DESIGN AND IMPLEMENTATION OF A PYTHON-BASED AUTOMATION FRAMEWORK FOR ANALOG AND MIXED-SIGNAL (AMS) VERIFICATION

A Thesis submitted in partial fulfillment of the requirement for the Award of the Degree of

MASTER OF TECHNOLOGY

in VLSI Design

Submitted By

SHASHI KANT SHARMA

Roll No: 6024620022

Under Supervision of

Dr. Mayank Kumar Rai

[Professor, ECE Department]

Dr. Arnab Pattanayak

[Assistant Professor, ECE Department

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY

(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB

2024-2026

DECLARATION

I, Shashi Kant Sharma hereby declare that the work presented in this thesis entitled "Design and Implementation of a Python-Based Automation Framework for Analog and Mixed-Signal (AMS) Verification" in partial fulfillment of the requirement for the award of degree of Master of Technology (VLSI Design) submitted at Department of Electronics and Communication Engineering, Thapar Institute of Engineering & Technology (Deemed to be University), Patiala is an authentic record of work carried out under the supervision of Dr. Mayank Kumar Rai [Professor, ECE Department] , Dr. Arnab Pattanayak [Assistant Professor, ECE Department] and my industry mentors Mr. Vinayak Kamanuri and Mr. Vingnan Pusunula (Infineon Technologies, Bengaluru). from July 2025 to June,2026. The matter presented in this thesis has not been submitted either in part or full to any other university or institute for the award of any other degree.

Date: 28-07-2026

Shashi Kant Sharma

Roll Number: 6024620022

Dr. Mayank Kumar Rai

[Professor, ECE Department]

Dr. Arnab Pattanayak

[Assistant Professor, ECE Department]

Department of Electronics and Communication Engineering

Thapar Institute of Engineering & Technology

Vinayak Kamanuri

Manager

[Infineon Technologies India Private Limited, Bengaluru]

Certificate | Apprentice Program

Shashi Kant Sharma

Has successfully completed the Apprentice Program at Infineon Technologies India Private Limited from 1 July 2025 to 31 May 2026.

Ling Ming Shawn
Senior Director Emerging Talent

Tan Eunice
Senior Manager Emerging Talent

This certificate is computer-generated and does not require a signature

ACKNOWLEDGEMENT

I would like to express my heartfelt gratitude to the individuals who have played an instrumental role in my internship journey at INFINEON TECHNOLOGIES INDIA PRIVATE LIMITED. Their guidance, support, and encouragement have been invaluable, and I am truly grateful for their contributions.

I would also like to express my deep appreciation to my manager Mr. Pradeepkumar Marikundam and my mentors Mr. Vinayak Kamanauri and Mr. Vingnan Pusunula. Their leadership, guidance, and trust in my abilities have been invaluable. They provided me with challenging projects, allowed me to take ownership of my work, and provided constructive feedback that greatly contributed to my personal and professional development. I am grateful for their mentorship and for creating a positive and inclusive work environment that fostered learning and growth.

I would like to express my gratitude to my college mentors, Dr. Mayank kumar rai and Dr. Arnab pattanayak for their continuous support and guidance throughout my internship. Their advice, wisdom, and encouragement have been invaluable in helping me navigate through the challenges and make the most out of this experience. I am thankful for their unwavering support and for being a constant source of motivation, their guidance, support, and encouragement have been invaluable, and I am truly grateful for their contributions.

I am deeply grateful to Dr. Kulbir Singh, Head of the Department (ECE) and Dr. Shireesh Kumar Rai, MTech (VLSI) Coordinator, THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY, PATIALA, PUNJAB for providing me with a learning Environment and Infrastructure in ECED.

Place: Patiala, Punjab
Date: 28 – 07 - 2026

SHASHI KANT SHARMA
Roll No: 6024620022

ABSTRACT

The rapid growth in the complexity of mixed-signal integrated circuits has placed enormous demands on the analog and mixed-signal (AMS) verification process. Manual setup of testbenches, AMS Connect Files (AMSCFs), and simulation environments across multiple PVT corners is time-consuming and a significant source of inconsistency and error. This thesis presents the design and implementation of a Python-based automation framework that addresses these challenges in a systematic and integrated manner.

The proposed framework automates key steps in the AMS verification flow: netlist parsing for top-module and pin extraction, duplicate module detection, testbench template generation, AMSCF generation across PVT corners, simulation control, and regression management. Built around Real Number Modeling, the tool currently parses SystemVerilog and Verilog-AMS netlists and integrates with the industry-standard simulators Cadence Xcelium and Spectre; extending this support to SPICE-based netlists is identified as a direction for future work. A graphical user interface (GUI) enables engineers to interact with the framework without requiring deep knowledge of the underlying implementation.

The framework was validated on the netlist of the design under test, reducing full test environment setup time from one to two working days to a few minutes. A complete set of twenty-seven PVT corner AMSCF files was generated automatically, all verified to be syntactically correct, and the generated testbench template, with a One-Wire protocol instantiated, compiled successfully in Cadence Xcelium without manual correction. An AI agent built on LLM such as Claude Sonnet 4.6 was further integrated to generate protocol-aware test sequences from natural language prompts, producing multi-step analog stimulus from a single description, though minor human review remained necessary for timing-critical parameters. These outcomes confirm that the proposed framework substantially reduces

manual effort in AMS verification setup while remaining accessible to engineers across varying experience levels.

Keywords: AMS Verification, LLM (large language models) , Python Automation, Testbench Generation, AMSCF, PVT Corners, Real Number Modeling, SPICE, Cadence Xcelium, Spectre, Mixed-Signal Simulation

TABLE OF CONTENTS

Sr. No Name of the Chapters Page No

LISTS OF FIGURES

Sr. No **Figure Details** **Page No**

LISTS OF TABLES

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

The semiconductor industry has seen a remarkable growth in the complexity of integrated circuit (IC) designs over the past two decades. What once used to be simple digital logic blocks has now evolved into highly sophisticated systems that incorporate both analog and digital components on a single chip. These are commonly referred to as mixed-signal designs, and their verification remains one of the most challenging aspects of the entire IC design flow.

In the modern SoC (System-on-Chip) era, analog blocks such as PLLs, ADCs, DACs, LDOs, and bandgap references are routinely embedded alongside large digital subsystems. The correctness of these analog components, and more importantly their interaction with the surrounding digital logic, is absolutely critical to the overall functionality of the chip. A single error in the analog-digital interface can lead to catastrophic failures — failures that are often discovered very late in the design cycle, making them extremely expensive to fix.

Verification, in this context, is not just about checking whether a circuit works. It is about systematically ensuring that the design behaves correctly across all intended operating conditions — different process corners, supply voltages, and temperatures, collectively referred to as PVT corners. The number of scenarios to be verified can run into hundreds, sometimes thousands, and managing all of this manually is simply not practical at the kind of scale that modern chips demand.

1.2 MOTIVATION

The AMS verification flow, even in well-established EDA environments like Cadence Xcelium and Spectre, involves significant manual effort. Engineers are typically required to write testbenches by hand, configure simulation environments for each PVT corner separately, manage large numbers of simulation runs, and then collate and interpret results. This process is not only time-consuming but also prone to human error. A missing corner, an incorrectly configured testbench, or an overlooked module can silently compromise the verification coverage without anyone realizing it.

During the course of the internship that forms the basis of this thesis, it was observed that a significant portion of a verification engineer’s time is spent on tasks that are largely repetitive

in nature. Setting up an AMS Connect File (AMSCF), for instance, follows a fairly well-defined structure — yet it is written from scratch every time, for every corner, for every design. Similarly, testbench templates for different modules follow predictable patterns, yet they are manually recreated with slight variations each time.

This kind of repeated, structured work is precisely what automation is good at. Python, with its rich ecosystem of text-processing libraries and its ease of integration with shell environments and EDA tool flows, is a natural fit for building such automation. This thesis is therefore motivated by a practical need observed in real-world AMS verification: the need to reduce manual overhead, improve consistency, and make the overall verification process more systematic and less error-prone.

1.3 PROBLEM STATEMENT

Despite the availability of powerful simulation tools like Cadence Xcelium (for mixed-signal simulation) and Spectre (for analog SPICE-level simulation), the process of setting up and managing AMS verification remains largely manual. The specific problems that were identified, and which this work attempts to address, are as follows:

1. **Netlist Parsing:** Identifying the top-level module, its sub-modules, and pin definitions from a netlist file requires manual inspection. For large designs, this is tedious and error-prone.
2. **Duplicate Module Detection:** In complex hierarchical designs, the same module may appear multiple times under different instances. Without automated detection, this can lead to conflicts or redundant effort during testbench creation.
3. **Testbench Generation:** Writing testbench files for each module manually consumes significant engineering time and introduces inconsistencies across the verification suite.
4. **AMSCF Generation:** The AMS Connect File, which defines the interface between analog and digital simulation domains, must be generated for each PVT corner. Doing this manually for tens of corners is impractical.
5. **Simulation and Regression Management:** Running simulations across multiple corners and tracking pass/fail status requires a structured flow that is often absent in manual setups.

6. **Flow Support:** The verification environment must support both RNM (Real Number Model) based flows and SPICE-level flows, each with their own configuration requirements.

These problems collectively result in a slow, inconsistent, and difficult-to-scale AMS verification process. There is clearly a need for an integrated automation tool that can address all of these challenges within a unified framework.

1.4 OBJECTIVES OF THE WORK

Based on the problem statement outlined above, the following objectives were defined for this work:

- To develop a Python-based tool that can parse Verilog and SPICE netlists and automatically extract the top module, its sub-modules, and the associated pin definitions.
- To implement logic for detecting and flagging duplicate module instantiations within the design hierarchy.
- To automate the generation of testbench templates that are ready to be used with Cadence Xcelium and Spectre simulation environments.
- To enable automated generation of multiple AMSCF files corresponding to different PVT corners, thereby eliminating the need for manual configuration per corner.
- To integrate simulation control and regression management capabilities within the same framework.
- To provide a graphical user interface (GUI) that allows verification engineers to interact with the tool without needing to understand the underlying Python implementation.

1.5 SCOPE OF THE THESIS

This thesis covers the design, implementation, and validation of the above-mentioned automation framework. The work is primarily focused on the front-end of the AMS verification flow — from design input (netlist parsing) to simulation setup (testbench and AMSCF generation) and regression execution. Full post-processing of simulation results and formal

verification techniques are outside the scope of this work, though they are acknowledged as important areas for future development.

The tool has been developed and tested in an industry environment using Cadence Xcelium and Spectre as the primary simulation engines. The Python scripts have been written to be reasonably portable, though some aspects of the flow are necessarily tied to the specific tool invocation conventions of the Cadence environment.

1.6 ORGANIZATION OF THE THESIS

The rest of this thesis is organized as follows:

Chapter 2 presents a review of relevant literature, covering prior work in EDA automation, AMS verification methodologies, and Python-based scripting frameworks for VLSI design.

Chapter 3 provides the necessary technical background on AMS verification — including an overview of AMSCF, PVT corners, RNM vs. SPICE flows, and the roles of Cadence Xcelium and Spectre in the simulation environment.

Chapter 4 describes the architecture and implementation of the GUI developed as part of this work, discussing design decisions and the overall user interaction model.

Chapter 5 focuses on the core automation capabilities: testbench template generation and AMSCF automation, along with the netlist parsing and duplicate detection modules.

Chapter 6 presents the results and discusses the outcomes of the tool in terms of time savings, correctness, and coverage improvements observed during testing.

Chapter 7 concludes the thesis with a summary of contributions and outlines directions for future work.

CHAPTER 2

LITERATURE REVIEW

2.1 INTRODUCTION

Analog and mixed-signal (AMS) verification sits at one of the more challenging intersections in modern semiconductor design. As integrated circuits have evolved into complex, multi-domain systems, the task of confirming correct behaviour across the analog-digital boundary has become harder, more time-consuming, and far more configuration-intensive than what purely digital flows demand. The body of research addressing these challenges spans several decades and covers topics ranging from simulation methodologies and modelling languages to tool environments and, most recently, software-driven and AI-assisted automation.

This chapter reviews the existing literature directly relevant to the work carried out in this thesis. The discussion is organised into eight thematic areas: the foundational challenges of AMS verification, real number modelling techniques, connect modules and AMSCF file structures, the Cadence simulation environment, UVM methodology and its extension to mixed-signal contexts, PVT corner analysis, Python scripting and GUI automation, and AI-assisted and agent-driven verification. Each of these threads corresponds to a specific feature or motivation within the automation framework developed during the course of this internship project.

2.2 FUNDAMENTALS AND CHALLENGES OF AMS VERIFICATION

The difficulty of automating analog and mixed-signal design has been recognised for a long time. Gielen and Rutenbar [1], in one of the most widely cited surveys on computer-aided design for analog and mixed-signal ICs, traced the divergence between digital and analog design automation tools and clearly showed that while digital circuits benefited enormously from structured synthesis and verification flows, the analog domain remained largely resistant to similar automation because of the continuous, often nonlinear nature of analog behaviour. This foundational observation, made at the turn of the millennium, has only grown more relevant as the number of embedded analog blocks in modern SoCs has increased.

On the simulation side, Kundert [2] provided a thorough treatment of SPICE-based simulation, arguing that the accuracy of analog simulation must always be viewed in relation to its computational cost. Full transistor-level simulation of every analog block in a large SoC is practically infeasible — not because of accuracy limitations, but because of time. This

fundamental tension between accuracy and simulation speed has shaped almost every methodology development in the AMS space since.

A more recent industry perspective from Siemens EDA [3] highlighted that the productivity gap between digital verification and mixed-signal verification has not narrowed over time — it has widened. Digital flows now routinely employ constrained-random testing, assertion-based verification, functional coverage closure, and regression automation. AMS flows, by contrast, still require the manual construction of configuration files, testbench structures, and simulation setups for each new design or corner combination. This persistent manual overhead is precisely the problem that the tool developed in this thesis is designed to reduce.

2.3 REAL NUMBER MODELING FOR PRACTICAL AMS SIMULATION

Among the various approaches proposed for improving AMS simulation efficiency, real number modelling (RNM) has emerged as one of the most practically impactful. The idea is conceptually straightforward: rather than simulating analog blocks using transistor-level SPICE equations, their behaviour is abstracted into event-driven floating-point models that execute on the digital simulation engine. Brennan and Testorf [6], in a DVCon paper that provides one of the clearest practical introductions to RNM, described this as borrowing the continuous value representation of the analog domain while adopting the discrete event-driven execution model of digital simulation. The outcome is a simulation that is orders of magnitude faster than SPICE, yet preserves sufficient behavioural accuracy for most functional verification purposes.

