaborate, hangs in simulation, drives the wrong pins, samples on the wrong edge, or

otherwise breaks the rules of UVM, SystemVerilog, or the protocol under test.

The root sits in how the model understood what it was writing. It produced code that reads as plausible but behaves wrongly, because it grasped the surface form of UVM without the semantic duties of the role it was filling. An APB slave driver that drives the master's outputs is the canonical example: valid syntax, wrong role.

These failures have nothing to do with how good the specification is. Hand a flawless, fully formal specification to a model that misreads UVM semantics and you still get a testbench that compiles and checks nothing. No specification can repair what the model misunderstands about its own output language.

7.2.2 Specification-Side Failures

A specification-side failure is the opposite: the artifact is mechanically fine, it compiles, elaborates, runs, and drives the protocol correctly, but it checks the wrong things. There is no monitor for an obligation the specification stated only in prose. The scoreboard confirms data but not completion order. The coverage plan walks the happy path and never the error path.

The root is upstream, in the specification handed to the model. It said what the device does; it did not say what the testbench must confirm. The model inferred the obligations the specification made explicit and, just as faithfully, failed to infer the ones it left implicit. The gap is above the model.

Adding model capability does not help here. A stronger model fed the same incomplete specification writes a more elaborate testbench that is still missing the same unnamed obligations.

7.2.3 Why the Distinction Matters

The two modes call for opposite responses.

A generation-side failure is debugged like any infrastructure defect: classify it (Chap. [5](#)), localize it, repair the testbench, re-run. Raising

generation quality across many programs is a tooling exercise, better prompts, better validation gates, better post-generation review.

A specification-side failure cannot be debugged inside the testbench, because the testbench is already correct as code. The repair is upstream: make the specification more formal, more structured, more complete. Skip that and a freshly regenerated testbench reproduces the identical hole.

Teams that fail to separate the two waste effort fixing testbenches the specification never equipped the model to write, and ship green dashboards on testbenches that run cleanly while checking nothing of substance. Both happen often; both are avoidable once the metrics distinguish the layers.

The next four sections introduce them. Sections [7.3](#) and [7.4](#) take the generation side; Sects. [7.5](#) and [7.6](#) take the specification side.

7.3 Generation-Side Failures: The Eight-Mode Taxonomy

Generation-side failures do not scatter randomly. Run AI at UVM testbench generation across enough programs and the same modes recur at predictable rates and, just as usefully, at predictable detection points. The taxonomy below comes from controlled runs of state-of-the-art commercial language models on protocol-bridge verification, the Educational Bridge Model of Appendix [A](#) that Chaps. [3](#) through [5](#) follow. These runs illustrate the metrics on the book's educational running example; the figures throughout this chapter are instruments for teaching the measurement framework, not a benchmark of models or a generalized empirical claim.

Eight modes account for most of what went wrong in those runs. They are ordered by the stage at which each becomes visible, which is the property that matters operationally: the stage fixes the cost, and the cost decides which validation gates earn their keep.

Where a failure surfaces is what it costs. Catching it at compile time costs minutes; at elaboration, tens of minutes; in simulation, hours of debugging. Among the eight modes below, exactly one is reliably caught at compile and one at elaboration; the other six appear only in simulation or while reading waveforms. **Compilation, then, finds one mode in eight.** That is why compile-success rate, the metric most current LLM-for-hardware work leans on, tracks testbench correctness only weakly.

Figure 7.2 draws the split: compilation accounts for one mode, elaboration for one, and simulation for the remaining six, with the cost of detection climbing at each stage.

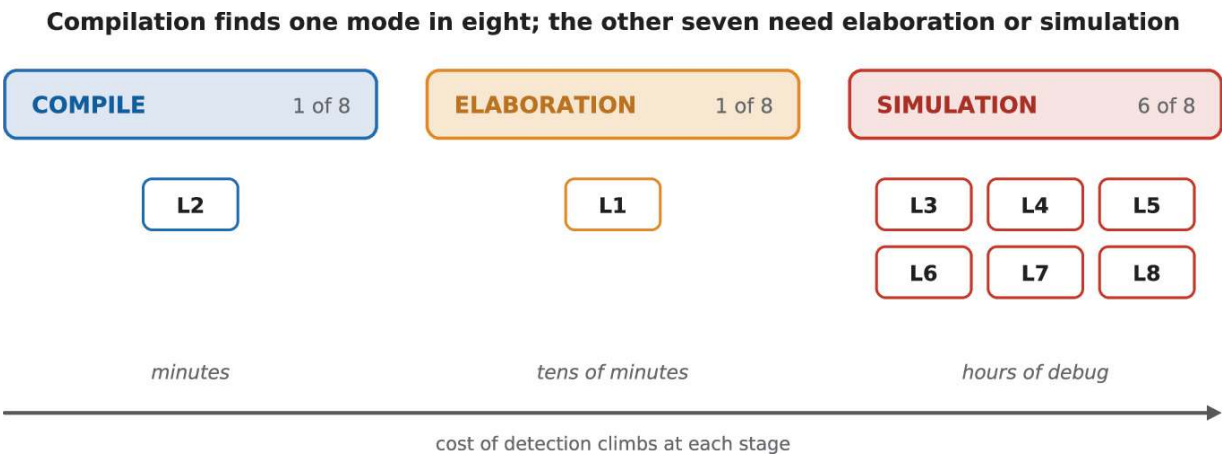


Fig. 7.2 Where the eight generation-side modes are caught. One is diagnosable at compile, one at elaboration, and six only in simulation, the most expensive stage. Because compilation accounts for a single mode, compile success is a weak proxy for a correct testbench, and the Verification Gap metric is what measures the remainder

Table 7.1 lists the modes with the stage that first exposes each and a characteristic signature.

Table 7.1 The eight LLM testbench-generation failure modes, ordered by detection stage: Compile, Elaboration, or Simulation. That simulation dominates the stage column is the core argument against treating compile success as a quality measure for generated testbenches

The taxonomy came out of a controlled run on the bridge in which a state-of-the-art commercial LLM emitted roughly twenty-five files that looked like a complete UVM environment. An agentic repair loop cleared thirty-seven compile errors over four passes, and the result compiled. It then hung on the first transaction, drove wrong data on every transfer, and logged zero scoreboard checks. The ensuing eight-hour debug session turned up eight distinct generation-failure modes. Chapter [5](#), Sect. 5.5 narrates that session in full as a classification exercise, so it is not repeated here; the point worth carrying forward is the split, compile caught one mode (L2, once its cluster collapsed), elaboration caught one (L1), and the other six were simulation-stage finds.

7.3.1 Error Clustering

One practical regularity: generation-side errors clump around a shared root cause. In the bridge run, a single repair that addressed the L2 hallucinated-fields mode cleared eighteen downstream compile errors at once, all of them tracing to the same wrong idea about which fields the sequence-item class declared.

That clustering has two consequences. It explains why repair-loop iteration counts do not track repair-loop effort: a few root fixes can dissolve a much larger error tally. And it argues for prompts that make the model find the root cause before patching, which run leaner than prompts that chase errors one at a time.

Both points lead into the first generation-side metric, repair efficiency.

7.4 Generation-Side Metrics: RES, VG, SCR, HR

Four metrics put numbers on the failure modes just described. Each captures a different facet of generation quality, and together they map the distance between a generated testbench that merely runs (compiles, executes, terminates) and one that actually verifies the protocol semantics it was assigned.

7.4.1 Repair Efficiency Score (RES)

RES reports how much work each automated repair call does when clearing compile errors in generated testbench code.

$$\text{RES} = \frac{\text{total compile errors observed}}{\text{total repair calls invoked}}$$

Higher is better. Above 1 means the average repair call clears more than one error; exactly 1 means errors fall one per call; below 1 means the loop spends more calls than the errors it removes.

On the bridge, $\text{RES} = 37/15 = 2.47$: across fifteen repair calls the loop cleared thirty-seven distinct compile errors. That average of about two and a half is skewed by one call that dissolved eighteen errors at once, the hallucinated-fields cluster; the other fourteen calls sat just above one error each.

RES works best as an early read on repair-loop health. A program hovering near 1.0 over several sessions is patching symptoms without finding roots, which forecasts long repair loops on the next generation. A program well above 1.0 is finding clustered failures and fixing them at the source, a property worth protecting when prompts get edited.

RES says nothing about whether the finished testbench is correct. A loop scoring $\text{RES} = 5.0$ can still hand back something that compiles and verifies nothing. The next two metrics take that on.

7.4.2 Verification Gap (VG)

VG reports the share of functional failures that outlive a clean compile.

$$\text{VG} = \frac{\text{functional failures remaining after compile-clean}}{\text{total functional failures identified}}$$

Lower is better. At 0.00 the testbench is both clean-compiling and functionally complete; at 1.00 every functional failure is still present once it compiles. The number isolates exactly what compilation does not catch.

On the bridge, VG against the eight-mode taxonomy stood at 0.75 once the automated repair loop finished: of the eight functional modes present, six survived the clean compile. Compilation and elaboration removed two; the remaining six were invisible to them and needed simulation to surface.

VG is the number the field mostly leaves uncomputed. The prevailing LLM-for-hardware benchmark reports compile-success rate, in effect one minus a compile-failure rate. That speaks to syntactic competence and stays silent on whether anything is verified. A model at 95% compile success and 0.80 VG ships testbenches that build dependably and check next to nothing; a model at 60% compile success and 0.10 VG builds less reliably but, when it does build, is nearly complete. The first wins the leaderboard. The second is the one to put on a real project.

VG also earns its keep in tool selection. Choosing between two generators, the question is not which emits more clean-compiling output but which clean-compiling output needs the least functional repair downstream, and VG on a representative protocol answers that with a number.

7.4.3 Specification Coverage Ratio (SCR)

SCR reports how much of the protocol specification the testbench actually drives.

$$\text{SCR} = \frac{\text{specification behaviors exercised by the testbench}}{\text{total specification behaviors required to be exercised}}$$

Higher is better. At 1.00 every specified behavior is exercised; below 1.00 some specified behavior goes unreached even by a testbench that

compiles cleanly and behaves correctly on what it does run.

SCR catches what the first two miss: a testbench that is mechanically sound, runs to completion, and still touches only part of the protocol's required behavior. That is not VG. VG asks whether the generated code works; SCR asks whether the work it does spans the specification.

For the bridge, the specification calls for NONSEQ, SEQ, and BUSY transfer types; OKAY and ERROR responses; the two-cycle ERROR sequence; back-to-back transfers with no IDLE; transfers across reset boundaries; and more. A generated testbench that drives NONSEQ with OKAY on a single master, the model's comfortable default, lands well under 1.0 on SCR even when its VG is near zero.

SCR ties straight to the XTP from Chap. 2: the XTP names the specification behaviors the program means to drive, and SCR is the fraction of them the generated testbench actually drives in regression. An SCR shortfall is a coverage gap of a particular kind, not the functional-coverage gap of Chap. 8 but a gap in the specification coverage of the generated stimulus.

Here the Verification Strategist reads the generated testbench against the XTP and names the specified behaviors it never reaches. This is one of the tasks AI cannot be trusted to automate: telling apart what the testbench does from what the specification requires is a judgment about intent, the same judgment the XTP encodes to begin with.

7.4.4 Hallucination Rate (HR)

HR reports how often the generated code refers to something that does not exist, a signal, a sequence-item field, a method, a parameter, or a type the package, interface, or specification never declares.

$$\text{HR} = \frac{\text{symbol references that resolve to nothing}}{\text{total external symbol references in the generated code}}$$

Lower is better. At 0 every name the code uses resolves to a real declaration; a high HR means the model is inventing the design it is supposed to be writing against.

Hallucination is the most distinctly machine failure in the taxonomy. A human engineer mistypes a name; a language model confabulates a whole vocabulary that reads as plausible and refers to nothing. Mode L2 is HR made concrete: the generated sequence item named `size_type` and `rw_type` against a class that declared `xfer_size` and `xfer_rw`, and that single invented abstraction stood behind eighteen of the run's thirty-seven compile errors.

HR sits upstream of RES. A high hallucination rate is the cause; a long repair loop and a low Repair Efficiency Score are the symptoms. Because confabulations cluster, one wrong idea spawning many references, HR also explains why repair effort tracks root causes rather than raw error counts: clearing the hallucinated abstraction clears its whole cluster at once.

HR is the one generation pathology that compile-success benchmarks partly see, since most hallucinated symbols fail to resolve at compile or elaboration. But a raw error count is not a rate. HR normalizes the confabulation against the size of what the model wrote, so three bad references in thirty reads very differently from three in three hundred: the first is a slip, the second is a model that did not have the design in view.

7.4.5 The Metrics Read Together

The four are not independent; they view one question, did the AI build a testbench that does its job, from different angles, and a program that reports all four sees more than any single number shows. Hallucination Rate is the leading indicator among them: it is read first, at compile time, and a high HR forecasts the repair load the other three then quantify.

Three combinations of the run-and-cover metrics recur:

High RES, high VG, low SCR. The loop clears errors efficiently and the testbench runs, but it drives only a slice of the specification. This is the usual signature of AI-generated testbenches on protocol designs, an artifact of the model gravitating toward simple, clean-compiling paths.

Low RES, low VG, low SCR. The loop is inefficient, many functional failures persist, and coverage of the specification is narrow. The worst case, and usually a sign the model does not grasp the protocol or the verification task structurally.

High RES, low VG, high SCR. Reached only with careful prompting, extra specification structure, and post-generation review. The target state.

The chapter's next sections pursue that third combination, how specification structure turns the model's mechanical competence into broad coverage at low verification gap. That means crossing from generation-side failures to specification-side ones, where the next four metrics live.

7.4.6 The Bridge, Scored

The generation-side family is the part of this chapter grounded entirely in the book's running example. Collecting the bridge's numbers in one place: its repair loop scored RES = 2.47 against a hallucination rate high enough that one invented abstraction drove eighteen of its compile errors, its post-compile Verification Gap was 0.75, and its Specification Coverage Ratio sat well below 1.0 until directed stimulus filled in the transfer types, response codes, and reset-boundary behavior the model skipped by default. Read together, they say the same thing about the generated bridge testbench: it was repaired efficiently and made to run, but compiling it settled almost none of its verification debt, and left to its defaults it exercised only a corner of the specification.

Three of the four specification-side metrics do not have a bridge number, and the reason is structural rather than incidental. PMC, CVL, and DFS measure obligations that live in a programming model: descriptor fields, completion records, an interrupt-acknowledge handshake. The bridge has none of these. It is a stateless protocol

translator with no descriptors to score and no firmware contract to leave unchecked. Observability Score is the exception, since it applies to any design with obligations and oracles: the bridge scoreboard checks every forwarded transfer, so its OBS is high. But PMC, CVL, and DFS need a device that carries programming-model structure. To show what those metrics catch, the chapter needs such a device, which is why the next sections move to a DMA controller. The bridge proves the generation-side case; the DMA is required to prove the specification-side one.