SystemVerilog has become the preferred language for writing real number models in industry, largely because it allows digital verification engineers to work within a familiar framework while constructing analog behavioural abstractions. Georgoulopoulos and Hatzopoulos demonstrated this concretely by developing and validating RNM-based models for a Flash ADC using SystemVerilog [4]. Their work confirmed that such models, when compared against transistor-level SPICE references, produce results within acceptable accuracy margins for functional verification. A later study by the same group applied the same approach to a SAR ADC [5], reporting that the RNM-based simulation completed in approximately 0.5 seconds versus a Verilog-AMS reference that required 20 seconds — a roughly 40x speedup. This kind of quantitative result is significant because it makes concrete the practical case for RNM adoption.

From an industry standpoint, Cadence has documented RNM deployment within the Xcelium simulation environment in detail [7]. Their white paper on the subject notes that expanding

metric-driven verification to include real number models enables automated data collection and reporting, substantially reduces regression turnaround time, and makes it possible to apply standard UVM-based verification techniques — including constrained randomisation and functional coverage — to analog blocks that would otherwise require a separate, slower analog flow.

2.4 CONNECT MODULES AND THE AMS CONNECT FILE

A necessary but often underappreciated aspect of AMS co-simulation is the handling of signals at the boundary between analog and digital simulation domains. Analog signals are continuous electrical quantities; digital signals are binary logic values. When these two domains meet at a shared net boundary, a connect module is required to handle the domain conversion automatically. Frey and O'Riordan [8] provided one of the foundational treatments of this concept, describing how Verilog-AMS supports the automatic or manual insertion of connect modules at discipline boundary crossings in a design hierarchy. These modules are configured through connection rules and discipline declarations that govern how signals of different types interact at shared nets.

More recently, Cadence introduced the concept of Universal Connect Modules (UCM) [10], which consolidate multiple legacy connect module configurations into a single, parameterisable interface element that can be automatically inserted by the simulator using information in the AMSCF file. The AMS Connect File itself is a Spectre-format control file that drives the entire co-simulation setup. As described in the NUS Mixed Signal Design Simulation Manual [11] and further elaborated in the Renesas mixed-signal verification whitepaper [9], the AMSCF carries information about technology corners, supply voltages, temperature settings, and connect module configurations. Each PVT corner may require a different AMSCF, with adjusted supply and temperature parameters and potentially different connect library settings for blocks operating at non-standard voltage levels.

Preparing these files manually across a full corner matrix is tedious and error-prone. It is, in effect, a repetitive structured task with a well-defined format that varies only in its parameter values — exactly the kind of task that is well suited to automation. This observation is one of the central motivations for the multi-corner AMSCF generation capability described in this thesis.

2.5 EDA TOOL ENVIRONMENT: CADENCE XCELIUM AND SPECTRE

The simulation infrastructure underpinning the work in this thesis consists of Cadence Spectre, used for analog and SPICE-level simulation, and the Cadence Xcelium logic simulator, which serves as the primary digital and mixed-signal simulation engine. The Spectre AMS Designer and Xcelium co-simulation environment supports two primary use models [12]: the Virtuoso-centric AVUM, suited for analog design teams working within the graphical Virtuoso environment, and the command-line-oriented AXUM, which uses Xcelium as the lead simulator and is far more amenable to scripting and automation. The tool developed in this thesis operates entirely within the AXUM model, invoking simulations through structured command-line calls.

Xcelium, as described in the Cadence datasheet [13], provides best-in-class simulation engine performance across SystemVerilog, VHDL, SystemC, UVM, and mixed-signal flows, with support for both SPICE netlists and real number models within the same simulation run. The introduction of Xcelium Apps in 2022 [14] added domain-specific capabilities including a dedicated Mixed-Signal App that enables native co-simulation with Spectre. Cadence reported that combining multiple Xcelium Apps can deliver up to 10x improvement in regression throughput. Understanding the command-line interface and file input structure of this environment was foundational to the simulation control and regression management features of the automation framework described in this thesis.

2.6 UVM METHODOLOGY AND ITS EXTENSION TO MIXED-SIGNAL CONTEXTS

The Universal Verification Methodology (UVM), standardised through the Accellera UVM 1.2 User's Guide [15] and later formalised as IEEE Standard 1800.2 [16], has become the dominant framework for functional verification of digital designs. Its SystemVerilog-based class library provides reusable verification components — sequencers, drivers, monitors, scoreboards, and agents — that can be assembled into scalable, modular testbench architectures. The coverage-driven and constrained-random capabilities of UVM have made it an industry standard in ASIC and SoC verification.

Extending UVM to cover analog and mixed-signal designs introduces complications, primarily because UVM operates within a digital simulation kernel while AMS verification requires coordinated interaction with an analog solver. Georgoulopoulos et al. [18] addressed this by developing a UVM-based verification architecture for a Flash ADC real number model, showing that UVM components can successfully drive and monitor analog blocks modelled

using SystemVerilog RNMs. Similarly, work published at DVCon on DV-UVM-based AMS co-simulation [28] demonstrated that integrating RNM and SPICE blocks into a UVM testbench environment substantially improves functional coverage while eliminating simulation bottlenecks.

A significant recent development is the formal approval of the UVM Mixed-Signal (UVM-MS) 1.0 standard by Accellera in January 2025 [17]. This standard extends the base UVM IEEE 1800.2 classes to formally support AMS-specific verification components, including mechanisms for driving and checking analog signal quantities in a UVM-driven testbench. The approval of this standard reflects a growing industry consensus that AMS verification must be treated as a first-class concern within the UVM ecosystem. Separately, Edelman and Bhutada [22] demonstrated that the structural boilerplate of a UVM testbench — sequences, agents, scoreboards, and port declarations — can be generated automatically from a description of the design interface, reducing construction time from days to hours. This approach is directly analogous to the testbench template generation capability of the automation framework described in this thesis.

2.7 PVT CORNER ANALYSIS IN AMS VERIFICATION

Process, voltage, and temperature (PVT) corners represent the operational extremes within which a chip must correctly function. Process corners reflect manufacturing variability, typically labelled as TT (typical-typical), FF (fast-fast), SS (slow-slow), and cross corners such as FS and SF. Adding voltage and temperature variation to this matrix results in a large number of combinations that must each be simulated before a design can be signed off. Lin et al. [19] pointed out in a DVCon paper that traditional corner simulation is based on the assumption that worst-case behaviour occurs at extreme PVT values, and that for analog and mixed-signal designs this assumption does not always hold — nonlinear signal behaviour means that genuine worst-cases may lie between standard corners. Their work proposed machine learning-based sampling to identify critical corners more efficiently.

From a practical automation standpoint, the challenge is one of scale and repetition. Each PVT corner requires a differently parameterised AMSCF file, a corresponding netlist configuration, and a dedicated simulation invocation. Doing this by hand for tens of corners across multiple design variants is unsustainable in any industrial schedule. The multi-corner AMSCF generation capability of the tool described in this thesis directly addresses this bottleneck.

2.8 PYTHON SCRIPTING AND GUI AUTOMATION IN VERIFICATION FLOWS

Python has, over the past decade, evolved from a general scripting language into a central tool in the VLSI verification engineer's workflow. Keysight's documentation on Python APIs for EDA [20] illustrates how Python can drive commercial EDA tools, automate simulation flows, parse results, and support machine learning workflows — all within a single scripting environment. Its readable syntax, rich standard library, and ease of integration with shell environments make it well suited for building automation tools that invoke Xcelium and Spectre through structured command-line calls.

A specific area where Python automation has shown tangible impact is UVM testbench generation. Edelman and Bhutada [22] described a methodology where boilerplate UVM code is produced automatically from structured design information, significantly reducing the time needed to stand up a working simulation environment. The approach used template-driven code generation — a concept central to the testbench template module of this thesis, which uses netlist-parsed information to automatically produce wire declarations, module instantiations, and port mappings.

For the graphical interface layer, the Python Tkinter library [21] was used in this project. Tkinter is part of Python's standard library and provides a lightweight, cross-platform toolkit for building desktop GUI applications. Its use in EDA automation tools is well established, and it requires no external dependencies beyond a standard Python installation — a practical advantage in restricted industrial environments. Providing a GUI rather than requiring direct script execution is a deliberate design choice aimed at lowering the technical barrier for verification engineers who may not be experienced in Python, enabling them to interact with the full automation flow through a structured graphical interface.

2.9 AI-ASSISTED AND AGENT-DRIVEN VERIFICATION APPROACHES

The application of large language models (LLMs) to hardware verification has moved quickly from exploratory research into concrete working systems. AutoBench, introduced by Qiu et al. at MLCAD 2024 [23], was the first published LLM-based testbench generator for digital circuit design. It requires only a natural-language description of the design under test and produces a comprehensive testbench using a hybrid structure that separates Verilog drivers from Python-based checking logic. A self-debugging mechanism — where the framework detects errors in generated code and iteratively corrects them — substantially improved the pass rate over direct one-shot LLM generation. CorrectBench [24], the successor framework, extended this with

functional self-validation, reporting a testbench pass ratio of 70.13% compared with AutoBench's 52.18% and a baseline single-step approach of 33.33%.

ChipNeMo [25], developed by NVIDIA Research, approached the problem differently. Rather than applying a general-purpose LLM to chip design tasks, the authors fine-tuned a domain-adapted model on a corpus of chip design data including RTL code, verification testbenches, EDA scripts, and hardware documentation. The model was evaluated on three tasks: an engineering assistant chatbot, EDA script generation, and bug summarisation. Notably, the domain-adapted model outperformed GPT-4 on the assistant chatbot and script generation tasks, demonstrating that specialisation on domain-specific data provides a real advantage in chip design contexts.

For multi-agent architectures, Liu et al. [26] proposed a multi-agent generative AI framework for IC module-level verification, in which distinct agents handle different stages of the workflow: specification parsing, verification strategy generation, and code implementation. Their experiments showed that this decomposed collaborative approach significantly outperformed single-LLM methods in producing correct and complete testbenches across modules of varying complexity. This architecture — where responsibilities are separated across specialised components — is directly analogous to the two-layer design adopted in this thesis: a scripted module that handles structural template generation from netlist data, and an LLM-based agent (using Claude Sonnet 4.6 as the backend model) that generates behavioural test sequences such as supply ramp waveforms, protocol-driven address and data stimulus, and signal transition patterns.

NVIDIA's developer blog on AI agents for test case creation [27] provided a practically grounded account of how LLM-based agent frameworks can not only generate test stimuli but also analyse simulation error logs and suggest corrective actions. This error diagnosis capability — identifying missing files, flagging incorrect path references, and advising on compilation fixes — directly parallels the guidance function implemented in the agent component of this thesis, which assists the user in resolving environment and configuration issues that arise during simulation setup.

2.10 RESEARCH GAP AND POSITIONING OF THE PRESENT WORK

The literature reviewed in this chapter reveals a consistent and practically significant set of unaddressed challenges in AMS verification. Real number modelling is well established as a speed-efficient simulation approach [4][5][6], but the infrastructure needed to deploy it — AMSCF files, testbench templates, PVT corner configurations — is still prepared manually in

most teams. EDA tools such as Xcelium and Spectre are powerful but require extensive manual input to configure for each simulation run [12][13]. UVM has been extended conceptually to cover AMS contexts [15][16][17][18], yet testbench construction remains a substantial manual effort. Python automation has been shown to address parts of this problem [20][22], but there is no published integrated tool that handles the complete flow from netlist parsing through to multi-corner regression execution within the Cadence AMS environment.

On the AI side, LLM-based testbench generation [23][24] and domain-adapted engineering assistants [25] have demonstrated real promise, but existing work focuses largely on digital designs. The application of agentic LLM systems specifically to AMS test sequence generation — where the stimulus must reflect analog behaviours such as supply ramping, protocol timing, and continuous signal transitions — remains largely unexplored in the published literature. The framework developed in this thesis addresses both gaps simultaneously: a Python-scripted automation layer for structural setup and configuration management, combined with an LLM agent for behavioural test sequence generation and error diagnosis, deployed within the Cadence Xcelium and Spectre AMS simulation environment.

CHAPTER 3

AMS VERIFICATION BACKGROUND

3.1 INTRODUCTION

This chapter provides the technical background necessary to understand the design and implementation of the automation framework described in Chapters 4 and 5. While Chapter 2 reviewed the broader research landscape around AMS verification, the present chapter focuses on the specific concepts, file formats, simulation flows, and tool interfaces that are directly relevant to the internship project.

The discussion begins with a high-level overview of the AMS verification flow, establishing the context within which the tool operates. This is followed by a detailed examination of netlist structure and design hierarchy, which forms the primary input to the automation framework. The chapter then covers the technical principles of real number modelling, the structure and configuration of the AMS Connect File (AMSCF), the PVT corner framework, the Cadence Xcelium and Spectre simulation environment, and finally the UVM testbench architecture as it applies to mixed-signal designs. Together, these topics build the foundation upon which the subsequent implementation chapters rest.

3.2 OVERVIEW OF THE AMS VERIFICATION FLOW

At its highest level, the AMS verification flow is a pipeline that takes a design description as input and produces simulation results that confirm or refute the design's functional correctness across all intended operating conditions. Unlike a purely digital verification flow, where the testbench and design under test (DUT) both reside in the same simulation domain, an AMS flow must coordinate two distinct simulation engines — a digital logic simulator and an analog solver — along with the interface mechanisms that connect them.

The primary inputs to the AMS verification pipeline are: the design netlist (written in Verilog, SystemVerilog, or SPICE format depending on the abstraction level); the testbench that applies stimulus to the DUT and checks responses; the AMS Connect File (AMSCF) that configures the co-simulation environment; and the process design kit (PDK) model libraries that describe device characteristics for each process corner. The outputs are simulation waveforms, log files, functional coverage reports, and pass/fail status for each simulation run.

The automation tool developed in this thesis operates at the setup stage of this pipeline — specifically at the point where the netlist is parsed, the testbench is constructed, and the AMSCF files are generated for each PVT corner. This is consistently the most labour-intensive stage in a manual verification flow, and it is where the tool delivers the most significant time savings.