7.5 Specification-Side Failures: The Firmware-Hardware Contract

Even a generated testbench with strong generation-side numbers, low VG, high RES, respectable SCR, can fail to verify a class of behavior the specification never formalized. This section pins that class down with a worked case: a controlled run in which a state-of-the-art agentic AI produced a full RTL design and UVM testbench for a two-channel AHB-Lite DMA controller, and the testbench worked as code while an entire RTL path went unverified.

The case stands apart from the bridge on purpose. The DMA controller was picked for a structural feature the bridge lacks: a firmware-hardware contract that lives outside the protocol's signal interface.

7.5.1 What the AI Generated

Handed a natural-language specification of the DMA controller, the agentic system produced more than fifteen SystemVerilog files, the RTL, the UVM environment, the scoreboard, three progress monitors, a sequence-telemetry agent, and five test classes. A capable junior engineer would have spent two to three weeks on that; the AI did it in a fraction of the time, with every test compiling and passing and the simulation running clean.

Reading what the testbench actually checked told a more complicated story. The AI had built the descriptor fields right, wired the channels right, and laid out the UVM hierarchy the way an experienced engineer would, and its observability was strong enough to expose six real RTL bugs in simulation along the way.

The miss was elsewhere. The AI never modeled the firmware duty the hardware existed to serve, and from a prose specification it had no way to know it should. That omission left a whole class of failure unchecked, and it is what the four specification-side metrics in the next section exist to measure.

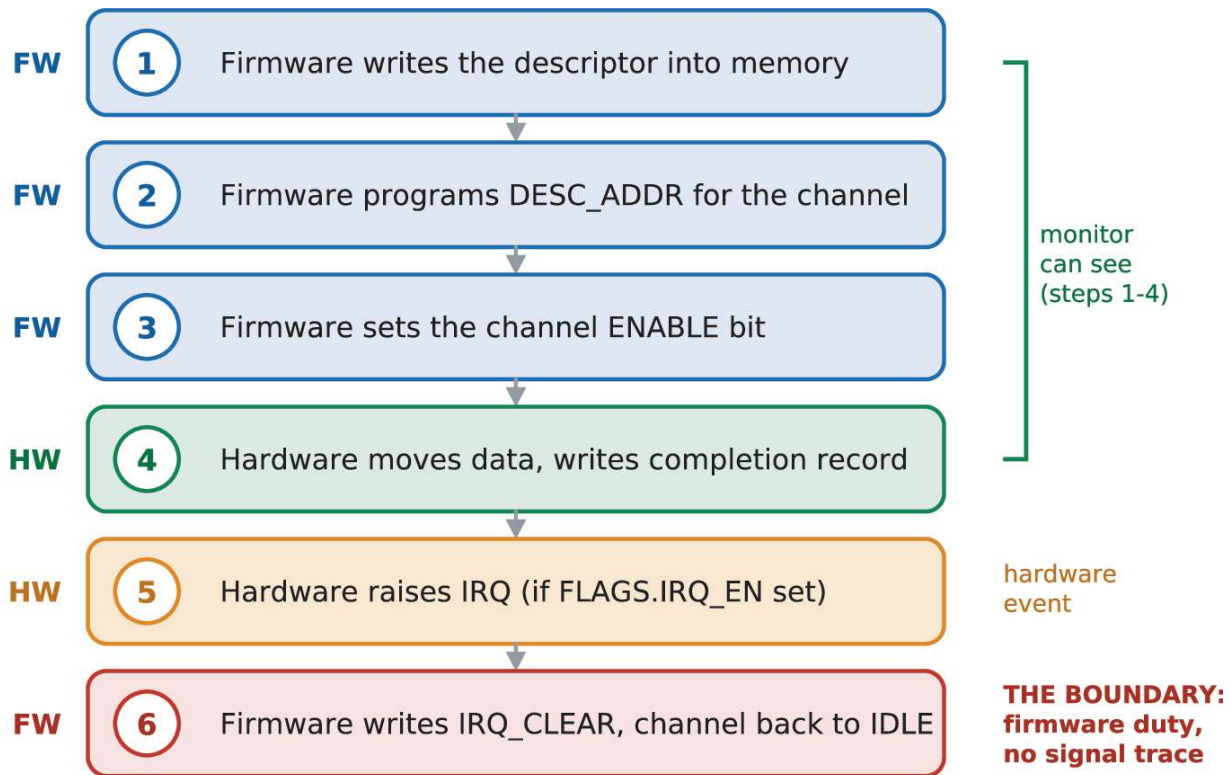
7.5.2 The Programming Contract

Every descriptor-based device comes with a programming model, the exact sequence of operations firmware must run to submit work, collect results, and acknowledge completion. That model is part of the correctness contract. A device whose RTL implements every state machine perfectly but whose firmware contract is broken is, as far as the system is concerned, broken.

For the DMA controller here, the contract ran:

1. Firmware writes the descriptor into memory.
2. Firmware programs the descriptor's address into the channel's `DESC_ADDR` register.
3. Firmware sets the channel `ENABLE` bit.
4. Hardware works the descriptor, moves the data, and writes the completion record.
5. Hardware raises the IRQ when the descriptor's `FLAGS` field had `IRQ_EN` set.
6. **Firmware writes `IRQ_CLEAR` to acknowledge**, returning the channel to IDLE.

Figure 7.3 draws this contract and marks the one step the protocol's signal interface cannot see.



Skip step 6 and the channel wedges in ST_IRQ while the testbench still prints PASSED

Fig. 7.3 The firmware-hardware contract for a descriptor-based DMA controller. Steps 1 to 4 leave signal-level traces the monitor can see, descriptor writes, register accesses, the data transfer, the completion record. Step 5 is a hardware event; step 6 is a firmware duty with no signal that separates firmware which performs it from firmware which skips it. Step 6 is the boundary: skip it and the channel blocks for good while a signal-level testbench still prints PASSED

Step 6 is the boundary. A testbench that issues correct descriptors and confirms correct completions but never models the IRQ-clear duty will print PASSED on a channel that is wedged: the data landed, sixteen bytes reached the destination, the UID matched, the byte count was right, and the channel now sits in ST_IRQ indefinitely because nothing on the firmware side wrote IRQ_CLEAR to drop the interrupt and return it to ST_IDLE.

This is the failure class behind all four specification-side metrics. None of them scores test pass or fail in the usual sense; each scores

what the testbench never asserts, or cannot observe, about behavior the specification only implied.

7.5.3 Six Real Bugs the AI Testbench Found Anyway

One observation reframes everything above. For all the unverified firmware contract, the generated testbench did its job in one real sense, it isolated six genuine hardware defects in the AI-written RTL during simulation. None threw a compile error, none crashed the simulator; each produced wrong results quietly and was caught only because the testbench's observability happened to include the right monitor.

The six fell into four categories that recur in LLM-generated RTL:

Language scoping failures. SystemVerilog's separate compilation needs explicit package declarations for cross-file class visibility. The AI spread transaction definitions over fifteen files with implicit dependencies among them; one package declaration cleared twenty-seven apparent errors that had all collapsed out of a single misread of compilation scope. *The pattern: LLM errors gather around a misunderstood abstraction, so fixing the abstraction clears many at once.*

Stale pipeline data. The descriptor-validation logic read the queue's combinational output one cycle after the pop, by which point the queue had moved on. Every valid descriptor quietly produced `DESC_INVALID`; not one transfer happened. Only the scoreboard's completion tally exposed it.

Protocol pipeline timing. AHB-Lite wants a one-cycle gap between the address and data phases. The generated RTL drove both together, which hung the WRITE state for 256 cycles and corrupted the data in every completion record. The AI got the protocol's structure right and missed its pipeline timing. *Expect this at any pipelined interface, AHB, AXI, and the like.*

Concurrent-channel arbitration races. A round-robin arbiter re-deciding grants every cycle could move off a channel between the address and data phases of one AHB beat, so Channel 0's data phase carried Channel 1's descriptor, wrong data, wrong destination, no error

flagged. Invisible with one channel; it showed only under out-of-order multi-descriptor traffic, exactly what a DMA regression is built to drive.

High coverage is not what surfaced these bugs. They surfaced because the scoreboard tallied completions, the monitors watched FSM state every cycle, and a UID-based tracker tied out-of-order completions back to the descriptors that produced them.

Observability infrastructure catches bugs on the first run; coverage completeness catches the stragglers over the weeks that follow. Both matter, and they are not the same thing.

This matters because coverage and observability are different axes. Specification coverage, contract violation latency, and descriptor fidelity ask whether the right behavior was exercised and checked; observability asks whether the testbench could catch a violation at all. A testbench with flawless coverage numbers but no observability catches nothing on its first run; a testbench with poor coverage but strong observability catches six bugs on its first run while leaving a whole RTL path unverified. Both happened in the case above. The fourth specification-side metric, the Observability Score, puts a number on that second axis, and the next section defines it alongside the three coverage metrics.

7.6 Specification-Side Metrics: PMC, CVL, DFS, OBS

Four metrics put numbers on specification-side failures, each on a different part of the gap between what the AI built and what a complete verification plan would demand. The first three grade coverage, whether the right behavior was exercised and checked; the fourth

grades observability, whether the testbench could catch a violation at all.

7.6.1 Programming Model Coverage (PMC)

PMC reports the share of the device’s behavioral space the generated testbench actually reached, measured after the fact from what the AI produced rather than against a plan written in advance.

$$PMC = \frac{\sum_{i=1}^N (\text{bins observed in dimension } i)}{\sum_{i=1}^N (\text{bins defined in dimension } i)}$$

Higher is better. The figure separates what the model covered on its own from what would have taken directed engineer effort.

PMC spans a set of behavioral dimensions chosen for the device class. For a descriptor-based DMA controller, seven capture the operating behavior: channel utilization, descriptor field combinations, completion status codes, error response codes, out-of-order completion ordering, fault-injection scenarios, and IRQ delivery paths.

In the case study PMC came to 44.1%. The aggregate is informative, but the per-dimension breakdown says more. Table 7.2 lists what the generated testbench actually drove across the seven.

Table 7.2 Per-dimension PMC for the DMA controller. Channel utilization, out-of-order ordering, and fault scenarios hit 100%, the dimensions the AI was built to drive; completion status reached only 20%, the decisive gap

The shape matters more than the headline. Channel utilization, out-of-order ordering, and fault scenarios all reached 100%, the dimensions with purpose-built directed tests. Completion status codes reached 20%: of the ten status and error-code bins in the specification, only `STAT_SUCCESS` and `ERR_NONE` appeared across all five tests. Happy-path tests came naturally; error paths needed directed fault injection. The IRQ dimension scored zero, the testbench drove `IRQ_EN`

but never finished the firmware handshake, so the IRQ-deassert path stayed dark.

The breakdown reads more usefully than the total. *The AI converged on the dimensions that protocol VIPs and stock test classes target, and gave near-zero attention to the ones that need specification-level reasoning, status-code coverage and IRQ-contract enforcement.* That is not a deficiency in any strict sense; it is the model correctly concluding that a specification which describes those dimensions in prose, with no instruction to exercise them, was not asking it to.

7.6.2 Contract Violation Latency (CVL)

CVL reports how long it takes, from the instant a hardware misbehavior or a testbench hole becomes observable, until the testbench actually flags it. It is reported per fault class, not as one aggregate.

$$\text{CVL}_{\text{fault}} = t_{\text{detected}} - t_{\text{detectable}}$$

Lower is better. Zero is immediate detection; ∞ means the testbench has no way to detect that fault class at all, which is a different problem from detecting it slowly.

The property that makes CVL worth tracking is the gap between slow and infinite. A CVL of a million simulation nanoseconds is slow detection, fixable by shrinking the global timeout. A CVL of ∞ is the absence of any detection mechanism, which no timeout addresses. The two need different fixes, and an aggregate that averages them buries the worse one.

The DMA case study instrumented three fault scenarios, shown in Table [7.3](#):

Table 7.3 Three DMA fault scenarios and their contract violation latencies. F1 and F2 are slow but bounded; F3 is unbounded because the failure class has no monitor

F1 and F2 are the ordinary CVL story, detection at 595 ns and 1735 ns, comfortably under the million-nanosecond global timeout. Useful, but not the headline.

F3 is the headline. The scoreboard printed PASSED on F3, and by every signal-level oracle it had, it was right, the transfer finished, sixteen bytes arrived, the UID matched, the byte count held. What it could not see was that the RTL's IRQ-deassert path had never run. The testbench raised `IRQ_EN=1` and never wrote `IRQ_CLEAR`, so the logic that drops the interrupt, resets the channel, and returns it to `IDLE` was never triggered, checked, or covered, and the channel stuck in `ST_IRQ`. Without a monitor, F3's CVL is not slow but infinite: the run simply exits PASSED with no sign that a whole RTL path was skipped.

The distinction that should change your debugging. A CVL of 1,000,000 ns is slow detection; shrink the global timeout and it improves. *A CVL of ∞ is no detection at all, and no timeout setting touches it.* Only the IRQ monitor exposed that the whole IRQ path had gone unexercised.

7.6.3 Descriptor Fidelity Score (DFS)

DFS reports testbench quality one descriptor field at a time, the descriptor being the unit of work for any descriptor-based device. Each field earns three independent marks: *generate* (was correct stimulus driven), *check* (was the field validated in the completion record), and *boundary* (were edge values exercised).

$$\text{DFS} = \frac{\text{generate}_{\text{hit}} + \text{check}_{\text{hit}} + \text{boundary}_{\text{hit}}}{3 \cdot N_{\text{fields}}}$$

Higher is better, and all three sub-scores are reported separately; a single aggregate hides which capability is the weak one.

In the DMA case study the descriptor had seven fields, and the sub-scores were generate 7/7 (100%), check 3/7 (43%), and boundary 1/7 (14%), an aggregate DFS of 52.4%. Table [7.4](#) gives the per-field breakdown.

Table 7.4 DFS by descriptor field. Three independent capabilities: the LLM understood the hardware mechanism fully (generate), understood only part of what to validate back (check), and had no systematic sense of which values to stress (boundary).

The three sub-scores are independent capabilities, and they fell off independently. The model had the hardware mechanism cold at 100% generate, knew only part of what to check back at 43%, and had no systematic stress model at 14% boundary.

FLAGS is the concrete instance of the check gap that set up this whole chapter: the AI drove `IRQ_EN` correctly but never wrote the matching checker, write `IRQ_CLEAR`, confirm the IRQ drops, confirm the channel returns to `IDLE`. That RTL path was generated and never verified. DFS records it exactly: FLAGS scores generate = 1, check = 0, boundary = 0, contributing one-third of a single field to the total.