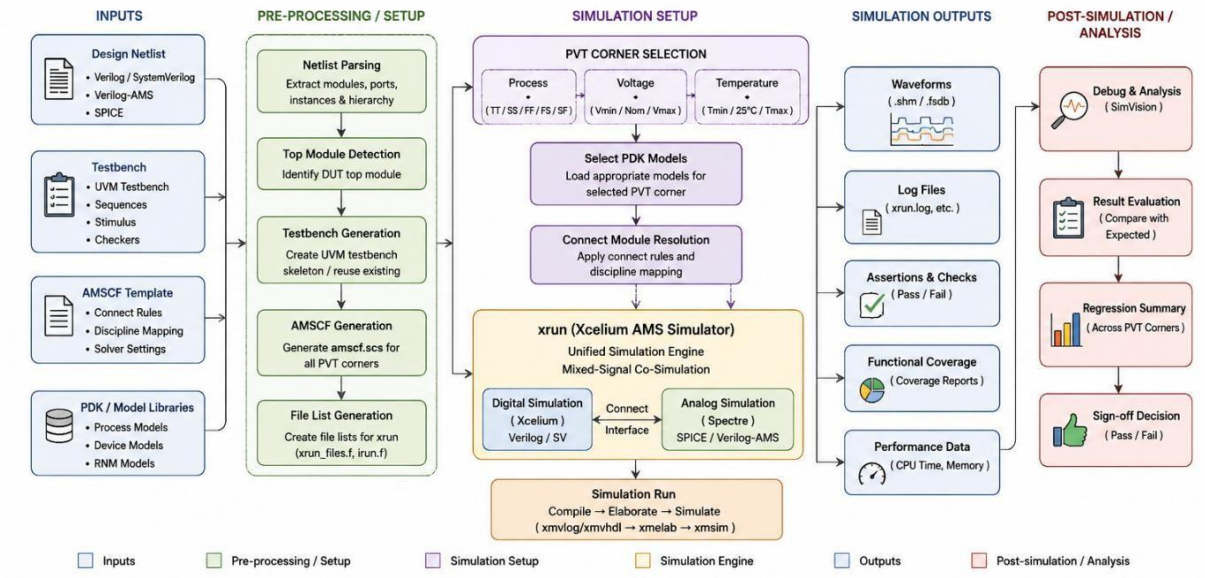


Figure 3.1 High-level AMS verification flow from netlist input to simulation result

The figure 3.1 shows the complete AMS verification flow. The process starts with design inputs including the netlist, testbench, AMSCF template, and PDKmodels. In the pre-processing stage, the netlist is parsed, the top module is identified, the testbench and AMSCF files are generated, and file lists are prepared for simulation. During simulation setup, appropriate PVT corners and models are selected, and connect rules are applied for discipline mapping.The xrun simulator performs unified mixed-signal co-simulation by coupling the Xcelium digital engine with the Spectre analog engine through connect modules. The outputs include waveforms, logs, coverage, and performance data, which are then analyzed using SimVision and other tools to determine the verification status across all PVT corners.

3.3 NETLIST STRUCTURE AND DESIGN HIERARCHY

A netlist, in the context of VLSI design, is a textual description of a circuit in terms of its constituent modules and the connections between them. In an AMS design, netlists may be written in Verilog or SystemVerilog (for digital and RNM blocks), Verilog-AMS (for behavioural analog models), or SPICE (for transistor-level analog subcircuits). The automation tool described in this thesis parses both Verilog and SPICE-format netlists, making it versatile across different levels of design abstraction.

3.3.1 Verilog Netlist Format

A Verilog module is the fundamental building block of a Verilog netlist. Each module declaration begins with the module keyword, followed by the module name, a port list, and the port direction and type declarations. A simplified representation is as follows:

```
module module_name (
    input  wire  signal_a,
    output wire  signal_b,
    inout  wire signal_c
);
    // internal declarations and logic
endmodule
```

Figure 3.2 Verilog module declaration structure

Figure 3.2 depicts the port list defines the interface of the module to the outside world. Ports are characterised by their direction (input, output, or inout), their data type (wire, reg, or real-valued types for RNM), and their name. In mixed-signal designs that use real number modelling, analog ports are declared using the wreal net type, which instructs the simulator to treat the signal as a floating-point real value rather than a binary logic value. The netlist parser in the automation tool extracts the module name, the port names, and the port directions from these declarations as the first step of its processing pipeline.

Module hierarchy is established through instantiation. When one module instantiates another, it creates an instance of that module with specific net connections. In a complex SoC netlist, the top-level module may instantiate several sub-modules, each of which may in turn instantiate further sub-modules, forming a tree-like hierarchy. The parser must traverse this hierarchy to identify the top-level module, enumerate all sub-modules, and detect any cases where the same module is instantiated more than once at the same hierarchical level — a condition that, if undetected, can lead to conflicts during testbench construction.

3.3.2 SPICE Netlist Format

SPICE netlists describe analog circuits at the transistor level. The fundamental unit in a SPICE netlist is the subcircuit, defined using the .subckt directive. The general form is shown below in Figure 3.3

```
.subckt module_name  port1  port2  port3
* component descriptions (resistors, capacitors, transistors, etc.)
.ends module_name
```

Figure 3.3 SPICE subcircuit declaration structure

Figure 3.2 depicts the structure of a Verilog module declaration, showing the module keyword, the port list, and the input, output, and inout qualifiers used to define a module's external interface. When a SPICE-level block is used in an AMS simulation alongside a Verilog testbench, the simulator must handle the translation between the digital and analog domains at the module boundary. The AMSCF specifies how this translation is performed, using connect modules to bridge the two domains. The parser in the automation tool handles SPICE netlists by identifying .subckt declarations, extracting the subcircuit name and port list, and incorporating this information into the testbench template alongside any Verilog modules present in the design.

3.4 REAL NUMBER MODELING: CONCEPTS AND IMPLEMENTATION

Real number modelling (RNM) is a simulation technique that represents analog signal values as floating-point real numbers, processed entirely within the digital simulation engine, without invoking a SPICE analog solver. The approach exploits the fact that for most functional verification purposes, the precise continuous-time waveform of an analog signal is less important than its effective value at the points where it interacts with digital logic. By abstracting the analog block into a behavioural model that outputs and accepts real values, the simulation can run at near-digital speeds while still capturing the essential analog characteristics of the design.

3.4.1 The wreal Net Type

The wreal net type is the primary mechanism for passing real-valued signals across module boundaries in a SystemVerilog RNM environment. When a port is declared as wreal, the simulator treats it as a floating-point-valued net rather than a binary logic net. At elaboration time, the simulator automatically performs wreal coercion — propagating the wreal type along connected net segments so that the entire signal path between analog modules is consistently typed as real-valued. A port declaration using wreal for a simple LDO behavioral model would appear as follows:

```
module ldo_rnm (
    input  wreal vin,
    input  wreal vdd,
    output wreal  vout,
    input  logic enable
);
```

Figure 3.4 wreal port declaration in a SystemVerilog RNM module

Figure 3.4 explains how a wreal port is declared inside a SystemVerilog real-number-modelled module, showing input, output, and supply ports typed as wreal so the simulator treats them as analog values. This declaration tells the simulator that vin, vdd, and vout are real-valued analog signals, while enable is a conventional logic signal. The testbench generator developed in this thesis must correctly identify wreal ports in the netlist and declare the corresponding signals in the testbench using real or wreal types, distinguishing them from logic-typed ports which require a different declaration and driver construct.

3.4.2 RNM Versus Other Modeling Approaches

The choice of modeling approach in AMS verification involves a fundamental trade-off between simulation accuracy and simulation speed. At one extreme, a full SPICE transistor-level simulation delivers high accuracy but is computationally intensive — making full-chip regression simulations across many PVT corners impractical. At the other extreme, a purely digital abstraction ignores analog signal characteristics entirely. Verilog-AMS sits between these extremes, supporting both continuous-time analog constructs and discrete digital events, but requiring the analog solver to be active even for behavioural models that do not need it.

RNM, implemented in SystemVerilog, occupies a practically useful position in this space. It runs entirely on the digital engine, achieves simulation speeds comparable to a purely digital simulation, and yet can represent the key input-output characteristics of an analog block with sufficient fidelity for functional verification. As demonstrated in the literature [5], this approach can yield speedups of 40x or more over Verilog-AMS while maintaining acceptable accuracy. For the regression-scale verification that the automation tool in this thesis enables, RNM is the default and recommended flow.

3.4.3 Selecting Between Behavioural Views Using Verilog Configuration Blocks

In a typical AMS project, the same design block may exist in more than one representation by the time verification reaches the AXUM stage — a Verilog-AMS or SPICE-level view created for analog sign-off, and a SystemVerilog real-number-modelled view created specifically for fast digital-centric regression. Rather than editing the netlist or testbench source every time the active view needs to change, Verilog provides a configuration construct, declared with the config and endconfig keywords, that performs this binding entirely outside the module source.

A configuration block names a design root and then maps individual cells or instances onto specific library sources using two complementary clauses: an instance clause binds a particular hierarchical instance path to a chosen view, while a cell clause applies the same binding to every instance of a given module name wherever it appears in the hierarchy. Each binding is

expressed using either a use clause, which names the exact library and configuration to bind, or a liblist clause, which gives an ordered list of libraries to search, with the first library in the list taking priority over the next. Because the configuration is specified as a file separate from the module declarations themselves, the underlying behavioural or structural source code does not need to be touched at all when switching which view is simulated — only the configuration file is edited.

```
config bind_example;
    design         sv_lib.TOP_TB;
    instance       TOP_TB.INST1.DUT_VAMS  liblist  vams_lib;
    instance       TOP_TB.INST1.DUT_RNM    liblist  sv_lib;
endconfig
```

Figure 3.5 Representative Verilog configuration block binding two instances to different library views

Figure 3.5 depicts a Verilog configuration block that binds two instances of the same design to different library views, letting the simulator switch between SPICE and SystemVerilog models without editing source files. For the framework developed in this thesis, this mechanism is what allows the same testbench, generated once by the tool described in Chapter 5, to be pointed at either a Verilog-AMS reference model or the SystemVerilog RNM model of a given block for comparison purposes, simply by changing the liblist ordering in the configuration file passed to the AXUM xrun invocation — without regenerating or editing the testbench itself.

3.5 THE AMS CONNECT FILE (AMSCF): STRUCTURE AND CONFIGURATION

The AMS Connect File is a Spectre-format control file that governs the entire co-simulation setup in an Xcelium AMS simulation. Every mixed-signal simulation run requires at least one AMSCF, and for regression-scale verification across multiple PVT corners, each corner configuration typically requires its own AMSCF with appropriately adjusted parameters. The file is passed to the simulator at invocation time using the -amscf option of the xrun command.

The AMSCF begins with a simulator language declaration, which identifies the file as Spectre-format and sets the context for subsequent statements. Model library files are loaded through include statements, which point to the specific PDK model files for the process corner being simulated. The core of the file is the amsd block, which contains the mixed-signal configuration directives. A representative AMSCF structure is illustrated below:

```
// AMS Connect File for TT corner, 1.8V, 27 degrees C
simulator lang=spectre
.tran tran temp =27
```

```
include "/path/to/pdk/models/tt.scs"

amsd {
    ie vsup=1.8
    portmap subckt=ldo_rnm  autoconnect=yes
    connect_rules  ucm_18v
}
```

Figure 3.6 Representative AMSCF structure for a TT corner simulation at 1.8 V and 27°C

Figure 3.6 describes a representative AMSCF file for a TT process corner at 1.8 V and 27°C, showing the include statement, voltage source definitions, and the connect-rule directive used during simulation. The ie directive within the amsd block sets the supply voltage (vsup) and temperature for the simulation. The portmap directive tells the simulator which sub-circuits should be automatically connected to the digital testbench, and the connect_rules directive specifies the connect module library to use for domain boundary handling. For designs with blocks operating at multiple supply voltages, separate portmap entries with different connect rules may be needed for each block.

Generating an AMSCF manually for one corner is a straightforward if tedious task. But as the PVT corner matrix grows — five process corners combined with three voltage levels and three temperature points yields 45 configurations, each requiring its own include statement pointing to a different model file and its own ie parameters — the task quickly becomes unmanageable without automation. This is the specific problem that the AMSCF generation module of the tool in this thesis addresses.

3.5.1 Connect Modules and Discipline Resolution in Practice

Building a connect module from scratch is rarely necessary in day-to-day verification work, since Spectre AMS Designer ships with a library of built-in connect rules covering the common voltage and logic-family combinations encountered in industrial designs. Where a design requires behaviour not covered by these defaults, the environment supports the development of user-defined connect modules, allowing the verification engineer to control precisely how analog-to-digital and digital-to-analog signal conversion is performed at a given boundary.

In the AXUM flow used in this thesis, connect-rule selection is expressed directly inside the AMSCF file through the connect_rules directive shown earlier in Figure 3.5. In the AVUM flow, the equivalent configuration is performed interactively through the Analog Design Environment, where a default connect-rule set (typically covering a standard supply-voltage configuration) is provided out of the box and can be customised by the user through the hierarchy editor as needed. Either way, the underlying mechanism is the same: a connect rule

maps a discipline boundary to a specific connect-module implementation, and discipline resolution determines, for every net that crosses an analog-digital partition, which connect module is automatically inserted at elaboration time.

3.6 PVT CORNER FRAMEWORK

Process, voltage, and temperature (PVT) corners define the operational envelope within which a chip must function correctly. Verification across this envelope is not optional — a design that passes at nominal conditions but fails at a manufacturing or operational extreme will result in field failures or yield loss. The corner framework therefore defines the minimum set of simulation conditions that the design team commits to verifying before tape-out.

3.6.1 Process Corners

Process corners reflect the variability inherent in semiconductor manufacturing. Even within a single wafer lot, individual devices will have slightly different threshold voltages, carrier mobilities, and oxide thicknesses depending on their exact position and the local conditions during fabrication. Process corner models capture the range of this variation by providing worst-case fast and slow parameter sets for NMOS and PMOS devices. The five standard corners are:

TT (Typical-Typical): Nominal NMOS and PMOS parameters. Represents the expected average behaviour.

FF (Fast-Fast): Both NMOS and PMOS devices are fast (low threshold voltage). Favourable for speed but can cause hold violations and higher current.

SS (Slow-Slow): Both devices are slow (high threshold voltage). Worst case for timing closure and minimum operating frequency.

FS (Fast-Slow): Fast NMOS, slow PMOS. This cross-corner is particularly relevant for circuits sensitive to NMOS-PMOS balance, such as transmission gates and certain analog comparators.

SF (Slow-Fast): Slow NMOS, fast PMOS. Relevant for similar reasons, with the skew in the opposite direction.

3.6.2 Voltage and Temperature Variation

Supply voltage corners typically span a range of ±10% around the nominal supply. For a nominal 1.8 V design, for example, simulations would be run at 1.62 V (low), 1.8 V (nominal),

and 1.98 V (high). Low voltage corners stress timing and functionality; high voltage corners can affect reliability and leakage. Temperature ranges depend on the product category — commercial designs are typically verified from 0°C to 85°C, industrial from −40°C to 85°C, and automotive from −40°C to 125°C or even 150°C.

The combination of process, voltage, and temperature defines the full corner matrix. A fairly typical industrial setup might involve 5 process corners × 3 voltage levels × 3 temperature points, yielding 45 corner combinations. Each combination requires its own parameterised AMSCF, its own xrun invocation, and its own results directory. Managing this matrix manually, especially when the design changes and all files must be regenerated, is one of the most time-consuming aspects of AMS verification setup — and is the core problem that the automation tool in this thesis solves.

3.7 CADENCE XCELIUM AND SPECTRE: SIMULATION FLOW

Cadence provides two primary use models for AMS co-simulation: the AVUM (AMS Designer Virtuoso Use Model), which is centred on the Virtuoso graphical environment and is typically favoured by analog designers, and the AXUM (AMS Designer Xcelium Use Model), which is a command-line-driven flow with Xcelium as the primary simulation engine. The tool developed in this thesis operates entirely within the AXUM model, which is more amenable to scripting, automation, and regression-scale execution.

3.7.1 The AVUM and AXUM Use Models

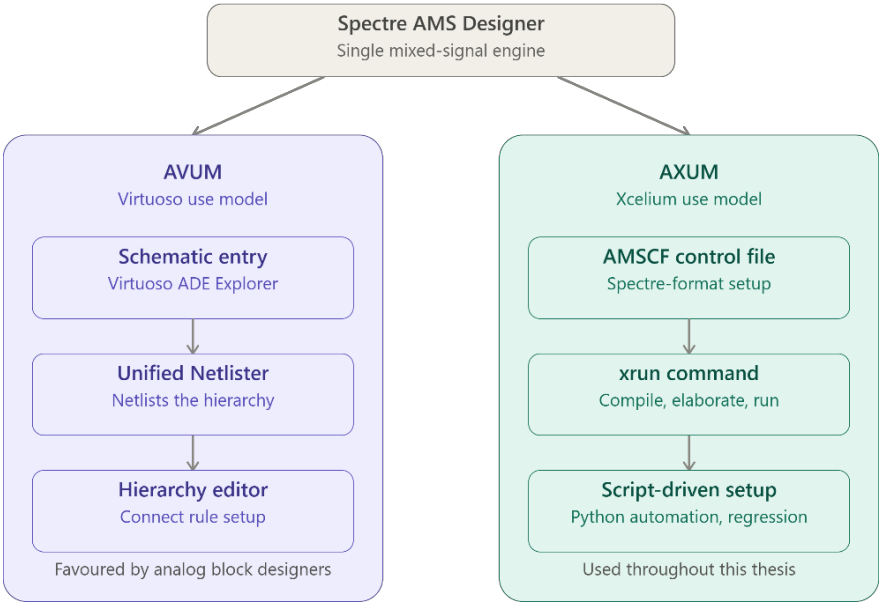


Figure 3.7 Cadence Spectre division for AMS verification

Figure 3.7 explains how Cadence divides AMS verification into the AVUM and AXUM use models, contrasting the Virtuoso-centric graphical flow with the Xcelium-centric command-line flow used throughout this framework. Cadence packages its mixed-signal simulation capability into two distinct use models, each aimed at a different kind of user and a different point in the design flow. The AMS Designer Virtuoso Use Model (AVUM) is built around the Virtuoso Analog Design Environment. In this flow, the engineer runs simulations on designs netlisted with the Unified Netlister inside ADE Explorer, using the xrun executable supplied by the Spectre AMS Designer simulator, and uses the Virtuoso Hierarchy Editor to build design configurations while exploring connect-module usage for signals that cross the analog-digital boundary. AVUM is generally favoured by analog designers who are already working schematic-first inside Virtuoso, since it lets them stay within the same graphical environment for both circuit design and mixed-signal simulation setup.

The AMS Designer Xcelium Use Model (AXUM), by contrast, is a command-line-based mixed-signal simulation flow tailored for digital-centric verification environments, where the engineer configures AMS designs, prepares mixed-language testbenches, and runs simulations directly through the xrun command, with connect-module insertion and discipline resolution handled through the same control file rather than through a graphical hierarchy editor. This is the flow used throughout the present work, since it integrates naturally with Python-based scripting and regression automation, and does not depend on a Virtuoso schematic database being present for the design.

Regardless of which use model is chosen, Spectre AMS Designer is positioned as a single simulation executable that accepts Verilog-AMS, VHDL-AMS, SystemVerilog, and SPICE-level descriptions interchangeably, and a single control file is used to define how analog blocks are integrated into the surrounding digital SoC — which means analog and RTL blocks can be substituted for one another to trade accuracy against simulation performance. The AMSCF file described in Section 3.5 is precisely this control file in the AXUM context.

3.7.2 The xrun Command

The xrun command is the unified compilation and simulation driver in the Xcelium environment. It accepts design source files in a range of formats — including Verilog, SystemVerilog, VHDL, SPICE, and Verilog-AMS — and manages the full flow from compilation through elaboration to simulation in a single invocation. A representative xrun command for an AMS simulation is shown below:

```
xrun  \
```

```
-f  filelist.f                    \
-amscf  amscf_tt_27.scs           \
-top  tb_top                      \
-access  +rwc                     \
-log  sim_tt_27.log               \
+ams_discipline_lib               \
-amsconnrules  ucm_18v
```

Figure 3.8 Representative xrun command for an AXUM flow AMS simulation

Figure 3.8 depicts a representative xrun command for an AXUM-flow AMS simulation, showing the file list, AMSCF reference, top-level testbench, access options, and connect-rule arguments passed to the simulator. The -f option points to a file list that enumerates all source files. The -amscf option passes the AMS Connect File for the specific corner being simulated. The -top option specifies the top-level testbench module. The -access option controls signal probing permissions, and the -log option directs the simulation log output to a named file. The +ams_discipline_lib and -amsconnrules options configure the analog discipline library and the connect module rules to be used at domain boundaries.

3.7.3 Simulation Phases

A complete Xcelium simulation run proceeds through three distinct phases. In the compilation phase, the simulator processes all source files, checks them for syntax, and converts them into an intermediate representation stored in the xcelium.d build directory. In the elaboration phase, the design hierarchy is assembled, module ports are connected, and the net types throughout the hierarchy are resolved — including wreal coercion for real-valued analog paths. In the simulation phase, the testbench drives the DUT according to the defined stimulus, the simulator tracks signal values, and results are written to waveform databases and log files.

In the AXUM flow, the three phases driven by xrun are each handled by a dedicated underlying executable, which the user does not normally invoke directly but which appears by name in error messages and log files. Compilation is performed by xmvlog (for Verilog and SystemVerilog sources) and xmvhdl (for VHDL sources), elaboration is performed by xmelab, and the simulation itself is run by xmsim — with the “xm” prefix referring to the Xcelium engine, in the same way that the legacy “nc” prefix referred to the earlier NC-Verilog/Incisive engine that preceded it. This breakdown is useful in practice: compilation errors are reported under the xmvlog or xmvhdl banner, elaboration errors such as unresolved port connections are reported under xmelab, and run-time errors appear under xmsim, so the prefix in a given error message immediately indicates which phase of the flow has failed.

For regression management, the automation tool generates a shell script that invokes xrun separately for each PVT corner, using the corresponding AMSCF file and a unique log file

name for each run. After all runs complete, the tool parses the log files for pass/fail keywords and compiles a summary report. This end-to-end regression orchestration eliminates the need to manually track which corners have been run and which have produced errors.

3.7.4 Debugging and Waveform Analysis with SimVision

Once a simulation has run, the engineer needs a way to inspect what actually happened on the DUT's signals, and this is the role played by SimVision. SimVision is Cadence's integrated debug environment for Xcelium simulations, combining a design browser, source-code browser, signal-flow browser, and waveform viewer within a single interface. Waveform data is stored in a dedicated simulation history database, and the viewer allows the engineer to trace a signal back to the exact line of source code or testbench sequence that produced it — which is particularly valuable when a generated AMS test sequence behaves differently from what was expected, for instance when verifying that a supply ramp generated by the AI agent described in Section 5.8 actually reaches its intended target voltage within the expected time window.

In the framework developed for this thesis, SimVision is invoked automatically whenever the -gui option is appended to the xrun command described in Section 3.7.2, opening directly into the waveform database produced by that simulation run. For regression-scale runs launched without the -gui option, the same waveform database can be opened separately in SimVision after the run completes, which is the mode used most often once a design has moved past initial debug and into routine multi-corner regression.

3.7.5 Tcl-Based Waveform Probing

Beyond the graphical waveform window described in Section 3.7.4, SimVision and the underlying Xcelium simulation kernel expose their probing and database functionality directly through Tcl, the command interpreter used throughout the Cadence command-line environment. A short Tcl script, passed to xrun through the -input option, can open a waveform database, instruct the simulator which signals to record and at what hierarchical depth, run the simulation for a specified duration, and then close the database — all without ever opening the graphical interface.

The two commands at the centre of this mechanism are database, which opens or closes a named waveform database, and probe, which creates a probe object that determines which signals within a given scope are recorded into that database. A probe can be scoped to record only the top-level ports of a module, or extended with a depth qualifier to recursively capture

every signal in every sub-instance beneath it — the configuration most often used during the early debug of a newly generated testbench. A representative probe script is shown below:

```
database -open waves -shm -into regression_waves.shm -default
probe   -create top_tb -depth all -all -database waves
run
exit
```

Figure 3.9 Representative Tcl probe script for batch-mode waveform capture

Figure 3.9 describes a representative Tcl probe script used for batch-mode waveform capture, showing the database, probe, run, and exit commands that record signals without opening the graphical interface. Running simulations this way — with probing controlled by a small Tcl script rather than by manual interaction within the waveform window — is what makes batch-mode regression practical: the same probe configuration can be reused unchanged across every corner in a PVT sweep, and the resulting set of waveform databases can be opened individually in SimVision afterwards whenever a specific corner needs closer inspection.

3.8 UVM TESTBENCH ARCHITECTURE FOR AMS VERIFICATION

The Universal Verification Methodology (UVM) provides a standardised, class-based framework in SystemVerilog for constructing modular and reusable verification environments. In the context of the automation tool developed in this thesis, understanding the UVM component hierarchy is essential because the testbench templates generated by the tool are structured according to UVM conventions.

3.8.1 UVM Component Hierarchy

A UVM-based testbench is organised as a hierarchy of components, each with a well-defined role. At the top sits the uvm_test class, which controls the overall test scenario. Below this is the uvm_env (environment), which serves as a container for all verification components. Within the environment, one or more uvm_agent instances handle the stimulus and monitoring for a given DUT interface. Each agent contains a uvm_sequencer (which dispatches sequence items to the driver), a uvm_driver (which translates sequence items into actual signal transitions on the DUT interface), and a uvm_monitor (which observes the DUT outputs and passes them to the scoreboard). The uvm_scoreboard compares observed outputs against expected values and flags mismatches.

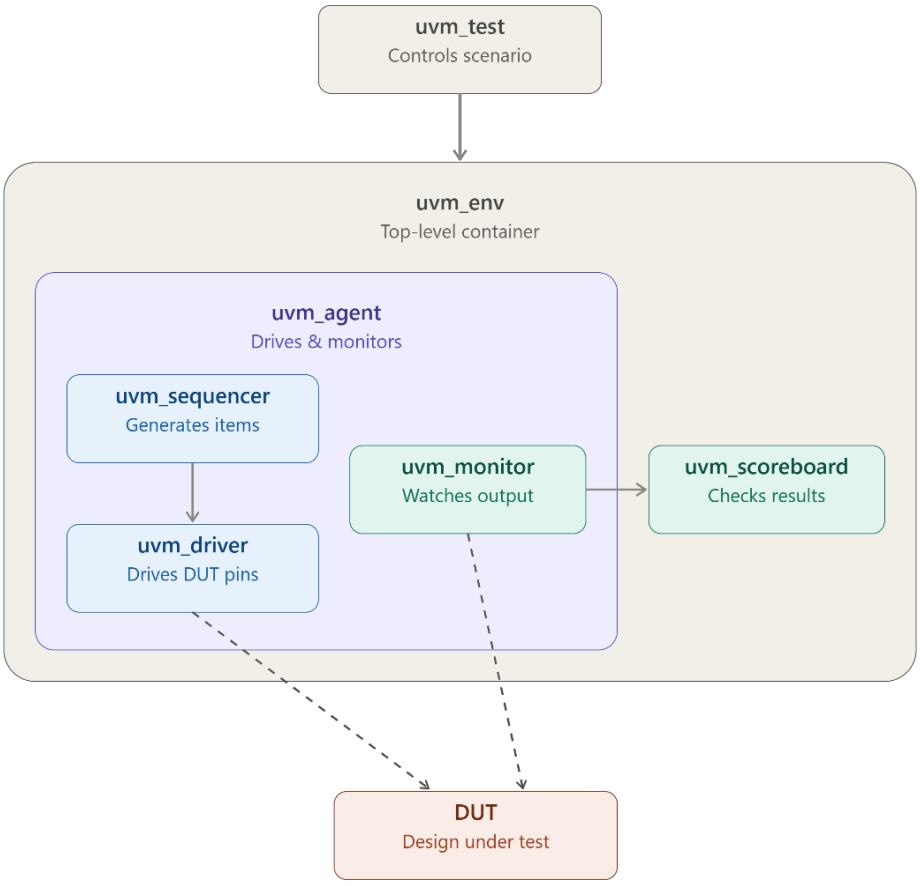


Figure 3.10 Standard UVM component hierarchy as used in AMS testbench generation

In Figure 3.10, solid arrows represent direct UVM component instantiation, while dashed arrows represent the virtual interface connection through which the driver and monitor exchange signal-level values with the DUT.