7.6.4 Reading the Three Sub-Scores Together

Across LLM-generated testbenches the three sub-scores drop off in a consistent order, generate climbs first, check trails it, and boundary lags well behind, because each asks a different kind of inference of the model.

Generate asks the model to understand the device’s data structure. Any specification spells out the descriptor format, so generation is close to direct translation.

Check asks the model to understand what counts as a correct completion. Sometimes the specification says so outright (“the completion record carries the byte count”); often it only implies it, and the implied obligations are the gap.

Boundary asks the model to understand which field values stress the implementation. A specification almost never spells that out to the

depth needed for systematic boundary tests, so boundary scores tend to reflect what a careful engineer adds afterward rather than what the model derives from prose.

The three fall off independently, and each has a different upstream fix: a sharper data-structure description (rarely needed, usually already adequate), a sharper validation contract (the common gap), and explicit boundary obligations (almost always a deliberate engineering act).

7.6.5 Observability Score (OBS)

OBS reports the share of the design's obligations that the testbench can actually catch, an obligation being observable when a checker, monitor, or assertion exists that fails if the obligation is violated.

$$\text{OBS} = \frac{\text{obligations with a live oracle}}{\text{obligations that require one}}$$

Higher is better. At 1.0 every obligation can be caught; below 1.0 some obligation can be violated while the testbench prints PASSED.

OBS is the metric the rest of the chapter kept pointing at. The coverage numbers, PMC here and the closure numbers of Chap. 8, all assume an oracle is watching: a bin that closes on an obligation with no checker means the stimulus ran and nothing judged the result. OBS is what makes coverage meaningful, and the two are orthogonal. A testbench can post high coverage and low observability, which is the most dangerous combination in this chapter, exercised behavior that nothing checks.

The DMA case is OBS read end to end. Six bugs were caught on the first run because the oracles existed, the completion-tally scoreboard, the per-cycle FSM monitors, the UID tracker. The F3 IRQ fault was missed because, for that one obligation, no oracle existed: OBS for the IRQ-clear contract was zero, which is the same fact CVL records as ∞ and DFS records as `FLAGS check = 0`. OBS generalizes those per-fault

and per-field readings to the whole testbench, the fraction of everything the design must guarantee that the testbench is equipped to falsify.

AI generates observability unevenly for a specific reason: a model builds oracles for the obligations a specification states outright and omits them for the ones it only implies, exactly the boundary the firmware-hardware contract exposed. OBS turns that tendency into a number a reviewer can act on before sign-off rather than after silicon.

7.7 Process-Side Metrics: Autonomy and Reproducibility

The first two families grade the artifact: whether the generated testbench is built right, and whether it checks the right things. A third question sits beside them and is easy to leave unasked, how much of what the AI produced did the program actually trust, and would a second run have produced it again. Two metrics answer it. They do not grade a testbench; they grade the act of generating one.

7.7.1 Autonomy Index (AUI)

AUI reports how much of the AI's output survived, unmodified, into the testbench the program signed off against.

$$\text{AUI} = \frac{\text{AI-authored content retained unchanged at sign-off}}{\text{total content of the final artifact}}$$

Higher means more leverage delivered, but only when read against the artifact metrics: surviving unmodified is a virtue only if the surviving code was also correct and observable.

AUI puts a number on the claim that “the AI wrote the testbench.” That is rarely wholly true; the honest figure is the fraction an engineer kept after review and repair. A high AUI is the goal, but only in company. High AUI with a high Verification Gap is the danger reading, not the

success one: it means a large body of generated code shipped without the review that would have found its functional holes. AUI is therefore never read alone. Read against VG and OBS, it converts “how much did the AI do” from an impression into a measured, reviewable share.

7.7.2 Regeneration Stability (RGS)

RGS reports how consistent the AI’s output quality is across independent regenerations from the same specification.

$$\text{RGS} = 1 - \frac{\sigma(Q)}{\mu(Q)} \quad \text{over } K \text{ regenerations,}$$

where Q is a chosen quality metric (VG, SCR, or PMC) measured on each run.

Higher is better. Near 1.0 the same specification yields near-identical quality every time; a low RGS means a single good or bad measurement was partly luck.

RGS bounds the confidence of every other number in this chapter. Language-model generation is non-deterministic: the same prompt and the same specification can produce materially different testbenches from one run to the next. A Verification Gap of 0.10 measured once is a fact about one run; a Verification Gap of 0.10 with high RGS is a property of the tool and the specification together, the only version of the claim a program can plan around. RGS is what tells a team whether the measurement it just took will hold when the testbench is regenerated after the next specification revision.

Crucially, RGS answers to the same lever as the rest of the chapter. A vague prose specification leaves the model wide latitude, and latitude is variance, so RGS is low. A structured specification narrows the space of plausible testbenches, the regenerations converge, and RGS rises.

Reproducibility, like coverage and observability, is bought with specification rigor, not with a steadier model.

7.8 The Unifying Thesis: Specification, Not Model

Different layers, one conclusion. The 44.1% PMC and 52.4% DFS of the DMA case are specification failures before they are model failures.

Completion-status coverage stalled at 20% because the error behavior lived in prose: “`STAT_TIMEOUT` is asserted when the timeout counter expires before completion, and the matching error code is written to the completion record.” That tells a reader what the device does. It does not tell the model to generate a test that hits every status-code bin.

Boundary coverage scored 1/7 because the specification gave field widths without saying what to stress. It said `LENGTH` is sixteen bits; it did not say boundary tests must include 0, 1, 2, the maximum, the maximum minus one, and unaligned values. Absent that, the model reasonably concluded no particular values were required and emitted one representative value.

The F3 IRQ gap exists because the handshake was narrated: “software must clear the interrupt by writing `IRQ_CLEAR`.” That tells the AI the register exists. It does not tell it to write a test that exercises the register, confirms the IRQ drops, and confirms the channel returns to `IDLE`. The AI drove the field that armed the contract and never built the checker that should follow.

A structured specification schema closes each gap by turning the prose obligations into fields a generation pipeline can read directly:

Structured specification schema: IRQ contract and completion coverage

```
irq_contract:
  trigger:          transfer_complete AND irq_en = 1
  response_required: irq_clear_write
  timeout_cycles:   IRQ_TIMEOUT
  violation_class:   firmware_contract
  expected_state:   channel returns to ST_IDLE
                   after irq_clear_write

completion_coverage:
  required_bins:
    - STAT_SUCCESS
    - AHB_ERROR
    - STAT_TIMEOUT
    - DESC_INVALID
    - STAT_PARTIAL
    - ERR_NONE
    - ERR_BUS
    - ERR_TIMEOUT
    - ERR_PROT
    - ERR_DESC
  boundary_lengths: [0, 1, 4, 65532, 65535, unaligned]
```

Given a schema like this, the AI derives monitors, coverage bins, and boundary tests straight from the fields.

`irq_contract.violation_class` tells it to build a checker for the contract, not just stimulus for the trigger;

`completion_coverage.required_bins` names exactly which bins must close; `boundary_lengths` names exactly which values to stress. Without the schema the model rightly declines to infer what the specification never stated; with it, the model emits the very artifacts that PMC, CVL, and DFS would otherwise have to expose after the fact.

This chapter's central claim:

The largest return in LLM-based verification automation does not come from a more capable model. It comes from a more rigorous

specification.

The PMC shortfall and the DFS boundary shortfall both trace straight back to obligations that lived only in prose. Double the model's capacity against the same prose and the same holes appear; halve it but hand it a structured schema and the holes close.

This claim links straight back to the framework built in the earlier chapters. The XTP of Chap. [2](#) is exactly the structured specification artifact this section argues for, features, interactions, and expected results, written from the specification at plan time in a form humans and AI tools can both reason over. The manifest of Chap. [3](#) plays the same role at the testbench-architecture layer; the SIG of Chap. [6](#) extends the principle to integration. Each captures intent the prose specification leaves implicit.

The metrics here measure what happens when the AI gets prose instead of those artifacts. The holes they expose are the holes the framework's structured artifacts exist to close. Run the framework backward through the metrics and the point lands: the XTP, the manifest, and the SIG are not filing conveniences, they are the specification infrastructure that turns the model's mechanical competence into real verification leverage.

7.9 Evaluation and Adoption Guidance

This section speaks to three audiences, teams choosing an LLM-based verification tool, teams measuring a pipeline they already run, and teams writing the specifications that feed AI generation, and the ten metrics answer a different question for each.

7.9.1 Evaluating LLM-Based Verification Tools

A vendor demoing a testbench generator almost always leads with compile-success and pass rates on a benchmark suite. Both track verification quality weakly. Ask instead, in order of weight:

VG on a real protocol. 95% compile success with VG 0.80 means testbenches that build reliably and verify almost nothing; 70% compile success with VG 0.10 means they build less reliably but, when they build, are nearly complete. Adopt the second. A vendor who cannot produce VG on a real protocol has not measured the tool's verification quality.

DFS with all three sub-scores on a descriptor-based design. DFS = 70% from sub-scores 100/60/50 is a different tool from DFS = 70% from 80/80/50. The aggregate hides the profile, so ask for the parts; the boundary sub-score is usually lowest and is the clearest tell of whether the tool generates verification intent or only stimulus.

CVL per fault class on a multi-fault benchmark. An aggregate CVL of 100,000 ns can be slow detection on most faults averaged with infinite CVL on one, and those need different fixes, so ask for the distribution.

RES over several repair sessions. RES near 1.0 session after session is symptom-patching; RES near 2.5 is root-cause fixing. At scale the difference compounds.

OBS and HR on the same benchmark. Observability Score is the cap on every coverage number the tool will later report: it says how much of the design its testbench can catch at all. Hallucination Rate, read early, says how much of the output names symbols that do not exist, the leading sign of a long repair loop ahead.

RGS across repeated generations. A strong score from one run is one run. Ask the vendor to regenerate from the same specification several times and report the spread; a low Regeneration Stability means the demo number will not survive the next revision.

7.9.2 Measuring an Existing Verification Pipeline

For a pipeline already running with AI assistance, the metrics give a baseline to measure improvement against.

Track VG and SCR per generated testbench, summed across the program. A VG that falls as the team learns the tool's failure modes is

progress; a flat VG means the program is being built around the tool's gaps rather than closing them.

Track CVL per fault class as a standard regression diagnostic. The fault classes are the ones directed tests already drive, so CVL rides on top of existing tests, and a regression report carrying CVL per class makes infinite-CVL holes visible at once instead of post-silicon.

Track DFS per descriptor-based component as part of the sign-off package. Across several such components, DMA controllers, network interfaces, accelerator fabrics, the DFS sub-score pattern flags which ones have weak specifications and need more structured documentation before the next program reuses them.

Track OBS as the gate on those coverage numbers, and AUI as the honest record of how much of each generated testbench the program actually kept. A high AUI next to a high VG is not a success but a warning, generated code that shipped without the review its functional gaps needed.

7.9.3 Writing the Specification That Feeds AI

The audience that matters most is the one writing the specifications that feed AI generation. The PMC and DFS holes a program finds after the fact are nearly always upstream omissions, and closing them is a matter of deliberate specification practice. Three habits close most:

Encode contracts as structured fields. The IRQ-contract example in Sect. [7.8](#) is the template: every behavior the device implements as a contract, transfer completion, error acknowledgment, configuration handshake, power-state transition, becomes a schema field with an explicit trigger, response, timeout, and violation class, and the model builds checkers from it without inference.

Enumerate the required coverage bins. “The completion code can be `STAT_SUCCESS`, `STAT_TIMEOUT`, `STAT_PARTIAL`, or `STAT_ERROR`” does not drive closure on that dimension; `required_bins: [STAT_SUCCESS, STAT_TIMEOUT, STAT_PARTIAL, STAT_ERROR]` does, because the list removes the inference the model would otherwise have to make.

State the boundary obligations. A field width is not a boundary obligation; `boundary_values: [0, 1, maximum, maximum-1, unaligned]` is. Name the values and the testbench drives them; name only the width and it drives the default and nothing more.

These three are the working form of the thesis in Sect. [7.8](#), turning prose into structure and implicit obligations into explicit ones, and they make the model's output only as good as the specification it was handed, which, practiced deliberately, is good enough to verify the design.

7.9.4 Connection to the Sign-Off Argument

These metrics feed straight into the sign-off argument Chap. [8](#) assembles: when AI-generated testbench artifacts are part of the record, the residual-risk register that chapter builds has to carry their metric values at sign-off.

A sign-off that calls an AI-generated testbench verified without recording its VG, OBS, PMC, CVL distribution, and DFS sub-scores is a sign-off that does not know what it verified. The metrics do not alter the structure of Chap. [8](#)'s argument; they fill the modeling-assumptions slot of its residual-risk register with specific numbers, *this testbench was AI-generated; its VG was 0.20; its DFS check sub-score was 60%; the seven percent of behaviors these metrics flag as unverified is on the deferred-behaviors list*.

That is what defensible AI-assisted sign-off looks like. The testbench may be AI-generated; the artifacts may carry measured gaps; the gaps sit in the register with numeric bounds, accepted by name, with re-verification triggers tied to the specific specification updates that would close them. The alternative, signing off AI-generated testbenches without these numbers, is not safer for having a shorter register; it is more dangerous, because the risk is unmeasured rather than absent.

7.10 Conclusion

Ten metrics, three families. RES, VG, SCR, and HR on the generation side; PMC, CVL, DFS, and OBS on the specification side; AUI and RGS on the process side. One claim tying them together, that the strongest return in AI-assisted verification is a more rigorous specification rather than a more capable model, and a structured-specification practice, threaded through the earlier chapters, that turns the claim into artifacts the framework already produces.

Two closing observations.

The metrics only help if you compute them. A program that adopts AI-assisted testbench generation without them has taken on a verification risk it cannot describe, and the risk is not hypothetical: in the controlled studies behind this chapter's taxonomy and metrics, the AI produced testbenches that compiled, simulated, and printed PASSED on fault scenarios where an entire RTL path went permanently unverified. No metric a conventional program already computes catches that class of failure. The metrics here do, and they are the only thing that does.

The metrics are diagnostic, not prescriptive. They find gaps; they do not close them. In every case this chapter covered, the fix is upstream of the model, better-structured specifications, explicit contract encoding, enumerated coverage obligations, named boundary values. The XTP, the manifest, and the SIG of the earlier chapters are the structured artifacts that close these gaps for the protocol problems the framework targets; other problem classes, descriptor-based devices, coherent fabrics, security-aware integration, will need their own structured artifacts, but the pattern holds. The artifacts are the lever; the model is the multiplier; with no lever, the multiplier acts on nothing.

The discipline is shifting because its tools are shifting. The Verification Strategist's part in that shift is not to surrender judgment to automation but to steer automation through structured artifacts and to grade its output against metrics that tell a working testbench from one that merely compiles. With those numbers in hand, the program is

ready to make its case. Chapter [8](#) takes that up, how coverage evidence becomes a defensible decision to tape out.

8. Verification Closure: From Coverage Evidence to Sign-Off Argument

Vikash Kumar¹ 

(¹) Leander, TX, USA

 **Vikash Kumar**

Email: vikash.singh261@gmail.com

Most programs do not sign off; they hit a deadline. Coverage rounds up, exclusions go unreviewed, waivers linger, and the schedule says ship. This chapter replaces that with a sign-off argument: an artifact that makes an explicit, bounded claim about what a design does and an equally explicit statement of what falls outside it. Done is reframed as a claim, verified against named specification versions, to the level the test plan asserts, under the conditions the testbench modeled, with everything else in a . Coverage becomes evidence only when each goal is derived from a plan row at plan time, which an AI does quickly because the work is translation, not invention. Unclosed coverage is then classified from the simulation log into three kinds, test-plan gap, architectural exclusion, and infrastructure gap, so closure comes from reading evidence rather than burning random seeds. The sign-off argument has five required components: XTP completeness, coverage closure, a zero bug-find-rate conclusion over a pre-defined window, the residual risk register, and explicit re-verification conditions. When testbench artifacts are AI-generated, their quality-metric values are

recorded in the register by name. The result is a defensible, reviewable position that the next program inherits as its baseline.

8.1 The Sign-Off Problem

Every engineer who has run a verification program has faced the same moment. The regression is green. The coverage numbers look good. The schedule says sign-off is tomorrow. And someone in the review asks: *are we actually done?*

Nobody has a good answer. Not because the team did poor work. Because the program never defined what done means in terms anyone can defend.

Sign-off in most programs is a negotiation, not a conclusion. Functional coverage reached 95%, so management rounds up to complete. The remaining 5% is documented as “known exclusions” that receive no scrutiny. Regression is clean except for three waivers that have been present for two months. The design team characterizes the remaining issues as low risk. The schedule says ship.

This is not a sign-off. It is a deadline. The difference matters when the silicon comes back.

The problem is structural. Verification programs accumulate evidence, passing tests, coverage numbers, bug counts, without ever assembling that evidence into a coherent argument. An argument requires a claim, evidence for the claim, and explicit acknowledgment of what the claim does not cover. A checklist of metrics is not an argument.

This chapter gives sign-off a structure: the **sign-off argument**. Not a stricter checklist, but a different kind of artifact. One that makes the claim explicit, ties the evidence to the claim, and documents residual risk honestly rather than hiding it in exclusion files and open waivers.

8.2 What Done Actually Means

Done is not a state. It is a claim. The claim has a specific structure:

This design, as specified in the documents listed below, has been verified to the level the XTP asserts, under the stimulus and environmental conditions the testbench modeled. Behaviors outside that scope are explicitly listed in the residual risk register with documented acceptance.

Every phrase in that claim is load-bearing.

As specified in the documents listed below. Verification is always against a specification. If the specification was wrong, the verification was correct against the wrong thing. The sign-off argument must name the specification versions it was conducted against. A design revision or a specification clarification that changes an expected behavior invalidates the claim for the affected behaviors and triggers re-verification.

To the level the XTP asserts. The XTP is the claim about what was verified. If the XTP had gaps, behaviors not planned for, interactions not anticipated, coverage that closed against it proves nothing about those gaps. Sign-off is bounded by the XTP. No more, no less.

Under the stimulus and environmental conditions the testbench modeled. Every testbench makes modeling assumptions. The AHB master was modeled as a single master; multi-master arbitration was a subsystem concern. APB slave responses were modeled as correct; adversarial slave behavior was not in scope. These are not failures of the verification program; they are scope decisions. They must be explicit.

Explicitly listed in the residual risk register with documented acceptance. Silence is not acceptance. Every behavior outside XTP scope, every architectural exclusion, every infrastructure limitation must be named, characterized, and accepted by someone with authority to accept it. An exclusion file that grows without review is a liability, not a methodology artifact.

This framing changes how sign-off is conducted. The question shifts from “did we hit the metrics?” to “is the claim we are making

defensible?” Those are different questions with different answers and different quality.

8.3 Coverage as XTP Evidence

Coverage numbers mean nothing without a reference. A functional coverage closure of 95% against a covergroup not derived from the XTP proves the testbench executed. It does not prove the design was verified.

The VEGA practice: every XTP row carries a coverage goal written at the time the row is written. The goal specifies precisely what observable signal or transaction a covergroup must capture for this row to be considered exercised. Coverage is derived from intent, not retrofitted after simulation.

8.3.1 Deriving Coverage Goals from XTP Rows

An XTP row for the bridge, with a coverage goal written at plan time:

XTP Row with Coverage Goal: Example

```
Row_ID:          T-007
Feature:          F-003 (Error Response Propagation)
Stimulus:         APB peripheral asserts PSLVERR during read
                  transfer
Expected:         Bridge asserts HRESP=ERROR per AHB spec
                  two-cycle
                  response: HREADY=0+HRESP=ERROR, then HREADY=
                  1+HRESP=ERROR; proceeds normally
Priority:          P0
Coverage_Goal:
  covergroup:      err_propagation_cg
  coverpoints / bins:
    - pslverr_during_read:  PSLVERR==1 && PENABLE==1 &&
      PWRITE==0
    - hresp_error_cycle1:  HRESP==ERROR && HREADY==0
    - hresp_error_cycle2:  HRESP==ERROR && HREADY==1
    - recovery_normal:      next_HTRANS==NONSEQ && HRESP==OKAY
  cross:           pslverr_during_read X hresp_error_cycle1
```

This coverage goal has a direct and specific meaning: the scenario exercising T-007 must produce every one of these observable conditions. When all bins close, T-007 is verified. When any bin remains open, T-007 is unverified regardless of what other coverage numbers show.

The practical consequence: when coverage is not closed, the question is not “which bin is open?” It is “which XTP row is unverified?” These produce different conversations. A bin number is a measurement. An unverified XTP row is a risk to chip functionality.

8.3.2 Coverage Goals Written at Plan Time, Not Regression Time

The sequence matters. Coverage goals written after regression tend to describe what the scenarios happened to produce, not what the XTP requires. An engineer writing covergroups from simulation output will naturally write bins for signals already visible, and omit bins for corner cases the scenario never reached.

Coverage goals written at plan time are derived from the specification. They capture what the design should do under conditions the engineer has not yet simulated. This is harder. It is also the only way to ensure that coverage closure means something.

Output contract: coverage goal from an XTP row

You provide: the XTP rows, the monitor set (from the Chapter 3 manifest), and the relevant protocol-spec sections.

Per row, it must return: the row ID; the features covered; a `covergroup` name; its `coverpoints` and the `bins` within each, written as the signal sampling condition that closes the bin; whether a cross is required and over which `coverpoints`; and the monitor or checker that makes it observable.

Bounds: write goals from the specification, not from simulation output; bins describe what the DUT *produces*, never what is driven onto it; a cross is required for any row testing an interaction of two or more features; express the goal as `covergroup/coverpoint/bins` so it drops into the collector unchanged; flag any row whose expected behaviour is not observable from the current monitor set as `INFRA_GAP`.

8.4 The Bridge Coverage Architecture

The coverage collector for the bridge verification program is a single file:

```
ch03_testbench/ahb2apb_bridge_uvmtb/env/ahb2apb_
bridge_coverage.sv
```

It implements six covergroups across 128 bins. The collector lives in the infrastructure layer instantiated from the environment. It is not in sequences, not in tests, and not in monitors. It observes transactions reported by monitors via analysis ports independently of which test is running, so coverage closes across the regression as a whole, not within any individual test.

The final coverage report is at `ch08_coverage_signoff/coverage_report.txt`. After running the full regression from `ch05_debug_regression/`, check overall coverage status:

```
cat ch08_coverage_signoff/coverage_report.txt
# Expected:
#   AHB_TXN           : 59/59   (100.0%)
#   APB_TXN           : 11/11   (100.0%)
#   REGISTER_ACCESS: 12/12   (100.0%)
#   ERROR_SCENARIOS: 11/11   (100.0%)
#   AHB_PROTOCOL      : 13/13   (100.0%)
#   APB_PROTOCOL      : 22/22   (100.0%)
#   OVERALL           : 128/128 (100.0%)
```

Coverage accumulates across tests: bins closed by the directed `coverage_closure_001` test remain closed in the regression union regardless of what other tests contribute. The 100% closure number in the coverage report reflects the regression union, not any single test.

8.5 The Three Kinds of Unclosed Coverage

Not all unclosed coverage requires the same response. The dominant practice treats every unclosed bin identically: run more stimulus. This is the wrong action in two of the three cases, and it is why regression plateaus are so common.

Before any coverage gap is addressed, it must be classified.

Figure 8.1 shows the triage: an AI classifies each unclosed bin from the simulation log and the RTL into one of three kinds, and each kind demands a different response and a different owner.

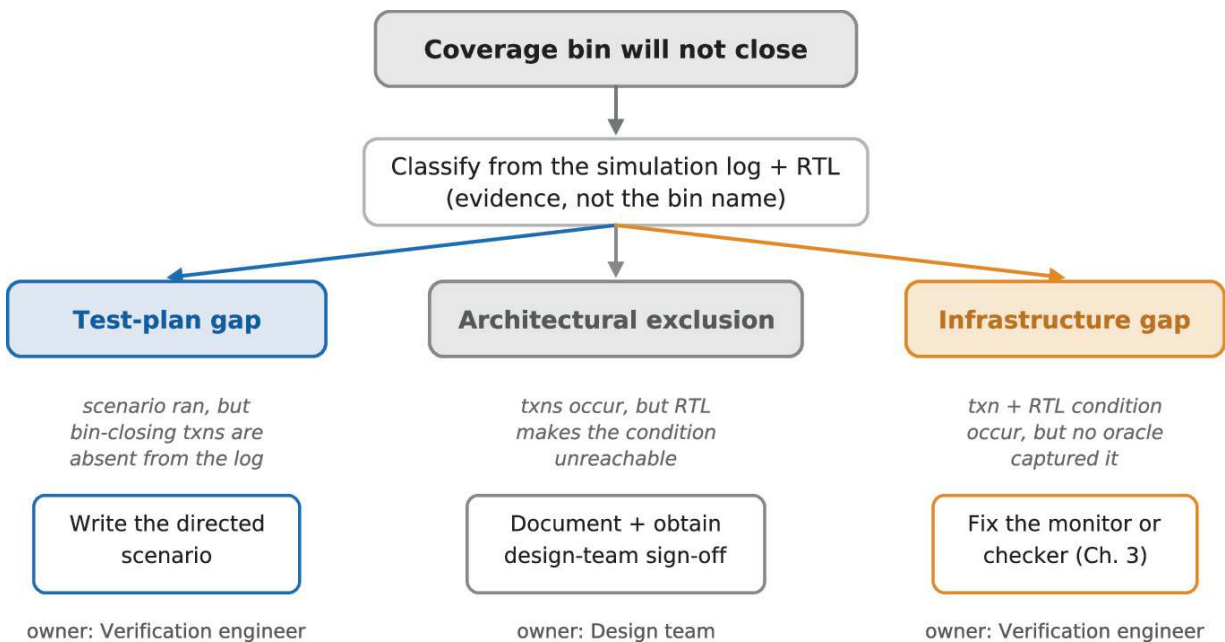


Fig. 8.1 The closure triage. When a coverage bin will not close, it is classified from the simulation log and RTL evidence (not the bin name) into one of three kinds: a test-plan gap, an architectural exclusion, or an infrastructure gap. Each demands a different response and a different owner. Misclassifying sends the wrong action: more simulation against an exclusion, or a waiver for a behavior that simply has not been tested

Table 8.1 names the three kinds and the response each demands.

Table 8.1 Classification of unclosed coverage and required response. Misclassifying a gap leads to the wrong action: running more simulation against an architectural exclusion, or filing an exclusion for a behavior that simply has not been tested

Two diagnostic questions narrow the classification quickly.

Question 1: Does the scenario produce relevant transactions?

Examine the monitor log when the scenario runs. If transactions that should close the bin are absent, the scenario does not exercise the behavior it was written for: a test plan gap. If transactions are present but the bin remains open, it is an infrastructure gap: the monitor saw the transaction but the covergroup did not sample the bin-closing condition.

Question 2: Can the RTL produce this behavior? Enable full branch and FSM coverage and run the scenario. If the branch or state transition that would close the bin is never taken despite valid input, examine the RTL. A constraint that makes the branch unreachable under all legal inputs is an architectural exclusion candidate, but it requires design team confirmation that the constraint is intentional and not accidental dead code.

Output contract: unclosed coverage bin classification

You provide: the unclosed bins with their sampling conditions, the relevant monitor-log excerpts, the relevant RTL (FSM or datapath), and the XTP rows.

Per bin, it must return: the bin ID and XTP row ID; the classification, one of `test_plan_gap`, `arch_exclusion`, `infra_gap`, or `NEEDS_ANALYSIS`; the evidence used; the required action; and the owner (verification engineer / design team / architect).

Bounds: classify from the log and the RTL, never the bin name alone. *Test-plan gap*: the scenario ran but bin-closing transactions are absent from the monitor log. *Architectural exclusion*: transactions occurred but RTL analysis confirms the bin-closing condition is unreachable under any legal input sequence (RTL analysis, not assumption). *Infrastructure gap*: the transaction and the RTL condition both occurred but the collector did not capture it (confirm the log shows the transaction first). If the evidence is insufficient, output `NEEDS_ANALYSIS` naming what is required.

8.5.1 Worked Example: Defining and Closing Bridge Coverage with AI

Defining the bridge coverage was fast because it was translation, not invention. The six `covergroups` and 128 bins were derived from the XTP feature rows: each row's `Coverage_Goal` already names the observable conditions, and those conditions become the bins. The

coverage-goal output contract shown earlier in this chapter is the whole job, run once per row.

Closing it is where programs lose weeks, and where classifying before acting pays off. The bridge regression first measured 72.7% (93 of 128 bins) union closure. Each open bin was classified from the simulation log and the RTL, not from its name, and all three kinds appeared:

- **Infrastructure gap.** Every burst and size bin was open, yet the datapath scenarios plainly drove INCR4 and halfword transfers. The first diagnostic question settled it: the transactions were present in the monitor log but the bins would not close, the signature of an infrastructure gap. The AHB monitor was calling `post_randomize()` after sampling the bus, and that routine reset `HBURST` and `HSIZE` to `SINGLE/WORD`, overwriting what the monitor had just captured. A monitor must report what it observed; deleting the call recovered the bins and lifted closure to 76.6%, with no new stimulus.
- **Test-plan gap.** The bins still open were burst encodings and error-address ranges no scenario had ever driven. The response is a directed scenario, not more randomization: one closure test drives every remaining `HBURST` encoding as a write and a read, a byte-size transfer, and the missing error-address and read-direction cases. It took the suite to 128/128.
- **Architectural exclusion.** The 13 bins of Table [8.2](#) were never part of the 128. They are unreachable in this single-master, single-transfer configuration, so they are documented, not chased.