3.8.2 Sequences and Test Stimulus

The stimulus applied to the DUT is encapsulated in UVM sequence objects. A sequence is a class that extends uvm_sequence and contains a body() task that defines the stimulus pattern. Each stimulus event is represented as a uvm_sequence_item, which carries the values of one transaction — for example, a supply voltage level, a digital control word, a clock frequency, or a data payload for a protocol transaction.

For AMS designs, the test sequences must capture analog-specific stimulus patterns that go beyond simple digital toggling. Common examples include supply ramp sequences (where the supply voltage is gradually increased from zero to nominal over a defined time period), enable sequence patterns (where a digital control signal transitions after the supply has settled), and protocol-based sequences (where address and data transactions are driven according to a

specific communication protocol such as I2C or SPI). These are precisely the kinds of sequences that the AI agent component of the tool generates, using the extracted pin information from the netlist as the basis for producing contextually appropriate stimulus.

3.8.3 Real-Valued Interfaces in AMS Testbenches

One of the key adaptations required when applying UVM to mixed-signal designs is the handling of real-valued signals at the testbench-DUT interface. Standard UVM drivers and monitors operate on logic-typed virtual interfaces, but AMS designs with wreal ports require an interface that can carry floating-point values. This is achieved by declaring real or wreal typed signals in the SystemVerilog interface block and using assign statements in the driver to connect the sequence item values to these signals. The testbench template generator in the automation tool handles this distinction automatically — after identifying wreal ports in the parsed netlist, it generates the appropriate real-valued interface declarations and driver constructs for those ports, while using conventional logic declarations for digital ports.

3.9 SUMMARY

This chapter has covered the technical background that underpins the automation framework described in this thesis. The AMS verification flow was introduced as a pipeline requiring coordinated management of netlists, testbenches, AMSCF files, and simulation invocations. The structure of Verilog and SPICE netlists was examined in the context of the parsing task performed by the automation tool. Real number modelling was explained as the simulation technique adopted for this project, with the wreal net type identified as the key mechanism for carrying real-valued signals across module boundaries. The AMSCF was described in detail, highlighting the per-corner parameterisation that makes its manual generation impractical at scale. The PVT corner framework was presented, quantifying the size of the corner matrix that the tool must manage. The Cadence Xcelium AXUM simulation flow was described, covering the xrun command structure and the three-phase simulation process. Finally, the UVM testbench architecture was outlined, with particular attention to the sequence-driven stimulus model and the real-valued interface extensions needed for AMS designs.

With this background established, Chapter 4 proceeds to describe the architecture and implementation of the GUI developed as part of the automation framework, and Chapter 5 covers the core automation modules in detail.

CHAPTER 4

GUI ARCHITECTURE AND IMPLEMENTATION

4.1 INTRODUCTION

The AMS Verification Automation Framework is implemented as a Python-based desktop application with a graphical user interface built using the Tkinter library. The design philosophy behind the tool was guided by one central requirement: the framework must be immediately usable by a verification engineer who understands the AMS design flow but may not be proficient in Python scripting. Providing a structured, stage-driven GUI achieves this goal by making each step in the setup process explicit, sequential, and visually guided.

This chapter describes the architecture of the GUI, the rationale behind key design decisions, and the implementation details of each stage. The chapter covers the tool's entry point and launch mechanism, the four processing stages presented to the user, the sub-window designed for AMSCF and PVT corner configuration, and the protocol selection and management interface. The chapter concludes with a description of the VS Code-based AI agent component, which operates independently of the GUI to assist with test sequence generation.

4.2 TECHNOLOGY STACK AND DESIGN RATIONALE

Python was chosen as the implementation language for the AMS Verification Automation Framework for several practical reasons. Its standard library includes robust support for text processing, file system operations, regular expressions, and subprocess management — all of which are central to the tool's operation. Equally important is that Python is already in common use in EDA environments for scripting, log analysis, and regression management, which means the tool can be deployed on most verification workstations without installing additional dependencies.

For the graphical interface layer, the Tkinter library was selected. Tkinter is part of Python's standard library and provides a complete set of widgets for building desktop GUI applications, including windows, frames, buttons, labels, entry fields, listboxes, dropdown menus, and text areas. Its key practical advantage in a restricted industrial environment is that it requires no external installation — any machine with a working Python installation already has Tkinter available. The library also supports the creation of additional Toplevel windows, which was used to implement the dedicated AMSCF configuration sub-window described in Section 4.6.

The overall GUI is structured as a single main window that guides the user through four sequential stages. Each stage is presented as a distinct section of the window, and the user progresses through them in order — from netlist parsing at the top, through testbench generation and protocol selection, through AMSCF configuration, to simulation launch. This linear stage-based design was a deliberate choice over a tabbed interface: it reinforces the sequential nature of the verification setup workflow and reduces the risk of a user attempting to generate an AMSCF before the netlist has been parsed, or launching a simulation before the testbench has been generated.

4.3 TOOL LAUNCH AND ENTRY POINT

The AMS Verification Automation Framework is launched from the command line. The user passes the path to the design netlist file as a command-line argument when invoking the tool:

```
python  ams_verification_master.py  <path_to_netlist_file>
```

Figure 4.1 Command-line invocation of the AMS Verification Automation Framework

Figure 4.1 describes the command-line invocation used to launch the AMS Verification Automation Framework, showing how the netlist file path is passed as an argument before the GUI window opens. Upon invocation, the tool reads the netlist file path from the command-line argument and immediately proceeds to parse the file. The Tkinter main window is then initialised and opened, with the results of the initial parsing already populated in the Stage 1 display area. This design — where parsing begins before the GUI is rendered — ensures that by the time the user sees the interface, the most time-consuming part of Stage 1 is already complete and the results are ready to review.

The choice to accept the netlist as a command-line argument rather than through a file-picker dialog in the GUI was intentional. It allows the tool to be invoked from a shell script or a CI pipeline invocation, which means it can be incorporated into automated setup flows where the netlist path is known programmatically. At the same time, the GUI provides all subsequent interactions visually, balancing scriptability with usability.

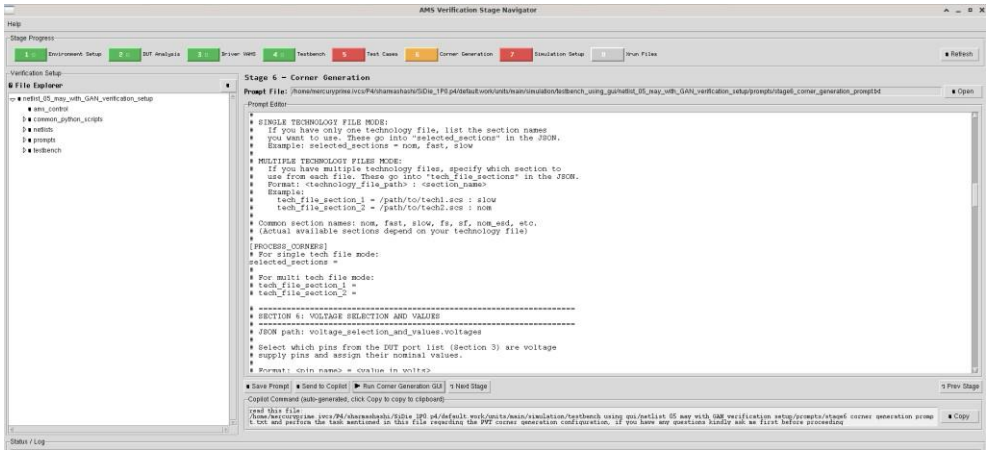


Figure 4.2 Main window of the AMS Verification Automation Framework showing all four stages

Figure 4.2 depicts the main window of the AMS Verification Automation Framework, showing all four processing stages arranged sequentially from netlist parsing through to simulation and regression launch.

4.4 STAGE 1: NETLIST PARSING AND HIERARCHY EXTRACTION

Stage 1 is the foundation of the entire framework. Its purpose is to extract structured information from the input netlist file and make that information available to all subsequent stages. The parsing module is implemented in Python using the re module for regular expression-based text processing. It handles both Verilog-format netlists (including SystemVerilog with wreal port declarations) and SPICE-format netlists.

4.4.1 Top Module Detection

Identifying the top-level module in a hierarchical netlist is not always as straightforward as it might appear. In a flat netlist with a single module declaration, the answer is obvious. In a hierarchical netlist with multiple module declarations, the top module is the one that is declared but never instantiated by any other module in the file. The parsing logic implements this principle by first building a complete list of all module names declared in the netlist, and then scanning all instantiation statements to determine which modules appear as sub-modules. The module that does not appear in any instantiation list is identified as the top-level module.

If more than one such module is found — a condition that indicates either a structurally incomplete netlist or an error in the design files — the tool reports this to the user via the terminal and halts the GUI population process. If no top module can be determined, a corresponding error message is displayed. In the nominal case, the identified top module name is displayed prominently in the Stage 1 area of the GUI.

4.4.2 Pin List Extraction

Once the top module has been identified, the parser extracts its complete port list. For each port, the parser records three attributes: the port name, the port direction (input, output, or inout), and the port data type (wire, reg, or wreal for real-valued analog ports). This port-level information is critical to the testbench template generator in Stage 2, which uses it to generate correct wire and reg declarations and to construct the DUT instantiation with proper port connections.

The extracted pin information is written to a structured text file that serves as a persistent record of the netlist interface. This file is also displayed in the GUI, giving the user a clear, readable summary of the design interface before proceeding to testbench generation. The display distinguishes between input pins, output pins, and bidirectional inout pins, as well as flagging any ports that are declared as wreal type, since these require special handling in the testbench.

4.4.3 Submodule Hierarchy and Duplicate Detection

Beyond the top module, the parser builds a complete view of the design hierarchy by tracing all module instantiations recursively. For each module found in the netlist, the parser identifies which other modules it instantiates, thereby constructing a parent-child relationship tree that reflects the full hierarchical structure of the design. This hierarchy is displayed in the GUI, giving the user a visual overview of how the design is organised before simulation setup begins.

Duplicate module detection is performed as part of this hierarchy-building process. The parser maintains a dictionary that maps each module name to the number of times it appears in the netlist. Any module that appears more than once is flagged, and the tool reports the specific module name and the number of duplicate declarations to the user through the terminal. This check is particularly important in designs where SPICE subcircuits and Verilog modules are combined in the same netlist, as naming conflicts between the two domains are a common source of elaboration errors in Xcelium.

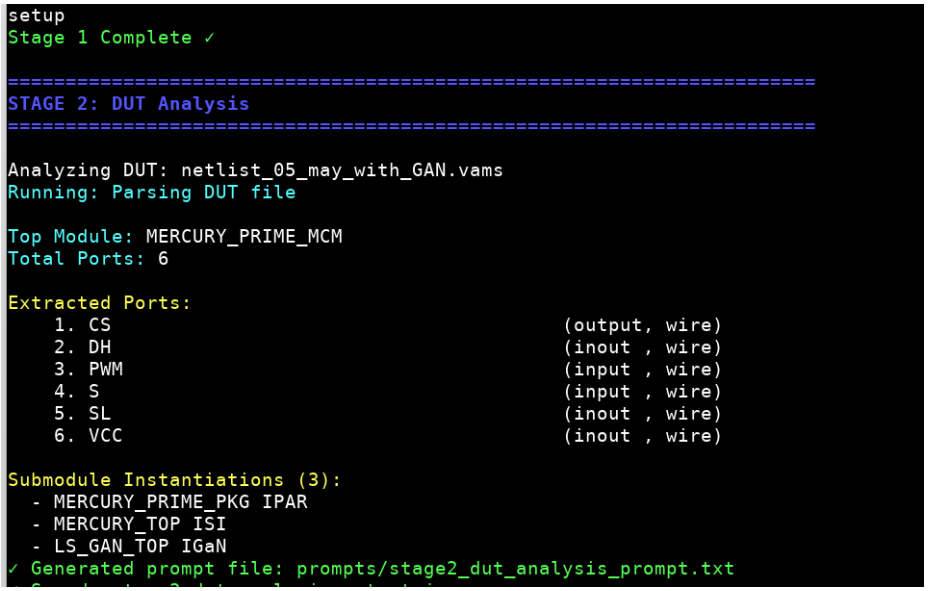


Figure 4.3 Stage 1 output displaying extracted top module name, categorised pin list, and module hierarchy

Figure 4.3 explains the Stage 1 output panel, showing the extracted top module name, the categorised input, output, and inout pin list, and the resulting submodule hierarchy displayed in the GUI.

4.5 STAGE 2: PROTOCOL SELECTION AND TESTBENCH TEMPLATE GENERATION

Stage 2 is where the parsed netlist information is transformed into a usable testbench template. Before the testbench can be generated, the user must select the communication protocol that the design uses. This selection determines how the testbench interacts with the DUT's interface and what protocol-specific modules are instantiated in the testbench.

4.5.1 Protocol Package and Selection

The framework includes a built-in protocol package that currently supports three standard communication protocols: UART, SPI, and 1-Wire. Each protocol in the package consists of a pre-written SystemVerilog module that implements the protocol's signal driving and monitoring logic, along with the corresponding port interface definition. The GUI presents these protocols to the user as a selectable list. Depending on which protocol the specific design uses, the user selects the appropriate option before triggering testbench generation.

The protocol selection is not merely cosmetic — it has a direct and concrete effect on the generated testbench template. When UART is selected, for instance, the testbench includes instantiation of the UART protocol module, with connections to the appropriate TX, RX, clock,

and enable ports on the DUT. When SPI is selected, the corresponding MOSI, MISO, SCLK, and CS connections are made instead. For 1-Wire protocol, the single bidirectional data line and pull-up logic are instantiated accordingly. The protocol package is designed to be extensible — new protocols can be added by placing the corresponding module file in the protocol directory, after which it becomes available for selection in the GUI.

4.5.2 Testbench Template Generation

With the top module information from Stage 1 and the selected protocol from the package, the testbench template generator produces a syntactically complete SystemVerilog testbench file. The generated template contains the following structural elements:

Module declaration: The testbench is declared as a module with no ports, following standard UVM testbench convention.

Wire and reg declarations: For every port extracted from the DUT, a corresponding wire or reg signal is declared in the testbench. Ports declared as wreal in the netlist receive real-typed signal declarations.

DUT instantiation: The top-level design module is instantiated with named port connections, mapping each DUT port to its corresponding testbench signal.

Protocol instantiation: The selected protocol module is instantiated with connections to the relevant DUT ports and testbench signals.

Clock and reset generation: Standard initial blocks for clock generation and reset sequencing are included.

Test sequence placeholder: A clearly marked section is included where the AI agent or the engineer will insert the specific test stimulus sequences.

The generated testbench template is saved to the testbench directory created by the tool. This file serves as the starting point for the AI agent in the VS Code environment, which reads it to understand the design interface and the protocol in use before generating the test sequences.

4.5.3 Directory Structure Creation

As part of Stage 2, the framework automatically creates the directory structure required for the simulation run. This includes a dedicated directory for AMSCF files, a directory for the protocol source files, and a directory for testbench files. Creating this structure programmatically eliminates the manual directory management step that verification engineers

typically perform before setting up a new simulation, and ensures that the AMSCF generation stage in Stage 3 has a well-defined location to write its output files.

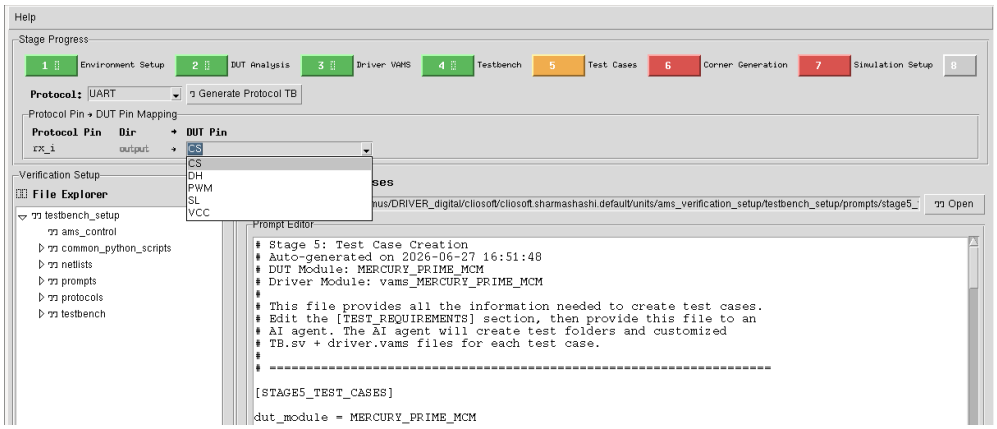


Figure 4.4 Stage 2 showing the protocol selection panel (UART / SPI / 1-Wire) and the generated testbench template preview

Figure 4.4 depicts Stage 2 of the GUI, showing the protocol selection panel for UART, SPI, and One-Wire alongside a live preview of the generated testbench template.

4.6 STAGE 3: AMSCF AND PVT CORNER CONFIGURATION

Stage 3 handles the generation of AMSCF files for the selected PVT corners. Given the multi-dimensional nature of PVT corner management — where process, voltage, and temperature parameters must be independently combinable — a dedicated sub-window was developed for this stage rather than embedding it in the main window. This sub-window is opened by clicking a button in the Stage 3 area of the main GUI.

4.6.1 AMSCF Configuration Sub-Window

The AMSCF configuration sub-window is organised into three vertical columns, each corresponding to one dimension of the PVT corner matrix. The first column handles process corner selection, presenting the standard process corners (TT, SS, FF, FS, SF) as individually selectable checkboxes. The user checks the corners that need to be covered in the verification run. The second column handles voltage levels, where the user enters the supply voltage values to be included in the corner matrix. The third column handles temperature, where the user enters the temperature values to be used.

In addition to the PVT parameter columns, the sub-window provides input fields for the technology model file paths. The user enters the paths to the process-corner-specific model library files provided by the PDK — for example, the path to the TT corner Spectre model file, the SS corner model file, and so on. The top module pin list, already extracted in Stage 1, is pulled automatically into the AMSCF configuration and used to configure the portmap directives in the generated files, so the user does not need to re-enter this information.

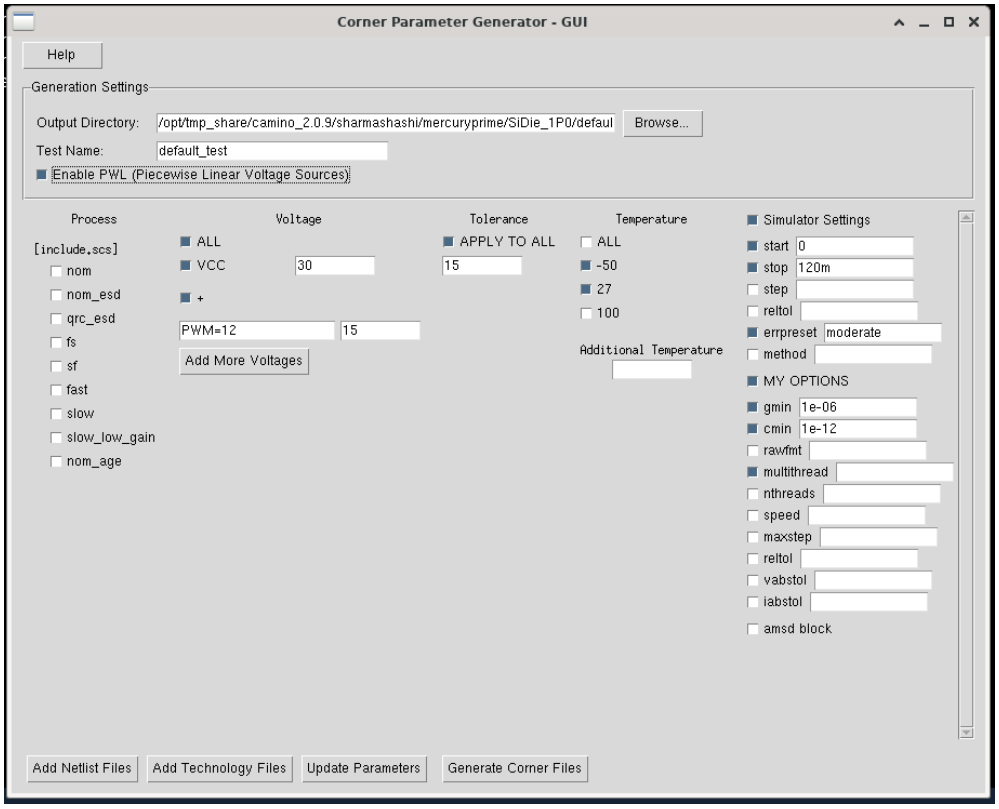


Figure 4.5 AMSCF configuration sub-window showing process corner selection, voltage input, and temperature input columns

Figure 4.5 describes the AMSCF configuration sub-window, showing the three-column layout used to select process corners, enter voltage levels, and specify temperature points for corner-file generation.

4.6.2 Multi-Corner AMSCF File Generation

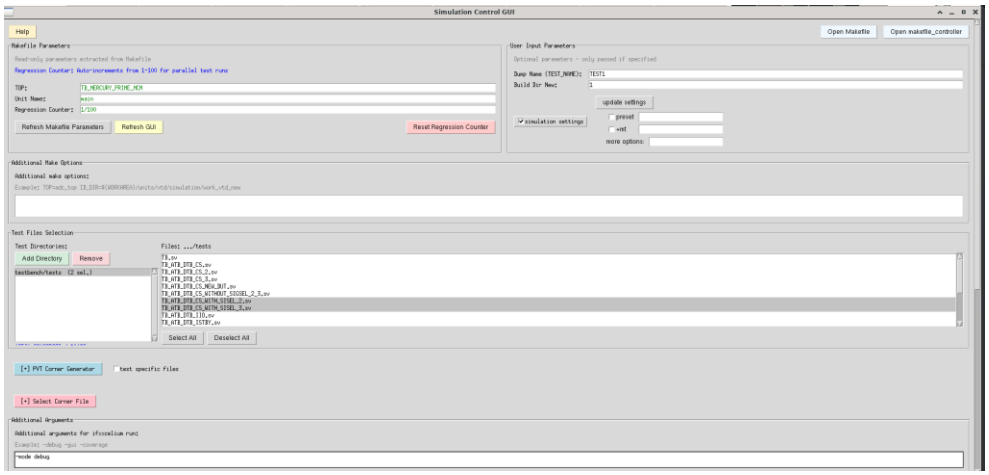
When the user confirms the PVT configuration and clicks the generate button, the framework computes the full cross-product of selected process corners, voltage levels, and temperatures. For each combination, it generates a complete AMSCF file following the Spectre format described in Chapter 3. Each file is named according to a consistent naming convention that encodes the corner parameters, for example amscf_tt_1v8_27c.scs for a TT corner simulation at 1.8 V and 27°C. All generated files are saved to the AMSCF directory created in Stage 2.

The content of each AMSCF file is constructed by the generator using the technology file path, the supply voltage, and the temperature for that specific corner, combined with the portmap and connect rule configuration derived from the top module pin information. This automated generation eliminates what would otherwise be a substantial manual effort — for a configuration with 5 process corners, 3 voltage levels, and 3 temperatures, the tool generates 45 correctly configured AMSCF files in the time it takes a user to click a button.

4.7 STAGE 4: SIMULATION AND REGRESSION LAUNCH

Stage 4 provides the interface through which the user launches simulations using the generated testbench files and AMSCF files. By this point in the workflow, the output of the preceding stages is available in the tool's working directories: one or more testbench files (including the template and any agent-generated test sequence variants) and the set of AMSCF files corresponding to the selected PVT corners.

The regression launch module constructs xrun invocation commands for each simulation in the regression matrix. Each command references the appropriate testbench file, the corresponding AMSCF file, the netlist source files, and the simulation log file path. The framework supports both sequential and parallel execution modes. In sequential mode, simulations are run one at a time in the order they are queued, which is appropriate for environments where compute resources are limited. In parallel mode, multiple xrun processes are launched simultaneously using Python's subprocess module with non-blocking calls, which significantly reduces the total wall-clock time for a large regression. The cross-combination of testbench files and AMSCF files forms the full regression matrix — for example, three testbench files combined with 45 AMSCF corner configurations yields 135 simulation runs that the framework manages without manual intervention.



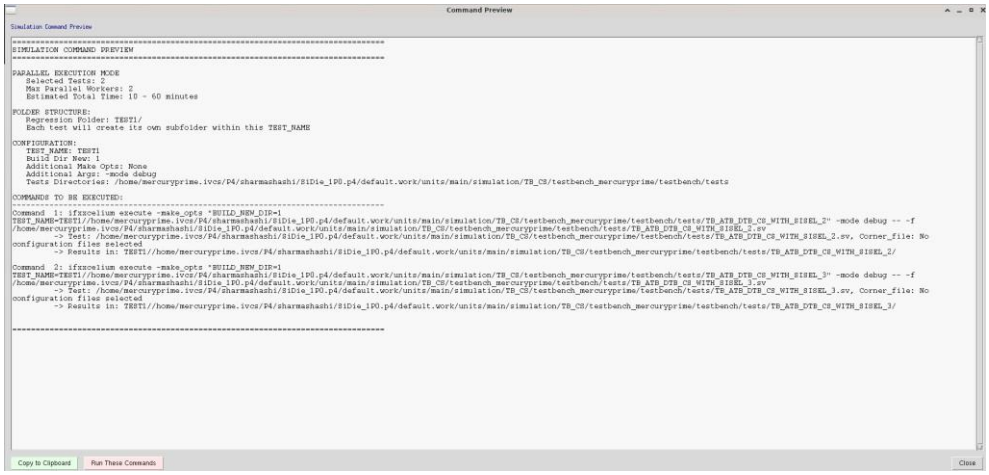


Figure 4.6 Stage 4 regression launch panel showing testbench file selection, AMSCF file association, and execution mode

Figure 4.6 explains the Stage 4 regression launch panel, showing how generated testbench files are associated with AMSCF corner files before sequential or parallel simulation execution begins.

4.8 AI-ASSISTED TEST SEQUENCE GENERATION VIA VS CODE AGENT

The AI agent component of the AMS Verification Automation Framework is developed and deployed as an extension within the Visual Studio Code (VS Code) editor environment. It operates independently of the Tkinter GUI described in the preceding sections, and is accessed through the built-in chat interface that VS Code provides for extension-based AI agents. The backend language model powering the agent is Claude Sonnet 4.6, developed by Anthropic, accessed via API from the extension.

4.8.1 Agent Architecture and Context Awareness

The agent's most important capability is its awareness of the testbench file context. When a user opens a testbench file in VS Code and interacts with the agent, the agent reads the current contents of the open file before generating any response. This reading step allows the agent to determine which protocol is instantiated in the testbench — whether UART, SPI, or 1-Wire — and to tailor its generated test sequences accordingly. This context awareness is what distinguishes the agent from a generic code generation tool: rather than producing abstract or generic stimulus code, it produces sequences that are correctly structured for the specific protocol and port interface of the design under test.

The agent also processes the DUT instantiation and port declarations visible in the testbench file, which allows it to use correct signal names in the generated sequences. This means a user

does not need to manually specify port names in the prompt — the agent infers them from the file content.

4.8.2 Prompt-Driven Test Sequence Generation

The user interacts with the agent by typing a natural language prompt in the VS Code chat window. The prompt describes the desired stimulus in functional terms rather than in HDL syntax. For example, a user might write a prompt such as: “Ramp the supply voltage from 0 V to 1.8 V over 100 ns, then assert the enable signal and send address 0x04 with data 0xFF over SPI.” The agent interprets this description, resolves it against the protocol and port context read from the testbench file, and generates the corresponding SystemVerilog test sequence code — including the timing delays, signal assignments, protocol transaction calls with address and data values, and any necessary wait statements.

The generated test sequence is written directly into the testbench file at the designated test sequence placeholder section. The output of the agent’s reasoning and the generated code are both visible in the VS Code chat window, allowing the user to review the agent’s interpretation before the code is committed to the file. If the generated sequence is not quite right — for instance, if the ramp rate or the address value needs adjustment — the user can provide a follow-up prompt to refine it, and the agent updates the sequence accordingly.