The speed-up from AI is in the loop, not a single prompt. It reads the `UNCOVERED BINS` report and the monitor log, returns a classification per bin under the contract above, and, for each test-plan gap, drafts the directed scenario straight from the bin's own sampling condition; the engineer reviews the verdict and the diff. The entire move from 72.7% to full closure was two reviewed changes, one infrastructure fix and one directed test, found by reading evidence instead of burning seeds.

8.6 Structural Coverage and What It Proves

Functional coverage proves intent was exercised: the behaviors planned in the XTP were stimulated and the design responses were observed. Structural coverage proves the testbench reached the RTL: lines, branches, FSM states, and signal transitions were exercised.

Both are necessary. Neither is sufficient alone.

What functional coverage cannot see. A functional covergroup measures only behaviors it was written to measure. If the RTL contains a behavior the XTP did not anticipate, such as a state machine transition not in the test plan, or a combinational path that activates only under a signal combination not modeled in any scenario, functional coverage will not detect the gap. The bin was never written.

What structural coverage reveals. RTL branches, FSM states, and signal transitions that no scenario exercises. Two explanations exist for each uncovered structural element.

The first is an **XTP gap**: behavior the design can produce that the test plan did not include. A never-taken branch in the bridge FSM represents either a behavior the XTP missed or dead code. The verification engineer determines which. An XTP gap requires a new XTP row and a new scenario. Dead code requires design team review; unreachable RTL may indicate a specification misunderstanding or an implementation error.

The second is **confirmation of an architectural exclusion**. Structural analysis can confirm that a branch is unreachable: the upstream constraint makes it so, the RTL analysis confirms it, and the design team documents the reason. This converts a suspected exclusion into a verified one that can be entered in the residual risk register with confidence.

8.6.1 Structural Coverage Exclusion Governance

Every entry in the structural coverage exclusion file represents either dead code or a legitimate architectural constraint. Both require review. Exclusion files that grow without governance are among the most

common sources of undetected verification gaps in production programs, not because the exclusions are wrong, but because nobody checked.

The governance rule: every exclusion entry requires a one-line rationale and a reviewer name. The rationale explains why the branch or state is unreachable or why it represents intentionally dead code. At sign-off, the exclusion file is part of the verification record. A reviewer who cannot read the file and understand each entry cannot sign off on the structural coverage claim.

Table [8.2](#) lists the 13 bins excluded from the bridge coverage closure with their architectural rationale.

Table 8.2 Architectural coverage exclusions for the bridge verification program. Each entry is confirmed by the design team. All 13 bins are excluded due to upstream constraints that make the bin-closing condition unreachable under legal input sequences

8.7 The Regression Plateau and Directed Closure

Every simulation regression reaches a point where additional random seeds produce no new failures and no new coverage closure. The curve flattens. The number of failing tests per hundred new seeds approaches zero. The number of newly closed bins per regression run approaches zero.

This is the regression plateau. It is not sign-off. It is the signal that random stimulus has been exhausted and directed work must begin.

The plateau is diagnostic information. It communicates: the current scenario suite, driven by random constraints, has stimulated everything it can stimulate. Remaining unclosed bins require stimulus that random generation will not produce without targeted constraints.

Figure [8.2](#) traces the shape: random stimulus drives the failure and bin-closure rate down to the plateau, directed closure then targets the remaining bins, and a zero bug-find-rate held across a pre-defined window becomes part of the sign-off argument.

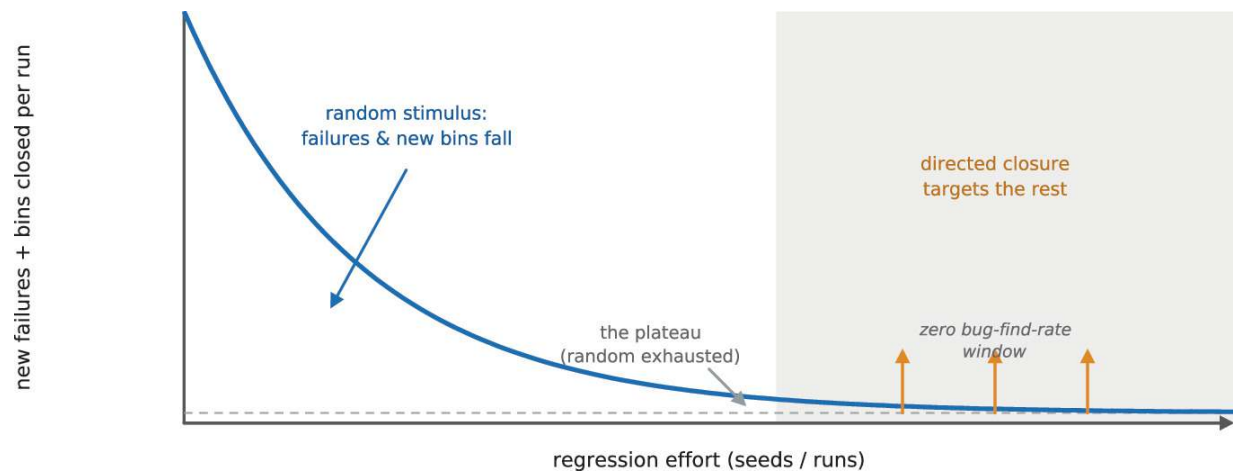


Fig. 8.2 The regression plateau. Random stimulus drives the failure and new-bin-closure rate down until the curve flattens, the plateau, which is the signal to begin directed work, not a sign-off. Directed closure then targets the remaining bins, and a zero bug-find-rate held across a window defined before regression began becomes a component of the sign-off argument.

8.7.1 Moving from Plateau to Directed Closure

Two activities at the plateau, in order.

Step 1: Classify all unclosed bins using the taxonomy from Sect. 8.5. Bins that are not yet closed appear in each test’s simulation log. To see which bins remain open after the full regression:

```
grep -A 50 "UNCOVERED BINS" \
    ch05_debug_regression/logs/test_coverage_closure_001.log
```

Test plan gaps require directed scenarios. Architectural exclusions require documentation. Infrastructure gaps require testbench work. Do not write directed scenarios for exclusions. Do not file exclusions for test plan gaps. The classification determines the action.

Step 2: Write directed scenarios for each test plan gap. A directed scenario targeting a specific unclosed bin is not a general stress test; it constructs the exact preconditions and stimulus that produce the bin-closing signal condition.

For the bridge: if the “HRESP=ERROR recovery” bin is not closed by random regression, the directed scenario builds the precise sequence: APB peripheral asserts PSLVERR during an active read, bridge asserts HRESP=ERROR with HREADY deasserted per AHB spec (two-cycle

ERROR response), subsequent HTRANS=NONSEQ accepted normally. The scenario has one success criterion: all four required bins in `err_propagation_cg` close on this run.

8.7.2 Bug Find Rate as a Sign-Off Leading Indicator

Coverage numbers are lagging indicators: they reflect what has already been exercised. Bug find rate is a leading indicator: it signals whether continued simulation is likely to find new issues.

Track bugs found per unit of regression effort from the beginning of regression. When the rate drops to zero and holds there across a defined observation window, the scenario suite has found what it can find in the current RTL.

Zero bug find rate is not proof that no bugs remain. It is evidence that the current scenarios are not finding them. The distinction matters: if the scenario suite does not cover the XTP, zero bug find rate is uninformative. If the scenario suite covers the XTP and structural coverage is clean, zero bug find rate across a meaningful observation window is a significant component of the sign-off argument.

Two rules keep the bug find rate conclusion honest.

The observation window is defined before regression begins, not chosen retrospectively to produce a convenient conclusion. A reasonable default: the window equals the longest interval between any two consecutive bugs found during regression, with a minimum of two weeks regardless of the inter-bug interval.

The scenario suite coverage is recorded when the zero rate is declared. If new scenarios are added after declaring zero rate, the clock resets. Adding scenarios changes what the regression is measuring, and the new scenarios must be given time to produce results before the rate can be declared zero again.

8.8 The Sign-Off Argument: Structure and Required Components

The sign-off argument is a document. It has five required components. All five must be present for sign-off to be complete. A sign-off argument missing any component is an incomplete argument with a date attached. Figure 8.3 lays out the five.

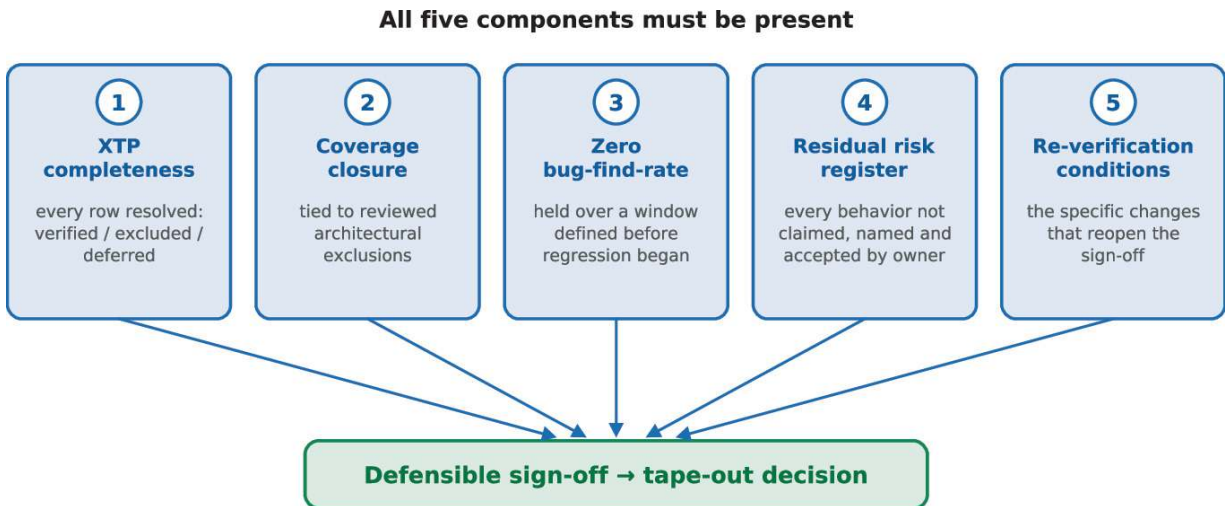


Fig. 8.3 The five components of a complete sign-off argument. Each addresses a distinct aspect of the verification claim; missing any one leaves a class of risk undocumented. Component 1 declares every XTP row resolved; Component 2 ties coverage closure to reviewed exclusions; Component 3 records a zero bug find rate over a pre-defined window; Component 4 is the residual risk register of what is not claimed; Component 5 names the specific changes that reopen the sign-off

8.8.1 XTP Row States at Sign-Off

Three states are valid. No other state is acceptable.

Verified. A passing scenario or formal property proof exists. The coverage goal defined at plan time is fully met. For simulation: the scenario is in the regression suite and has passed since the last RTL change affecting this behavior. For formal: the property proof reference and tool run are recorded; the proof is re-run if the RTL changes in ways that affect the property scope.

Excluded-with-justification. The behavior is in scope per the specification but is architecturally unreachable in the current RTL. The design team has confirmed the exclusion is intentional, not accidental dead code. A rationale is documented. A reviewer other than the proposing engineer has approved it.

Deferred-with-acceptance. The behavior is explicitly out of scope for this program. The decision to defer was made at plan time or during execution with documented rationale. The program's verification lead has accepted the deferral in writing. The behavior is in the residual risk register.

An XTP row with no state is a gap. Sign-off cannot proceed until every row is resolved into one of the three valid states.

8.8.2 The Residual Risk Register

The residual risk register is the component that makes the sign-off argument honest. It is where the verification program explicitly states what it did not verify and why that is acceptable.

Four categories of entries belong in the register.

Architectural exclusions. Behaviors planned in the XTP but unreachable in the current design. Entry: the XTP row, the architectural constraint, the design team sign-off.

Modeling assumptions. Behaviors the testbench modeled as correct that the real system might not guarantee. For the bridge: the APB slave model always responded within a bounded number of cycles; behavior under longer response times was addressed at subsystem level in Chap. [6](#) (EDGE_002). Entry: the assumption, what happens if violated, and whether it is verified at another level of the hierarchy. When the testbench artifacts were themselves AI-generated, this category also records their Chap. [7](#) quality-metric values, the VG, OBS, PMC, CVL distribution, DFS sub-scores, and the autonomy and regeneration-stability figures, so the sign-off states quantitatively how far the generated testbench was trusted.

Deferred behaviors. XTP rows out of scope. Entry: the row, the deferral rationale, the program or milestone at which the behavior will be addressed.

Infrastructure limitations. Behaviors the testbench cannot observe. Entry: the behavior, why the monitor cannot observe it, and whether a fix is planned. An infrastructure limitation accepted without

a fix plan means the behavior is permanently unverifiable. That must be explicitly accepted, not silently tolerated.

Output contract: residual risk register

You provide: the XTP with its row states, the structural exclusion file, the testbench manifest (Chapter 3), and the known modeling assumptions.

Per entry, it must return: an entry ID; the category, one of `arch_exclusion`, `modeling_assumption`, `deferred`, or `infra_limitation`; what is not verified; the rationale; the risk acceptor (name and role); and the re-verification trigger.

Bounds: populate all four categories; include only what is *not* claimed verified, never a verified behaviour; flag any entry with no acceptor as `NEEDS_OWNER`; flag any modeling assumption not verified at some other level of the hierarchy as `UNVERIFIED_ASSUMPTION` (escalate before sign-off).

8.9 Sign-Off as a Forward-Looking Artifact

Sign-off is not the end of verification. It is a documented position at a point in time.

The position: at this revision of this design, against these specification versions, with this scenario suite and testbench, the XTP claims are met, the residual risks are accepted, and the conditions requiring re-verification are explicit.

That position has value beyond the program that produced it. The next program to touch this design, a revision, a re-integration, or a derivative chip, starts from this sign-off as its baseline. Without a complete sign-off argument, the next program starts from uncertainty: some tests exist, some coverage was closed, some exclusions are in a file somewhere, but nobody can say what was and was not verified.

With a complete sign-off argument, the next program knows exactly. For the bridge program documented across Chaps. 2–6, the sign-off position is:

- **96 tests pass**, recorded in `ch08_coverage_signoff/test_results_summary.txt`
- **128/128 coverage bins closed**, recorded in `ch08_coverage_signoff/coverage_report.txt`

- **96 XTP rows resolved:** all 96 verified by a dedicated test class in the sign-off regression to subsystem level with documented acceptance
- **13 coverage bins excluded** with architectural rationale per Table [8.2](#)

If an RTL change touches the address decoder, re-verification of address-range coverage bins is required per Component 5 of the sign-off. If the change does not affect the address decoder, the existing sign-off continues to hold for those behaviors. This is a specific, reviewable statement, not “any RTL change triggers re-verification.”