4.8.3 Supported Stimulus Patterns

The agent is capable of generating several categories of test stimulus that are commonly required in AMS verification. Supply ramp sequences, where a voltage signal is gradually incremented from one level to another at a specified rate, are among the most frequently used in analog block verification. Protocol transaction sequences, where address and data values are transmitted according to the timing and framing rules of UART, SPI, or 1-Wire, represent another major category. Enable and disable sequences, where a digital control signal is toggled after a supply or clock has settled, are also commonly generated. For more complex stimulus requirements, the agent can combine multiple categories into a single sequence based on a multi-part prompt.

The use of Claude Sonnet 4.6 as the backend model provides the agent with the language comprehension needed to handle ambiguous or informally worded prompts — a practical consideration given that verification engineers often describe stimulus requirements in shorthand or domain-specific informal language rather than in precise technical specifications. The domain context provided through the testbench file content compensates for the fact that

the model is not specifically fine-tuned on hardware verification data, by giving it concrete and specific grounding for each generation task.

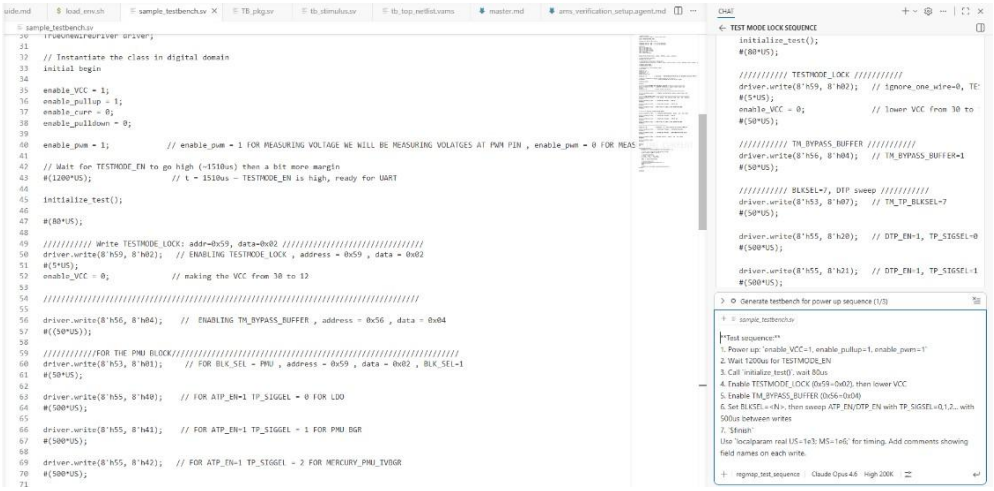


Figure 4.7 VS Code agent chat interface showing a user prompt and the corresponding generated SPI test sequence

Figure 4.7 depicts the VS Code agent's chat interface, showing a natural-language prompt from the user alongside the corresponding SPI test sequence generated by Claude Sonnet 4.6.

4.9 ERROR HANDLING AND DIAGNOSTIC REPORTING

Error handling in the AMS Verification Automation Framework follows a terminal-based reporting model. When the tool encounters a condition that prevents it from proceeding correctly, it prints a descriptive error message to the terminal from which the tool was launched, and either halts the affected stage or flags the issue without blocking the user from continuing with other parts of the workflow.

The most common error condition encountered during netlist parsing is the failure to identify a unique top-level module. This occurs either when the netlist contains no module declarations — indicating a malformed or incomplete input file — or when multiple modules satisfy the top-module detection criteria, suggesting that the file contains parallel top-level modules rather than a proper hierarchy. In both cases, the tool prints a specific message to the terminal identifying the nature of the problem and, where possible, naming the conflicting modules.

Duplicate module declarations are reported as a separate class of warning. If the same module name appears more than once in the netlist, the tool reports the module name and the count of duplicate appearances. This is treated as a warning rather than a hard error, because some simulation environments tolerate duplicate subcircuit definitions with a redefinition override; the user can choose whether to address the duplication or proceed. Incorrect or unrecognised

port declarations — for example, ports with unresolvable direction keywords or malformed type annotations — are reported with the specific line or construct that triggered the issue, helping the user locate and correct the problem in the source file.

The decision to surface errors through the terminal rather than through GUI popup dialogs was deliberate. In a professional verification environment, terminal output is easily captured and logged as part of a larger automated flow, whereas dialog boxes require manual dismissal and cannot be redirected to a log file. The terminal-based approach also aligns with the expectations of engineers accustomed to working with command-line EDA tools, where structured message output is the norm.

4.10 SUMMARY

This chapter has described the architecture and implementation of the AMS Verification Automation Framework in detail. The tool is implemented in Python using Tkinter as the GUI framework, and is launched from the command line with the netlist file as an argument. The main window presents the user with four sequential stages: netlist parsing and hierarchy extraction, protocol selection and testbench template generation, AMSCF and PVT corner configuration, and simulation and regression launch. A dedicated sub-window handles the three-dimensional PVT corner configuration, allowing the user to select process, voltage, and temperature parameters and automatically generate the full set of corresponding AMSCF files. A separate VS Code-based AI agent, powered by Claude Sonnet 4.6, complements the GUI by generating test sequences from natural language prompts, with awareness of the protocol and port context in the testbench file. Error conditions are reported through the terminal in a structured, loggable format.

Chapter 5 follows with a detailed discussion of the core automation modules — the netlist parser, the testbench template engine, and the AMSCF generator — describing the algorithms, data structures, and implementation choices that underpin the functionality introduced in this chapter.

CHAPTER 5

TESTBENCH TEMPLATE AND AMSCF AUTOMATION

5.1 INTRODUCTION

Chapter 4 described the GUI architecture and the user-facing workflow of the AMS Verification Automation Framework — what the engineer sees and does at each stage. This chapter goes one level deeper, examining the algorithms, data structures, and implementation logic that make each of those stages work. The focus here is on the five core automation modules: the netlist parsing engine, the Verilog-AMS driver generator, the SystemVerilog testbench template generator, the AMSCF and PVT corner file generator, and the regression script generator. The chapter concludes with a discussion of the AI-assisted test sequence generation component, which extends the framework's capabilities beyond structured file generation into intelligent, prompt-driven stimulus creation.

All modules are implemented in Python 3, using only the standard library. This was a deliberate design choice — no third-party parsing libraries such as PLY, pyparsing, or ANTLR were used. The rationale was straightforward: avoiding external dependencies ensures the tool can be deployed on any verification workstation with a standard Python installation, without requiring package management steps that may be restricted in an industrial environment.

5.2 NETLIST PARSING ENGINE

The netlist parsing engine is the entry point of the entire framework. It reads the design netlist — which may be in Verilog, SystemVerilog, or Verilog-AMS format — and produces the structured information that all subsequent stages depend on: the top module name, the complete port list with direction and type annotations, and the full submodule hierarchy.

5.2.1 Two-Pass Streaming Architecture

Large analog netlists can be substantial in size, and reading the entire file into memory at once is not always practical. The parser therefore operates using a streaming chunk-based approach, processing the file in 1 MB segments. This allows it to handle very large netlists without memory constraints.

The parsing proceeds in two passes over the file. In the first pass, all module definition statements are collected. The parser applies a regular expression pattern of the form shown

below to each processed chunk, accumulating a set of all module names defined anywhere in the file:

```
MODULE_DEF_RE = re.compile(r'\bmodule\s+(\w+)', re.IGNORECASE)
```

Figure 5.1 Regular expression used in Pass 1 to collect module definition names

Figure 5.1 describes the regular expression used in the parser's first pass to collect every module definition name appearing in the input netlist file. Before applying any regex patterns, each chunk is pre-processed to strip block comments, line comments, and attribute blocks. This pre-processing step is critical — without it, module names appearing inside comments or attribute blocks would be incorrectly collected as defined modules. A set of known Verilog gate primitives (and, or, not, buf, xor, and others) and common PDK library component names are maintained in an ignore list; any name matching these is excluded from the defined module set regardless of how it appears in the file.

5.2.2 Top Module Detection Algorithm

The second pass over the file tracks which module is currently being parsed and collects all module instantiations found within it. An instantiation is identified by the pattern of a module name followed by an optional parameter block, an instance name, and a port connection list terminated by a semicolon. The parser builds a dictionary mapping each parent module to a list of its child module instantiations.

Once both passes are complete, the top module is identified using a set subtraction operation. The set of all instantiated child modules is subtracted from the set of all defined modules:

```
instantiated  = {child for children in module_instances.values()
                       for child, _ in children}
top_candidates = defined_modules - instantiated
```

Figure 5.2 Set subtraction logic for top module identification

Figure 5.2 explains the set subtraction logic used to identify the top module, where every instantiated child module is removed from the full set of defined modules. Any module that is defined but never appears as a child in any other module's instantiation list is a top-level candidate. In most well-formed netlists there is exactly one such module, and that is taken as the top. When multiple candidates are found — which may occur if the netlist contains parallel top-level modules or if some modules have no instantiations at all — the framework selects the candidate with the greatest number of direct child instantiations, on the basis that the true top

module is likely the one with the richest hierarchy beneath it. If no top module can be identified, an error is reported to the terminal.

5.2.3 Port Extraction: ANSI and Non-ANSI Styles

Port extraction operates only on the identified top module. The parser first locates the module header in the file — the segment between the module keyword and the first semicolon following the port list — and then determines whether the netlist uses ANSI-style or non-ANSI-style port declarations.

ANSI style declares port directions and types inline within the module header. Detection is straightforward: if direction keywords (input, output, inout) appear within the port list string, the file is using ANSI style. Non-ANSI style lists only port names in the module header and places the direction and type declarations in the module body. Both styles are common in practice — older Verilog-AMS netlists often use non-ANSI format, while newer SystemVerilog files typically use ANSI.

For each port, the parser extracts three attributes: the direction (input, output, or inout), the data type (electrical, wreal, real, wire, logic, or reg), and the bit width if applicable. The wreal and electrical types are particularly significant in AMS designs — wreal ports require real-valued signal declarations in the testbench rather than logic types, and electrical ports are used in the Verilog-AMS driver. The comma-splitting logic used to separate port entries is depth-aware, correctly handling commas that appear inside bracket expressions such as [N:M] width specifiers.

5.2.4 Hierarchy Tree and Duplicate Detection

The module instantiation dictionary collected in Pass 2 is used to build the full submodule hierarchy tree. A recursive depth-first traversal starting from the top module produces a tree that shows every module and its children at each level, including instance names. Circular references are detected and flagged during the traversal to prevent infinite recursion.

Duplicate module detection compares the total count of module definitions against the unique set. Any module name appearing more than once in the defined set is reported as a duplicate, with the line boundaries of each occurrence recorded. The framework can then comment out subsequent duplicate definitions automatically — wrapping them in block comments with a timestamp annotation — leaving the first occurrence intact for simulation. Both the hierarchy tree and the duplicate detection report are written to a structured text file and displayed in the GUI.

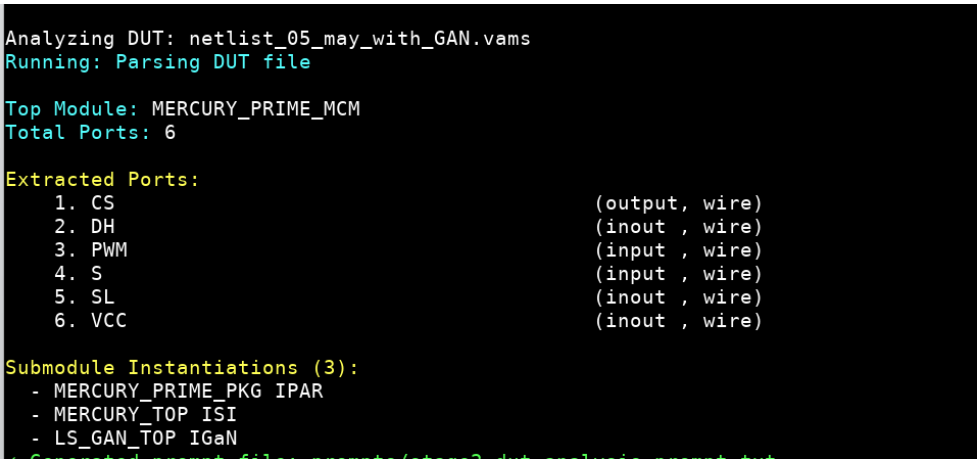


Figure 5.3 Sample output of the netlist analysis report for a mixed-signal DUT

Figure 5.3 depicts a sample netlist analysis report generated for a mixed-signal DUT, showing the extracted top module, categorised pin list, and hierarchy summary.

5.3 VERILOG-AMS DRIVER GENERATION

The Verilog-AMS driver is an essential bridge component in the AMS simulation flow. It connects the digital SystemVerilog testbench to the analog DUT ports, converting digital stimulus values into analog voltage sources through Verilog-AMS electrical constructs. The driver generation module produces a syntactically complete skeleton driver file from the port list extracted in Stage 2, which the engineer or AI agent then completes with actual stimulus values.

5.3.1 Port Direction Reversal Logic

The most conceptually important aspect of driver generation is the reversal of port directions. From the DUT's perspective, an input port receives a signal from outside. From the driver's perspective, that same signal must be driven outward — so the driver's corresponding port is declared as output. Conversely, DUT output ports are observed by the driver, so they are declared as input in the driver. Inout ports retain the inout direction in both. This systematic reversal is applied to every port in the extracted list:

```
input_ports  = [p for p in ports if p['direction'] == 'input']
output_ports = [p for p in ports if p['direction'] == 'output']
inout_ports  = [p for p in ports if p['direction'] == 'inout']
```

Figure 5.4 Port direction categorisation for driver generation

Figure 5.4 describes how DUT port directions are categorised and reversed during driver generation, turning every DUT input into a driver output and vice versa.

5.3.2 Internal Real Variables and Analog Block Construction

For every DUT input port, the driver generates three elements in the output file. First, an electrical port declaration is emitted, which gives the port its Verilog-AMS discipline and enables the use of V() access functions in the analog block. Second, a real internal variable is declared with the suffix _int appended to the port name. This real variable serves as the interface between the digital initial block (which uses procedural assignments) and the analog block (which uses continuous V() assignments). Third, the initial block initialises this variable to zero at time zero.