This is the compounding value of structured verification. Each program following the VEGA framework produces an artifact that makes the next program faster and more confident. The sign-off argument from this chapter is that artifact.

8.10 Conclusion

Verification closure is not a metric threshold crossed on a schedule. It is a structured argument that makes an explicit, bounded, defensible claim about what a design does, and an equally explicit statement of what falls outside that claim.

The five components of the sign-off argument together constitute a complete answer to the question every review asks: are we actually done? Not done in the sense of the schedule says so. Done in the sense of: here is the claim, here is the evidence, here is what the claim does not cover, and here is who accepted the risk of what remains.

That is a defensible position. It is what verification is supposed to produce.

With closure complete, the framework has carried the verification program from its first specification reading to its final documented sign-off position. Chapter [7](#) established how to measure the quality of AI-assisted verification itself, distinguishing a generated testbench that compiles from one that verifies; this chapter has shown how that measurement, together with coverage evidence and an honest residual risk register, becomes a defensible decision to tape out. The Afterword

that follows reflects on where the Verification Strategist's role goes from here.

What the Framework Produced

Seven chapters. One complete verification program. It is worth naming what was built, because the artifacts are not obvious from a distance.

Chapter [2](#) produced an XTP: not a test list, but a structured claim about what the design should do, written before a single line of testbench existed, reviewed by the architect who knows whether the claim is right. That document governed everything that followed.

Chapter [3](#) produced a manifest and a testbench architecture that separated infrastructure from test logic by construction. The separation was not a style preference. It was the structural condition that made Chapter [5](#)'s debug discipline possible, because when infrastructure is correct by construction, a failing test has a bounded set of causes.

Chapter [4](#) produced scenarios that are narrow by design, traceable to XTP rows, and by contract prohibited from modifying the infrastructure they run on. Not because engineers were told to be disciplined, but because the scenario contract made undisciplined scenarios structurally invalid. Chapter [4](#) also produced the first live evidence of what AI-assisted generation looks like in practice: a complete test generated from an XTP row, compiled, debugged to a passing verdict, and an offer from the tool to scale to all 36 rows, with one stated constraint.

Chapter [5](#) produced a debug process grounded in classification before investigation. It also produced something less visible: an XTP that is better at the end of regression than it was at the beginning, because every bug found was treated as evidence about the plan, not just about the design.

Chapter [6](#) produced a SIG and an assumption harvest, artifacts that made integration-space defects visible before integration began. The bugs caught by those artifacts were not new bugs. They existed the moment two components with misaligned assumptions were placed in the same system. The only question was when they were found.

Chapter [7](#) produced ten quality metrics across three families: RES, VG, SCR, and HR for the generation side; PMC, CVL, DFS, and OBS for the specification side; and AUI and RGS for the generation process itself. They made the quality of AI-assisted verification measurable rather than asserted, and they feed this chapter's residual risk register whenever a testbench artifact was AI-generated.

This chapter produced a sign-off argument: a document that a verification manager, a design director, and a project lead can read and understand in ten minutes, that names what was verified, what was not, and who accepted the difference.

These artifacts compound. The XTP from Chap. [2](#) is still present in the sign-off argument of this chapter. The manifest from Chap. [3](#) is still present in the subsystem manifest of Chap. [6](#). The classification taxonomy from Chap. [5](#) is still present in the edge-following debug process of Chap. [6](#). Nothing was discarded. Everything accumulated.

Appendix A

The Educational Bridge Model

This appendix is the canonical specification of the worked example used throughout the book. The chapters refer to it as the bridge example; this is the complete model behind those references, collected here for lookup so the chapters themselves stay focused on methodology rather than device detail.

Disclaimer and Intellectual Property Notice

Copyright. Copyright © 2026 Vikash Kumar. All rights reserved.

Disclaimer. The views expressed in this model are those of the author and do not represent the position of any employer, standards body, or organization. No endorsement, sponsorship, or affiliation with any employer or organization is implied. This work was developed by the author in personal capacity and outside any employer-funded project. References to industry protocols (such as AHB and APB) are based on publicly available information only and are intended for educational purposes.

Third-Party Intellectual Property Notice. This model may reference third-party technologies, standards, trademarks, or intellectual property, including but not limited to Arm[®] AMBA[®] protocols (e.g., AHB, APB). All such rights remain the property of their respective owners. No license, express or implied, is granted to any third-party intellectual property, including but not limited to that of Arm Limited or its affiliates. Use of any third-party technology or specification referenced herein may require a separate license from the applicable rights holder.

A.1 Introduction

The Educational Bridge Model is a simplified bus bridge used to exercise verification workflows. It converts transactions from a high-

performance source-side interface into accesses on a peripheral-side interface. Its purpose is educational: it provides a structured artifact for demonstrating AI-assisted verification planning, interface and transaction modeling, assertion and coverage development, debug and trace analysis, and early-career learning.

This is not a formal protocol standard and does not reproduce any official specification. It is an independently written educational model; references to widely used bus terminology are included only to help readers connect the example to familiar verification scenarios.

A.1.1 Scope

The model covers, at a conceptual level, a single source-side interface driving a single peripheral-side interface; 32-bit address and data examples; single transfers and simple multi-transfer sequences; basic status and control register examples; and representative reset, wait, and error scenarios. It does not attempt to define a complete or standards-accurate bus specification — in particular, it does not define all legal source-side transaction encodings, all corner-case burst behaviors, arbitration or multi-target decoding, clock-domain crossing, or production-grade security, parity, or ECC features.

A.1.2 Definitions

Term	Meaning in this model
Source bus	A high-throughput request interface used as model input
Peripheral bus	A simpler target interface used as model output
Bridge model	Educational logic translating source traffic to target accesses
Sequence	A related set of transfers processed in order
DUT	Device under test
FSM	Finite state machine

A.2 Functional Overview

A.2.1 Conceptual Block Diagram

A.2.2 Intended Use

The model supports verification test plan creation, assertion development, functional coverage definition, AI-assisted trace interpretation, and classroom or workshop exercises (Fig. [A.1](#)).

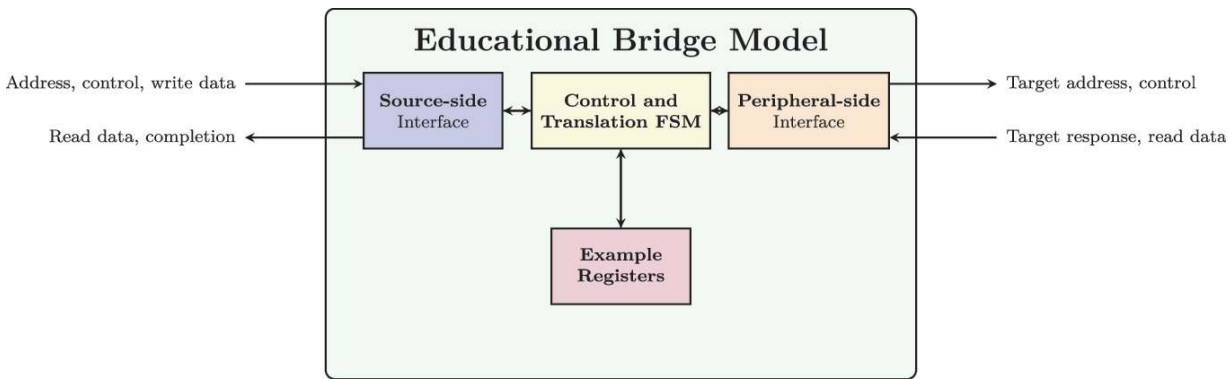


Fig. A.1 Conceptual organization of the educational bridge model

A.3 Interface View

A.3.1 Source-side Interface

The source-side interface uses familiar industry signal names to make the example easy to follow. These signals are a learning-oriented abstraction rather than a formal protocol definition (Table [A.1](#)).

Table A.1 Source-side example signals

Signal	Width	Dir	Educational description
HCLK	1	In	Clock used for the example timing model
HRESETn	1	In	Active-low reset used for initialization scenarios
HADDR	32	In	Source-side address example
HTRANS	2	In	Indicates transfer phase intent, such as new or continued activity
HWRITE	1	In	Direction indicator for read or write examples
HSIZE	3	In	Example transfer-size control
HBURST	3	In	Example sequence-pattern control
HWDATA	32	In	Source-side write payload
HSEL	1	In	Local selection control for the bridge model

Signal	Width	Dir	Educational description
HREADY_IN	1	In	Upstream completion context input
HRDATA	32	Out	Source-side read payload returned by the model
HREADY_OUT	1	Out	Model completion indicator / back-pressure output
HRESP	1	Out	Simplified status indicator for success or error

A.3.2 Peripheral-side Interface

See Table [A.2](#)

Table A.2 Peripheral-side example signals

Signal	Width	Dir	Educational description
PCLK	1	In	Target-side clock, commonly tied to HCLK in this example
PRESETn	1	In	Active-low reset for target-side model state
PADDR	32	Out	Target-side address example
PSEL	1	Out	Indicates active target selection in the example flow
PENABLE	1	Out	Marks the active transfer phase on the target side
PWRITE	1	Out	Target-side direction indicator
PWDATA	32	Out	Target-side write payload
PRDATA	32	In	Target-side read payload
PREADY	1	In	Target completion / stall indicator
PSLVERR	1	In	Simplified target-side error indicator

A.3.3 Clock and Reset Notes

The model assumes a single clock domain for simplicity. Reset behavior is intentionally simplified to support debug and verification exercises. Timing and reset behavior here should be treated as representative rather than normative.

A.4 Example Register View

The model includes a small set of example control and status registers to support observability and software-driven verification scenarios.

These are not intended to mirror any commercial register map (Table [A.3](#)).

Table A.3 Example register map

Offset	Name	Access	Purpose in the model
0xF00	CTRL	R/W	Enable and control knobs
0xF04	STATUS	R/W1C	Status and sticky error flags
0xF08	ERROR_ADDR	RO	Captured address for debug
0xF0C	ERROR_INFO	RO	Captured context for debug

A.4.1 CTRL Register

See Table [A.4](#).

Table A.4 Example CTRL register

Bits	Name	Access	Reset	Purpose
31:8	Reserved	RO	0	Reserved in this model
7	TIMEOUT_EN	R/W	0	Enables timeout behavior in verification runs
6:4	TIMEOUT_VAL	R/W	3'b111	Selects a timeout window for experiments
3	ERR_INT_EN	R/W	0	Enables sticky interrupt-style indication
2	Reserved	RO	0	Reserved
1	SOFT_RST	R/W	0	Triggers a model-local reset action
0	ENABLE	R/W	1	Enables bridge activity in the example flow

A.4.2 STATUS Register

See Table [A.5](#).

Table A.5 Example STATUS register

Bits	Name	Access	Reset	Purpose
31:8	Reserved	RO	0	Reserved in this model
7	ERR_INT	R/W1C	0	Sticky aggregated error indication
6	TIMEOUT_ERR	R/W1C	0	Timeout scenario observed

Bits	Name	Access	Reset	Purpose
5	PSLVERR	R/W1C	0	Target-side error observed
4	ADDR_ERR	R/W1C	0	Invalid-address scenario observed
3:2	Reserved	RO	0	Reserved
1	BUSY	RO	0	Model currently processing traffic
0	READY	RO	1	Model available for new work

A.4.3 Captured Debug Registers

See Table [A.6](#).

Table A.6 Captured debug state

Register	Purpose
ERROR_ADDR	Captures the address associated with the most recent modeled error event
ERROR_INFO	Captures context such as transfer direction, approximate beat index, and coarse error class

A.5 Behavioral Model

At a high level, the model accepts source-side transactions and turns them into target-side accesses. Each accepted source-side transfer is converted into one target-side operation. The bridge can delay source-side completion when the target side is not ready.

FLOW-1: Capture source-side address, control, and payload when the model accepts a transfer.

FLOW-2: Translate the transfer into a target-side setup phase.

FLOW-3: Advance to a target-side active phase.

FLOW-4: Hold progress if the target side indicates it is not ready.

FLOW-5: Complete the transaction and return any read data or error context.

A.5.1 Sequence Handling and Back-Pressure

The model supports simple ordered multi-transfer sequences: it tracks whether traffic is part of a continued stream, addresses may advance according to transfer size, and it may emulate wrapped or bounded sequences. Exact sequence encoding and all legal corner cases are outside the scope of this educational artifact. When the target side is not ready, the bridge keeps the target transfer active and delays source-side completion, letting verification engineers exercise stalled completion, timeouts, scoreboard synchronization, and latency-aware debug.

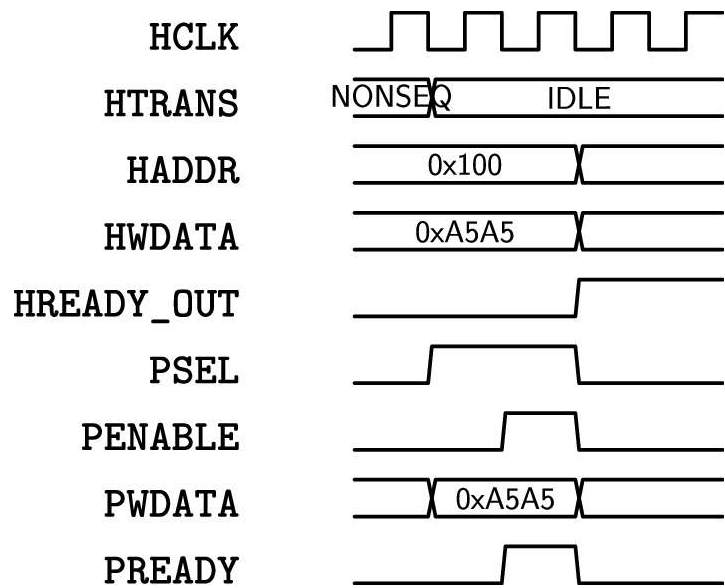


Fig. A.2 Representative transfer timing. Phases by cycle: source capture (T1), target setup (T2), target active with completion (T3)

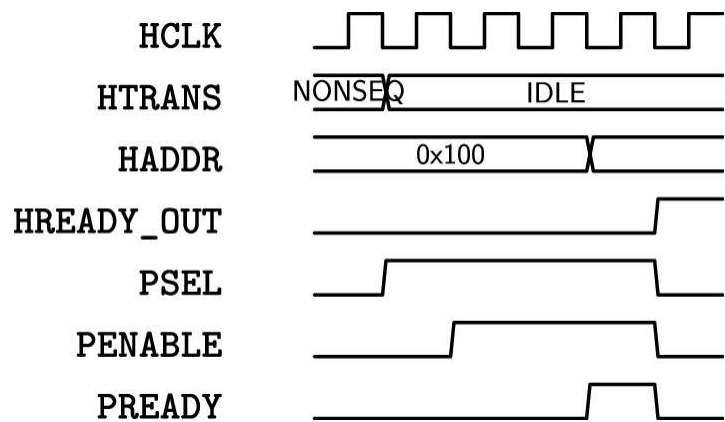


Fig. A.3 Representative delayed-completion timing. The target stalls (**PREADY** low through T3–T4), so the bridge holds the active phase and source-side completion is deferred until T5

A.5.2 Representative Transfer Timing

See Fig. [A.2](#).

A.5.3 Representative Wait-State Timing

See Fig. [A.3](#).

A.6 Reset and Initialization

Reset returns the bridge to a clean idle state, supporting scenarios such as reset during idle, reset during an active transfer, reset followed by immediate re-use, and reset interaction with sticky error state (Table [A.7](#)).

Table A.7 Example reset state

Signal/Register	Example reset value	Notes
HREADY_OUT	1	Ready for new activity
HRESP	0	No error indication
HRDATA	0	Cleared for simplicity
PSEL	0	No active target selection
PENABLE	0	No active target phase
PADDR	0	Cleared for simplicity
PWDATA	0	Cleared for simplicity
CTRL	0x00000001	Enabled, timeout disabled
STATUS	0x00000001	Ready, no sticky error set
ERROR_ADDR	0x00000000	Cleared
ERROR_INFO	0x00000000	Cleared

A model-local soft reset may be triggered through the example control register; it clears active transfer state, clears sticky status flags, and preserves basic configuration where appropriate.

A.7 Error and Exception Scenarios

The model includes a small number of coarse error classes (Table [A.8](#)).

Table A.8 Modeled error classes

Error class	Representative meaning in the model
Address error	Request targeted an unsupported or invalid range in the example map
Timeout error	Target-side readiness did not arrive within the programmed observation window
Target error	Peripheral-side error indication was observed during an active access

When an error is observed, the model records debug context, updates sticky status bits, returns a simplified source-side error indication, and becomes ready for subsequent debug or re-test activity.

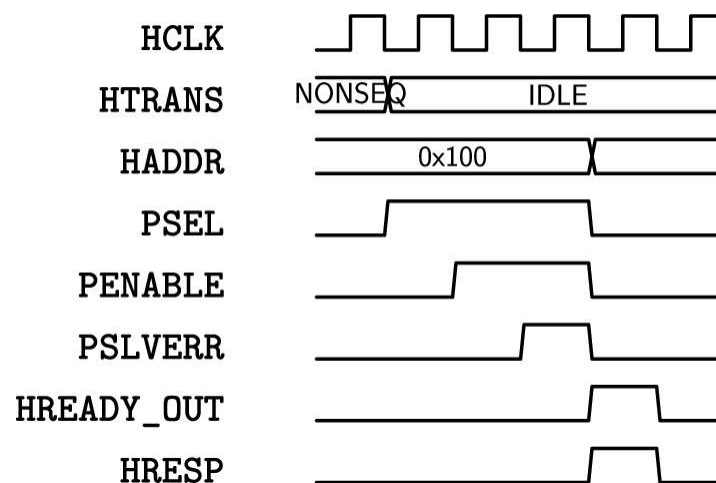


Fig. A.4 Representative modeled error timing. A target-side error (**PSLVERR**) observed during the active phase is reflected as a source-side error response (**HRESP** high) at completion

A.7.1 Representative Error Timing

See Fig. [A.4](#).

A.8 Verification Usage

The model exists to support verification framework development rather than to act as a shipping design document. It is useful for converting informal requirements into checkable behavior, deriving assertions from interface intent, generating coverage goals, creating AI-assisted debug prompts from observed traces, and demonstrating how an educational specification can drive verification assets (Table [A.9](#)).

Table A.9 Representative verification outputs

Artifact	Use in the VEGA/cognitive verification flow
Assertions	Check setup/active progression, completion gating, and error capture
Coverage items	Measure reset, wait-state, error, and sequence scenario exploration
Sequences	Generate single, delayed, and error-injected transactions
Debug prompts	Correlate model state with trace signatures and expected outcomes

Example Verification Questions

- Does source-side completion remain blocked while the target side stalls?
- Is read data returned only after target-side completion?
- Are sticky error flags captured and preserved until cleared?
- Does reset cleanly abort in-flight work?
- Are multi-transfer sequences handled in-order?

A.9 Example State Machine

This model is intentionally simplified to support teaching, verification planning, and AI-assisted methodology discussion. It should be read as an educational bridge model, not as a standards document or production design specification (Fig. [A.5](#)).

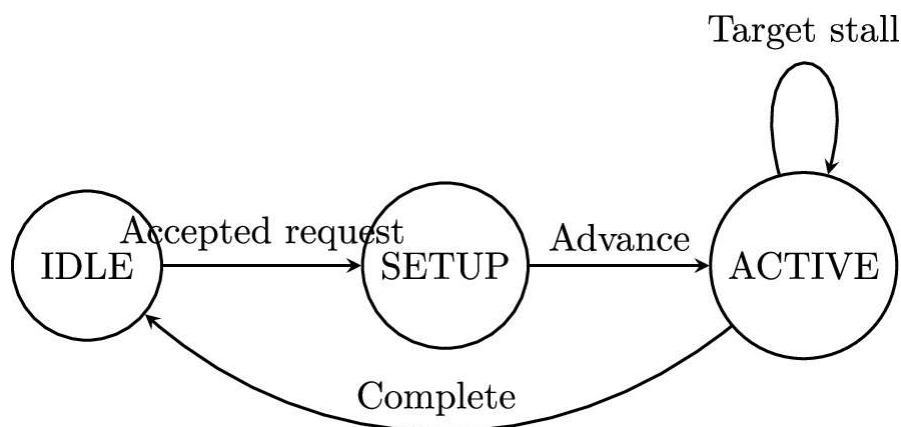


Fig. A.5 Simplified educational state machine

Afterword

The Verification Strategist Going Forward

The verification engineer's job is changing. Not because AI has arrived—it has been arriving for years in the form of constraint solvers, coverage closure tools, formal engines, and automated regression management. The change is one of degree, and the degree now matters. When AI generates a testbench scaffold that would have taken a week, the engineer's time is freed for the work the AI cannot do: deciding whether the scaffold is verifying the right things, whether the XTP reflects what the architect actually intended, whether the sign-off argument is honest or merely complete. That work is harder than writing the scaffold. It requires understanding the design, the protocol, the integration context, and the risk tolerance of the program. It requires judgment that is not in the prompt. The risk is not that AI replaces this judgment. It is that engineers stop exercising it because the tools produce plausible outputs with increasing speed. A testbench that compiles and simulates is not a testbench that verifies. A coverage number that is high is not a coverage number that means something. A sign-off that happened on schedule is not a sign-off that is defensible. The Verification Strategist role is the answer to this risk. It is the role that stays in the reasoning, even when the execution is automated. It is the role that reads the XTP row and asks whether the expected result is actually correct, not just whether the scenario closed the bin. It is the role that looks at the sign-off argument and asks whether the residual risk register is honest before signing the page. No framework automates that role. VEGA structures the work around it.

Responsible Adoption

Teams introducing VEGA should do so incrementally, validating each layer before automating it. Establish the XTP discipline first and run one complete program with it—writing coverage goals at plan time, maintaining row states, conducting architect review—before introducing AI-assisted generation. When AI arrives, the team already

knows what good output looks like and can evaluate it. The same principle applies at every layer: use the manifest and infrastructure boundary manually for one program before generating scaffolding from it; apply the scenario contract by hand before using AI to generate scenario skeletons. The manual application reveals what the framework requires; the automated application then accelerates delivery of that requirement. The failure mode to avoid is adopting the prompts without adopting the framework — a team that runs the Chapter [4](#) generation prompt without an XTP, without a manifest, without the infrastructure boundary enforced will get generated code, but whether that code verifies the design is a question the prompt cannot answer. The framework is the structure they accelerate. Order matters.

Where VEGA Extends Next

Two directions are worth naming because they extend directly from the framework's existing structure rather than requiring new foundations.

Post-silicon validation and bring-up. The sign-off argument identifies, explicitly, what was not verified in simulation. That residual risk register is the primary input to post-silicon validation planning. The behaviors that were architectural exclusions or deferred to system level in the sign-off argument are the first things to test when silicon arrives. The XTP, the SIG, and the residual risk register together constitute a bring-up guide that pre-silicon verification rarely produces systematically. VEGA programs that maintain their artifacts through sign-off arrive at silicon bring-up with a directed test plan rather than an improvised one. **Formal as a first-class VEGA input.** Chapter [8](#) introduced formal verification as a coverage complement — a tool for closing simulation gaps. The deeper integration is to treat formal properties as XTP rows from the beginning of the program. If a liveness property is important enough to include in the sign-off argument, it is important enough to be in the XTP at plan time with a coverage goal of “proved by formal.” This makes the formal scope a governed activity subject to the same Verification Strategist review as simulation scope, rather than an ad-hoc addition at the coverage closure stage. The XTP

becomes the single artifact that governs both simulation and formal scope, with the sign-off argument drawing from both. The verification discipline is not finished. Neither is the framework. What this book has established is the structure — the reasoning architecture that lets future extensions build on a principled base rather than on accumulated practice. That is enough to begin.

References

1. IEEE Standards Association (2023) IEEE standard for SystemVerilog—unified hardware design, specification, and verification language. IEEE Std 1800-2023
2. Accellera Systems Initiative (2014) Universal verification methodology (UVM) Class Reference. Version 1:2
3. Arm Limited (2021) AMBA AHB protocol specification. ARM IHI 0033C
4. Arm Limited (2010) AMBA APB protocol specification. ARM IHI 0024C
5. Bergeron J (2003) Writing testbenches: functional verification of HDL models, 2nd edn. Springer
[<https://doi.org/10.1007/978-1-4615-0302-6>]
6. Foster H, Krolnik A, Lacey D (2004) Assertion-based design, 2nd edn. Springer
7. Lee JD, See KA (2004) Trust in automation: designing for appropriate reliance. Hum Factors 46(1):50–80
[[MathSciNet](https://doi.org/10.1518/hfes.46.1.50.30392)][<https://doi.org/10.1518/hfes.46.1.50.30392>]
8. Amershi S (2019) Guidelines for human-AI interaction. Proceedings of the ACM CHI conference on human factors in computing systems
9. Doshi-Velez F, Kim B (2017) Towards a rigorous science of interpretable machine learning.arXiv:1702.08608

Index

A

AI-assisted test generation 132
AI-assisted verification 2, 4, 10, 106–108, 131, 147
Ambiguity resolution 13
Architectural exclusions 133, 135, 137–141, 143, 145
Assumption harvest 77, 81, 84–86, 93, 100, 101, 106, 147

B

Bug find rate 143, 144

C

CDC and reset sequencing 80
Claude Code 24–26, 28, 29, 34, 35, 37, 38, 40, 54, 72
Claude Code 38
Cognitive verification architecture 1, 4, 6, 59
Constrained-random verification 43
Coverage closure 5, 9, 14, 73, 75, 108, 133, 135, 136, 142, 144
Coverage planning 110
Cross-component contracts 80
Cross-feature interaction 13, 14, 17, 22

D

Debug methodology 5
Descriptor fidelity 107, 108, 120, 123
Design intent 5, 6, 10
Directed testing 9, 48, 49, 51, 68, 73, 98, 121, 130, 141

E

Executable TestPlan 13–15, 23, 24, 43

F

Failure classification 10, 63
Feature extraction 1, 13, 16, 22
Formal specification 4, 10, 109

Functional coverage 5, 49, 50, 75, 133, 135, 141

H

Hardware verification 1, 59

I

Infrastructure and test separation 23, 35

Infrastructure and test separation 25

Integration testing 104

Intent composition 10, 77

IP-XACT 23, 29, 30, 33, 35, 85, 105

L

Liveness and timeout debug 59

LLM code generation 9

LLM testbench generation 23, 26, 40, 112

M

Minimal reproduction 59, 68–70, 72, 73, 75

Minimal reproduction 68

P

Programming model coverage 107, 108, 121

Q

Quality metrics 7, 10, 107, 126, 147

R

Regression management 10

Regression plateau 137, 142

Regression suite 4, 45, 53, 77, 137, 144

Regression management 60, 62, 64, 66, 68, 70, 72, 74, 76

Requirements traceability 43

Residual risk register 10, 133, 135, 141, 144, 145, 147

Root-cause analysis 129

S

Scenario classes 43, 47–49, 56, 57, 77, 93, 95, 96, 102, 104, 106

Shift-left verification 9, 106
Simulator-agnostic 23
Specification coverage 107, 108, 115, 117, 120
Subsystem Intent Graph 7, 77, 79, 81, 84, 87
Subsystem verification 9, 10, 75, 77, 78, 81, 83, 87, 91, 101–106
Subsystem verification 77
SVA assertions 64
SystemVerilog 1, 2, 6, 25, 28, 29, 33, 59, 109, 118, 119

T

Tape-out decision 45
Test plan 3, 8, 10, 13–18, 20, 25, 44, 46, 53, 59, 60, 73–75, 77–80, 93, 107, 133, 137, 139, 141, 143
Test scenario design 10, 41, 43
Testbench architecture 10, 22, 147
Tool evaluation 129

U

Universal Verification Methodology (UVM) 1, 2, 6, 7, 9, 10, 23–26, 29, 30, 35–37, 40, 43, 45, 47, 51, 59, 63, 64, 68, 70, 72, 97, 99, 107–111, 117, 118
UVM log analysis 63
UVM sequences 97
UVM testbench 7, 24–26, 29, 108–110, 117

V

VEGA framework 1, 5–10, 53, 59, 76, 146
VEGA framework 6
Verification gap 1, 4, 5, 107, 108, 111, 114, 117, 126, 127, 141
Verification intent 9, 20, 31, 32, 37, 129
Verification sign-off 44
Verification strategist 1, 7, 9–11, 13–15, 17, 22, 46, 54, 57, 60, 62, 71, 74, 75, 79, 81, 85, 91, 100, 101, 104, 115, 132, 147
Verification gap 4
Verification strategist 9

X

XTP feedback 74, 101, 139

Table 8.2 Architectural coverage exclusions for the bridge verification program. Each entry is confirmed by the design team. All 13 bins are excluded due to upstream constraints that make the bin-closing condition unreachable under legal input sequences

Bin ID	Rationale	Owner
REG.ERROR_ADDR.WRITE	Read-only register; write path does not exist in RTL	Design team
REG.ERROR_INFO.WRITE	Read-only register; write path does not exist in RTL	Design team
APB_PROT.SETUP.WAIT	PREADY is undefined during SETUP phase per APB spec; bin condition is architecturally unreachable	Design team
AHB_PROT.HTRANS_BUSY	Requires pipelined burst driver; single-master model does not generate BUSY transfers	Verification lead
AHB_PROT.HTRANS_SEQ	Requires pipelined burst driver; single-master model does not generate SEQ in this configuration	Verification lead
AHB_PROT.NONSEQ.STALL	Requires back-to-back pipelined transfers; single-master model cannot produce this sequence	Verification lead
AHB_TXN.burst_x_resp.ERROR (7 bins)	Requires mid-burst error injection from APB slave; deferred to subsystem level	Verification lead

Table 6.1 SIG edge contracts for the AHB/APB subsystem. Rows marked HIGH represent known defect sites that must be addressed by subsystem scenarios.

From	To	Guarantee	Assumption	Risk
Bus Master	Interconnect	Valid HTRANS sequencing per AHB spec: NONSEQ for first beat, SEQ for continuation, IDLE when no transfer pending	HTRANS=IDLE before arbitration transfer completes	Medium
Interconnect	Bridge	Single master at a time; HTRANS=IDLE at ownership transition	Consistent HTRANS sequencing; no mid-burst ownership change	Medium
Bridge	Peripherals	Correct PADDR mapping; PSLVERR propagated as HRESP=ERROR	PREADY asserted within bounded cycles; PSLVERR on error only	HIGH
Peripherals	IntCtrl	Interrupt pulse meets minimum width per IntCtrl spec; de-asserted before re-assert	Interrupt arrives before polling timeout; no pulse overlap	HIGH
Reset Ctrl	All APB peripherals	PRESETn stable before HRESETn deasserts; HRESETn synchronous to HCLK	APB devices ready before AHB transactions arrive at bridge	HIGH
Reset Ctrl	Bridge	HRESETn synchronous to HCLK	No transaction in flight at HRESETn assertion	Medium
Bridge	All peripherals	Unmapped address returns HRESP=ERROR; HREADY asserted; no bus hang	All transaction addresses map to a valid peripheral or error response	HIGH

Table 7.1 The eight LLM testbench-generation failure modes, ordered by detection stage: Compile, Elaboration, or Simulation. That simulation dominates the stage column is the core argument against treating compile success as a quality measure for generated testbenches

#	Mode	Layer	Characteristic signature
L1	Virtual interface port misuse	E	The monitor reaches a signal straight through the virtual interface where a clocking-block member is required, or the reverse. The syntax compiles; elaboration is where the type mismatch in member resolution finally surfaces
L2	Hallucinated sequence item fields	C	The sequence item names fields the package never declared (<code>size_type</code> , <code>rw_type</code> against a class that actually declares <code>xfer_size</code> , <code>xfer_rw</code>). One root misunderstanding tends to spawn a cluster of downstream compile errors
L3	Role confusion	S	A driver takes the wrong side of the protocol. An APB slave driver ends up driving <code>PADDR</code> , <code>PSEL</code> , and <code>PENABLE</code> , the master's outputs, instead of confining itself to <code>PRDATA</code> , <code>PREADY</code> , and <code>PSLVERR</code> . Every signal holds a value so simulation proceeds, yet the slave never answers a real master
L4	Timing phase error	S	Pipeline timing is off by a phase. AHB <code>HWDATA</code> is presented in the same cycle as <code>HADDR</code> rather than one cycle later. Each transfer carries the wrong data, and the checks compare it against expectations that are wrong in the same way
L5	Response deadlock	S	A master sequence waits on <code>get_response()</code> for a <code>put_response()</code> the driver never issues. The run stalls at the very first transaction, with no <code>UVM_FATAL</code> and no error, only time passing
L6	Clocking block sampling skew	S	Input skew on the clocking block is set wrong, so the monitor samples one cycle away from the protocol's defined point. The resulting mismatches are intermittent and value-dependent; under random stimulus they masquerade as a probabilistic design bug, which they are not
L7	Analysis port disconnection	S	The monitor's analysis port has no subscriber, or the subscriber's <code>write()</code> is never called. The scoreboard reports no comparisons rather than no failures, and unless that absence is noticed the run looks like a pass
L8	Reactive-sequence finite loop	S	The reactive slave sequence iterates a fixed number of times instead of running <code>forever</code> . Once the count runs out the simulation hangs. Everything else about the sequence is correct; the choice of <code>repeat(N) over forever</code> is the whole bug

Step	Job	Output	Gate
1	Extract features, surface ambiguities	Feature list + ambiguity register	Architect sign-off on blockers
2	Review feature list against spec	PASS / FAIL per feature	Ready-for verdict
3	Generate test plan	Test table with traceability	Strategist approval
4	Generate and validate XTP	Signed-off XTP	All validation checks pass

Table 7.3 Three DMA fault scenarios and their contract violation latencies. F1 and F2 are slow but bounded; F3 is unbounded because the failure class has no monitor

ID	Scenario	CVL	Detected By
F1	Descriptor with invalid length	595 ns	Existing scoreboard
F2	Channel disabled mid-transfer	1735 ns	Existing scoreboard
F3	IRQ asserted, never cleared	∞	<i>No monitor exists</i>

Table 5.1 What is invariant across designs, and what each run changes

Invariant (every design, every run)	Design- or run-specific
The four failure classes	Which class a given failure belongs to
The four deviation types (count/value/ordering/liveness)	Which deviation a given failure produces
The order: classify, then log, then transaction level, then waveform	The actual logs, counts, and timestamps
Testbench failures tend to dominate early in a campaign	The exact testbench-to-design ratio, and the order they appear
Fix each defect at its owning layer, then re- baseline	Which layer each defect happens to live in

Table 4.2 XTP categories mapped to scenario classes

XTP Category	Class	Rationale
Bringup/Init	Functional	Basic bringup and reset-to-idle under normal conditions
Connectivity	Functional	Normal interface behavior under standard conditions
Register access	Functional	Read/write at defined addresses, field-level checks
Main datapath	Functional + Corner-case	Normal paths plus boundary burst lengths
Protocol	Corner-case	Exact APB timing requirements from spec
Reset	Corner-case	Specific timing conditions, exact cycle counts
Error handling	Error	Fault injection, error capture, recovery verification
Cross-feature	Error + Stress	Interaction under fault conditions and combined load

Table 6.4 Per-ECO validation effort: subsystem-scope fixes versus full-SoC fixes. Subsystem validation runs faster because the regression scope is bounded, the testbench setup is lighter, and the affected behavior is more directly observable.

Phase	Subsystem	Full SoC	SoC/Subsystem
Test rerun	0.5 days	2 days	4.0×
Test robustness	1 day	3 days	3.0×
Full regression	1 day	4 days	4.0×
Signoff time	2.5 days	9 days	3.6×

Table 5.3 What each failure type reveals about the XTP

Failure type	What it reveals	XTP update required
Design bug caught by existing test	XTP worked. Infrastructure caught the bug	Mark row verified. Record as coverage evidence
Design bug not covered by any XTP row	XTP has a gap. The behavior was not planned for	Add the row with architect review. Add regression test
Design bug where XTP expected result was wrong	XTP has an error. Specification was misunderstood during planning	Correct the row. File specification clarification
Infrastructure failure	Chap. 3 quality gate was insufficient or bypassed	Update the quality gate to catch this class of failure
Test logic failure	Scenario was not reviewed against its XTP row before regression	Apply the Chap. 4 scenario readiness review before re-running

Table 7.2 Per-dimension PMC for the DMA controller. Channel utilization, out-of-order ordering, and fault scenarios hit 100%, the dimensions the AI was built to drive; completion status reached only 20%, the decisive gap

Dimension	Coverage (%)	Notes
Channel utilization	100	Both channels exercised
Descriptor field combinations	60	Common combinations only
Completion status codes	20	Only <code>STAT_SUCCESS</code> and <code>ERR_NONE</code> observed
Error response codes	100	Fault injection paths
Out-of-order completion	100	UID tracker enabled this
Fault injection scenarios	100	Designed-in test class
IRQ delivery paths	0	No <code>IRQ_CLEAR</code> handshake

Table 7.4 DFS by descriptor field. Three independent capabilities: the LLM understood the hardware mechanism fully (generate), understood only part of what to validate back (check), and had no systematic sense of which values to stress (boundary).

Field	Generate	Check	Boundary	Notes
SRC_ADDR	✓	✓	–	Stimulus and check complete
DST_ADDR	✓	✓	–	Stimulus and check complete
LENGTH	✓	✓	✓	Boundary coverage only here
FLAGS	✓	–	–	IRQ_EN set; no checker
NEXT_DESC	✓	–	–	Chain semantics not checked
STATUS	✓	–	–	Field written; not validated
RESERVED	✓	–	–	Stimulus generated; no semantic check
Total	7/7	3/7	1/7	

File	Written by	Contains	Updates when
IP-XACT XML	Design team	Port names, widths, directions	DUT interface changes
manifest.json	Verification engineer	Clock, reset, interfaces, goals, tool paths	Hardware config changes
verification_intent.txt	Verification engineer	What to drive, observe, check	Strategy changes
Spec PDF	Architect team	Protocol timing, DUT behavior	Design spec revision

Table 4.3 Bridge XTP test counts by category
(Phase 1 input)

Category	Tests	Dominant scenario class
Bringup/Init	4	Functional
Connectivity	10	Functional
Register access	17	Functional
Main datapath	18	Functional + Corner-case
Protocol	17	Corner-case
Reset	8	Corner-case
Error handling	16	Error
Cross-feature	6	Error + Stress
Total	96	

Table 8.1 Classification of unclosed coverage and required response. Misclassifying a gap leads to the wrong action: running more simulation against an architectural exclusion, or filing an exclusion for a behavior that simply has not been tested

Classification	Definition	Required Response
Test plan gap	XTP row exists; coverage goal defined; no scenario exercises this behavior	Write the scenario. Not optional. An unexercised XTP row is an unverified claim. Running more random regression will not close a bin that requires specific preconditions the random generator does not produce.
Architectural exclusion	Behavior is specified but physically unreachable in the RTL as implemented due to upstream constraints confirmed by the design team	Document the exclusion with architectural rationale. Obtain design team sign-off. Mark the XTP row as excluded-with-justification. Silent exclusion is not acceptable and creates a liability at the next design revision.
Infrastructure gap	Behavior is in the XTP; a scenario exercises it; but the testbench monitor cannot observe whether it occurred correctly	This is a Chap. 3 feedback. Fix the monitor or checker. Do not waive an infrastructure gap; an unobservable XTP row is unverifiable, not merely unverified. These two conditions are not equivalent.

Tool	Job	When
vega_llm_tbgen.py	Generates TB scaffolding	Once per project
Claude Code	Fixes compile and sim errors	Iteratively
Git	Records every change	Automatically

Table 6.2 Mapping from intent category to assertion or coverage check. Each row is a protocol-agnostic template; the specific signal names and timing bounds are filled in from the SIG edge contract.

#	Intent category	What to assert/Check
1	CDC	Latency bound on synchronizer handshake; no data change during the transfer window; gray-code or handshake invariant on multi-bit crossings.
2	Reset	Minimum gap between reset deassertion and first transaction; FSM initialization completion check before any incoming transfer; reset during in-flight transaction produces clean termination.
3	Protocol Translation	Correct phase mapping from source to target protocol; no illegal state transitions at the boundary; response code translation table exhaustively covered.
4	Back-Pressure	Ready or credit deassertion duration does not exceed the contracted timeout; internal buffers do not overflow; back-pressure correctly propagates upstream.
5	Burst Decomposition	Downstream address sequence matches expected per-beat decomposition; wrap boundaries produce the correct address pattern; narrow transfer byte-lane mapping verified against specification.
6	Error Propagation	Every downstream error produces exactly one upstream error; remaining burst beats are handled per the cross-boundary contract; error codes map deterministically.
7	Ordering	No stale data on read-after-write to the same address through internal buffers; exclusive monitor scope correctly enforced; in-order completion guarantees verified across the boundary.
8	Address Decode	No hang on unmapped address access; default slave responds within defined timeout; device-select one-hot enforcement on every transaction.
9	Data Integrity	Byte lanes preserved across domain crossings; no stale data on narrow transfers after CDC; ECC propagation across protocol boundaries verified end-to-end.
10	Power and Isolation	In-flight transactions complete or cleanly abort before isolation asserts; isolation clamp values match contract; wake-up handshake sequencing verified before next transaction is accepted.

Table 4.1 Bridge XTP categories and what each one verifies

Category	What it verifies
Bringup/Init	Single-clock-domain bringup, reset to a clean idle state, basic readiness
Connectivity	AHB and APB interface signal behavior, address/data/clock/reset connectivity
Register access	CTRL, STATUS, ERROR_ADDR, ERROR_INFO read/write, W1C and read-only fields
Main datapath	Single transfer and burst conversion, address and data forwarding
Protocol	APB phase and wait-state timing, handshake completion
Reset	Soft and hard reset, sticky-flag clearing, recovery to a working state
Error handling	Address error, timeout, PSLVERR detection, capture, and recovery
Cross-feature	Interactions: reset during a transfer, error under load, register access during back-pressure

Table 6.3 Subsystem-found defects by bug class, projected SoC discovery phase, and estimated schedule impact if discovery had been delayed to that phase.

Bug class	Bugs found at subsystem	Projected SoC discovery phase	Estimated schedule impact
IP Integration / Connectivity	4	SoC TB bring-up	3?5 weeks
Reset / Power Sequencing	3	SoC Boot Flow	5?7 weeks
Design Aliveness	6	SoC Data Path bring-up	6?8 weeks
Heavy Descriptor Flow	8	SoC Regression	12?16 weeks
Corner / Stress Scenarios	7	SoC Stress Testing	8?10 weeks
SW/HW Configuration Ordering	5	SoC Boot Flow	3?5 weeks
Shared Resource Ownership	4	SoC Stress Testing	8?12 weeks
Partial Reset / Recovery	3	SoC Recovery Testing	7?10 weeks
Total	40		

Claude.ai Pro subscription	\$20/month
API credits (generation only)	\$5–10/month typical
Questa FSE	Free
Python, Node.js, Git	Free
Total	\$25–30/month

Table 5.2 Failure classification and first action

Category	Definition	First action
Design bug	RTL behavior does not match the specification or XTP expected result	File it. Reproduce minimally. Fix the RTL. Add a regression test
Test logic issue	Scenario or test code is wrong; infrastructure is correct	Fix the scenario. Do not touch infrastructure. Do not update XTP unless intent was wrong
Infrastructure issue	Driver, monitor, scoreboard, clocking block, or environment has a defect	Stop all regression. Every result since the infrastructure was broken is suspect. Fix Chap. 3 before proceeding
Environment artifact	Run script, seed, tool version, or configuration produced a non-reproducible result	Fix the run configuration. Re-run. Do not count as a real failure