The analog block of the generated driver is left empty as a skeleton, with a comment indicating where the engineer must add the V() voltage source statements for each port. A ground reference is declared and grounded using the Verilog-AMS ground construct, which the V() sources reference for their return path. This skeleton structure is illustrated in the representative excerpt below:

```
real vin_int;          // internal variable for DUT input port 'vin'
initial begin
    vin_int = 0;       // initialised to zero at time-zero
end

analog begin
    // Add V() assignments here — e.g.: V(vin, gnd) <+
transition(vin_int, 0, 5n);
end
```

Figure 5.5 Generated skeleton structure for a DUT input port in the Verilog-AMS driver

Figure 5.5 depicts the generated skeleton structure for a single DUT input port in the Verilog-AMS driver, showing the internal real variable and empty analog block. The driver file is constructed entirely through string building — each line is appended to a Python list, and the list is joined with newline characters at the end. No external template engine is used. This approach gives the generation logic precise, line-level control over the output structure, which is important for producing files that conform to the strict formatting requirements of the Verilog-AMS compiler.

5.4 TESTBENCH TEMPLATE GENERATION

The SystemVerilog testbench template generator takes the port list from Stage 2 and the driver information from Stage 3 as its primary inputs, and produces a complete structural skeleton of the testbench file. At this stage, the testbench contains all the wiring and instantiation

infrastructure but has no test stimulus — that is added later either by the engineer manually or by the AI agent described in Section 5.8.

5.4.1 Wire Declarations and Port Connections

For each port in the DUT's interface, the generator declares a corresponding signal in the testbench. The declaration type is determined by the port's data type as extracted from the netlist: logic-typed ports receive wire declarations, real-valued ports (wreal or real) receive real declarations. This distinction is important because wreal nets require a different connection mechanism in the testbench than logic nets — connecting a wreal driver to a logic wire would produce a type mismatch error at elaboration time in Cadence Xcelium.

The DUT instantiation is generated using named port connections rather than positional connections. This is a deliberate choice — named connections are less fragile when the DUT port list changes, since they do not depend on the order of ports in the declaration. The generator iterates over the extracted port list and constructs one .portname(signalname) entry for each port, assembling them into a complete instantiation block. The same named connection pattern is used for the driver instantiation immediately below.

5.4.2 Protocol-Aware Testbench Injection

When the user selects a protocol from the package in Stage 2, the testbench generator modifies its output to include protocol-specific content. This injection is controlled by a protocol parameter passed to the generation function. Three protocols are supported: UART, SPI (referred to as I2C in the package), and One-Wire. Each protocol requires a different set of additions to the generated testbench file.

At the top of the file, a package import statement is added to bring the protocol type enumeration into scope. A localparam declaration selects the specific protocol and sets its timing parameters — for UART these are pulse width values for logical 1 and 0 bits; for One-Wire these are baud period, address width, and data width. Additional wire declarations for the protocol's interface signals are added to the signal declaration block. The protocol master module is then instantiated with the selected protocol's ports connected and unused protocol ports tied to constants or left unconnected. An illustrative excerpt of the UART protocol instantiation is shown below:

```
localparam real UART_LONGER_PULSE_VAL  = 1320; // ns — bit=1 pulse
width
localparam real UART_SHORTER_PULSE_VAL =  240; // ns — bit=0 pulse
width
```

```
protocol_master #(
  .PROTOCOL(PROTO_UART),
  .UART_LONGER_PULSE_VAL (UART_LONGER_PULSE_VAL),
  .UART_SHORTER_PULSE_VAL(UART_SHORTER_PULSE_VAL)
) PROTO (
  .data_int(data_int), .addr_int(addr_int),
  .rw(rw),             .rx_i(rx_i),
  .sda_in(1'b1),       .scl_out(),  // unused ports tied off
  .sda_out(),          .ow_io()
);
```

Figure 5.6 Protocol master instantiation block generated for UART protocol selection

Figure 5.6 explains the protocol master instantiation block generated when UART is selected, showing the timing localparams and port connections inserted into the testbench. The unused ports of the protocol master module are deliberately tied to constants or left unconnected rather than left out, because the Xcelium elaborator requires all module ports to be accounted for. Leaving a port absent from the instantiation would produce a compilation warning or error depending on the tool version.

5.5 PROTOCOL PACKAGE INTEGRATION

The protocol package is a reusable library of SystemVerilog modules providing stimulus-generation capability for the three supported communication protocols. It was designed to be project-independent — the same package files are copied into every new verification project during Stage 1 of the framework, so no manual protocol setup is required for each new design.

5.5.1 Package Architecture

The package uses a split-file architecture separating the purely digital SystemVerilog components from the Verilog-AMS components. The central file is protocol_master.sv, which defines a package named protocol_pkg containing an enumeration type protocol_t with values PROTO_UART, PROTO_I2C, and PROTO_ONE_WIRE. The protocol master wrapper module accepts a PROTOCOL parameter of this enumeration type and uses a generate block to conditionally instantiate only the sub-module corresponding to the selected protocol. This approach means only one parameter change is needed in the testbench to switch the entire protocol in use.

Each sub-protocol is implemented as a self-contained SystemVerilog module: uart_master.sv provides UART bit-level stimulus generation; i2c_master.sv provides I2C start/stop/data sequence generation; and the One-Wire protocol is implemented across three files — a package definition, an interface definition, and a driver module — owing to its more complex handshake requirements.

5.5.2 Pin Mapping and Always Block Generation

Since the protocol's output signals must be connected to specific DUT input pins, and the correct mapping varies between designs, the framework provides a visual pin mapping interface in the Stage Navigator GUI. This interface presents all output and bidirectional pins of the selected protocol in one column and all 47 DUT pins extracted in Stage 2 in a combobox dropdown in the adjacent column. The engineer maps each protocol pin to its corresponding DUT pin through this interface.

On confirming the mapping, the framework generates SystemVerilog always blocks that bridge the protocol module's outputs to the driver's internal real variables. This bridge is necessary because the protocol module outputs are logic-typed wire signals, while the Verilog-AMS driver uses real _int variables in its initial block. The generated always block pattern takes the following form:

```
always @(rx_i) begin
    DRIVER.ds_reset_n_i_int = rx_i;
end
```

Figure 5.7 Generated always block bridging protocol output to driver internal variable

Figure 5.7 depicts a generated always block that bridges a protocol module's logic output to the corresponding internal real variable used by the Verilog-AMS driver. Each protocol output pin gets one always block of this form, with the right-hand side referencing the DRIVER module's corresponding _int variable using hierarchical path notation. The mappings are persisted to a JSON configuration file so they are not lost between sessions.

5.6 AMSCF AND PVT CORNER FILE GENERATION

The AMSCF and PVT corner file generation module is arguably the most impactful part of the framework from a practical verification perspective. Its role is to transform a user's PVT corner selection into a complete set of ready-to-use Spectre-format corner files, one per corner combination, without any manual file writing.

5.6.1 Technology File Parsing

The process begins with parsing of the technology model files provided by the PDK. These files are in Spectre format and contain named sections — commonly named nom, slow, fast, ultra_slow, ultra_fast, and similar — each of which represents a different set of process parameter values. The parsing function uses a regular expression to identify section header lines in the technology file and collects all unique section names found:

```
def parse_sections(*filepaths):
    sections = []
    for fp in filepaths:
        with open(fp) as f:
            for line in f:
                m = re.match(r'^section\s+(\w+)', line.strip())
                if m and m.group(1) not in sections:
                    sections.append(m.group(1))
    return sections
```

Figure 5.8 Process section discovery from technology files

Figure 5.8 describes how process sections are automatically discovered from technology model files, showing the regular expression used to collect unique section names. A second parsing function scans the technology file for voltage supply definitions, extracting the names of all voltage sources used in the design. These supply names are presented in the GUI as voltage configuration rows, pre-populated with the default values found in the technology file. The user can then modify the values, enable tolerance-based sweeps, or add manual voltage entries for supplies not present in the technology file.

5.6.2 PVT Cross-Combination Engine

With the process sections, voltage values, and temperatures all selected in the GUI, the cross-combination engine computes the full set of corner configurations that must be simulated. For each unique combination of one process section, one voltage configuration, and one temperature, a separate corner file is required. The engine computes the Cartesian product of these three sets:

```
combinations = []
for section in selected_sections:
    for voltage_combo in voltage_sweep_points:
        for temperature in selected_temperatures:
            combinations.append((section, voltage_combo,
temperature))
```

Figure 5.9 Cartesian product logic for PVT corner combination generation

Figure 5.9 explains the Cartesian product logic used to generate every PVT corner combination from the selected process sections, voltage points, and temperatures. The voltage sweep points are computed from the nominal voltage values and tolerance specifications entered by the user. If a voltage supply named vin has a nominal value of 5.0 V and a tolerance of ±10% is specified, the sweep generates 4.5 V, 5.0 V, and 5.5 V as separate voltage points for that supply. All selected voltage supplies are varied together in each combination, producing a compact but complete voltage configuration string for each corner.

5.6.3 Corner File Content and Naming Convention

Each corner file is a Spectre-format text file whose content is constructed line by line. The file begins with an include statement pointing to the technology model file and specifying the process section for that corner. Voltage source definitions follow, one per supply, using the Spectre vsource component syntax with the dc parameter set to the nominal voltage value for that corner combination. The temperature is set using a Spectre .option statement. If an AMSD block has been configured — specifying which modules are analog and how connect rules should be applied — that block is appended at the end of the file.

The naming convention for generated corner files encodes the full PVT configuration in the filename, making it immediately readable without opening the file. The pattern is section_voltage-string_temperature.scs, where the voltage string concatenates the supply name and value for each configured supply. For example, a nominal process corner with vin at 5.0 V, vip at 6.5 V, and a temperature of 27°C produces the filename nom_vin5.0_vip6.5_27.scs. All generated files are written to the spicemodels/corners/ directory created during Stage 1. For a typical selection of 5 process sections, 3 voltage sweep points, and 3 temperatures, the engine generates 45 corner files.

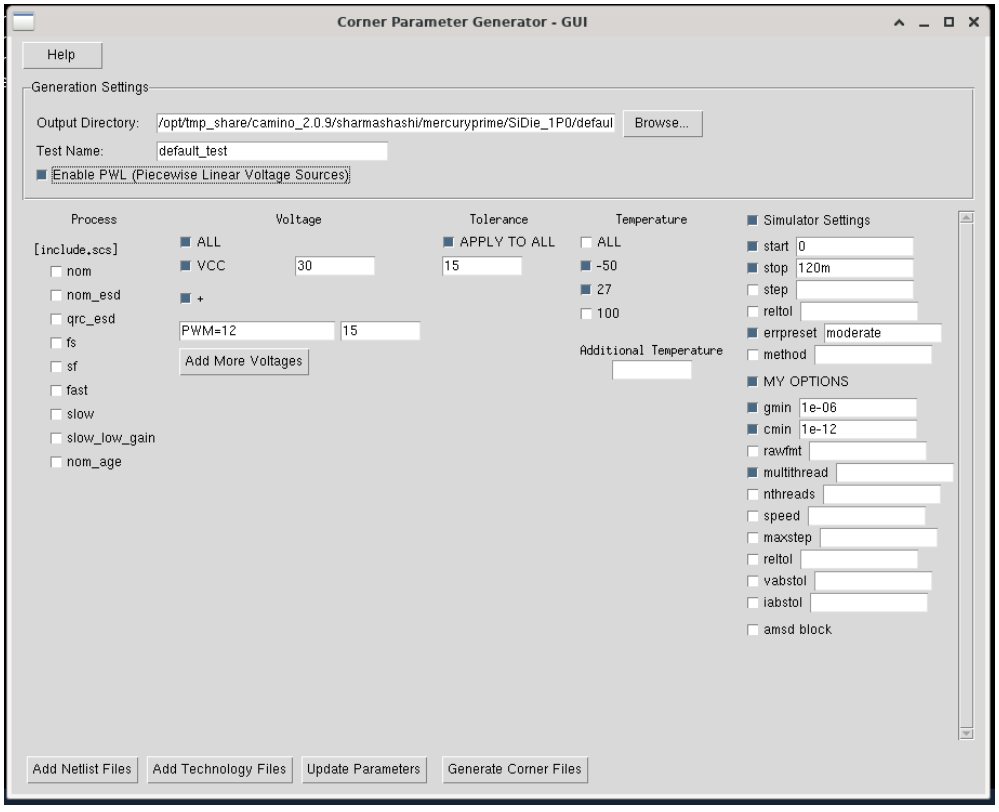


Figure 5.10 Corner Parameter GUI with 3-column PVT selection layout

Figure 5.10 depicts the Corner Parameter GUI, showing its three-column layout for process corner checkboxes, voltage tolerance entries, and temperature selection fields.

5.7 REGRESSION SCRIPT GENERATION

The regression script generator is the final automated stage before simulation execution. Its function is to take the set of testbench files and corner files selected by the user in the Simulation Control GUI and compute all required simulation commands, writing them into a shell script that can be executed on the distributed compute cluster without further manual intervention.

5.7.1 Test-Corner Cross-Combination Engine

The Makefile Controller module first parses the project Makefile to extract key build variables such as OUT_PREFIX, which specifies the base directory where simulation outputs are stored. This parsing is done using regular expression matching against Makefile variable assignment syntax, and handles both simple assignments and assignments that involve environment variable expansion. The extracted OUT_PREFIX is used to construct the TEST_NAME path for each simulation run, which determines where that simulation's output directory will be created.

The cross-combination engine then computes the complete T × C matrix, where T is the number of selected testbench test directories and C is the number of selected corner files. For each (test, corner) pair, one simulation command is generated. If compile-time define parameters have been specified, these are added as a third dimension of the combination, expanding the total number of commands to T × C × D.

5.7.2 run.sh Construction

Each simulation command is formatted as a Cadence Xcelium invocation that passes the test filelist wrapper and the corner configuration file as arguments. The TEST_NAME argument encodes both the test and corner identifiers in a hierarchical path format, so that each simulation's output is stored in a unique, identifiable subdirectory. A configurable sleep interval — set to 200 seconds by default — is inserted between consecutive commands. This sleep prevents the simultaneous submission of too many jobs to the distributed compute cluster, which could cause scheduling failures or farm overload.

Progress echo statements are added before and after each command to make the regression execution traceable when run as an overnight batch job. If a run.sh file already exists in the target directory, the framework presents a confirmation dialog before overwriting it, preventing accidental loss of a previously generated regression script. Once generated, the run.sh is automatically copied from the working directory to the simulation id-directory where Xcelium

expects to find it, using the PathDiscovery class to resolve the correct destination path without any hardcoded path assumptions.

5.7.3 Sequential and Parallel Execution Modes

The framework supports two execution modes. In sequential mode, the run.sh commands are executed one at a time, in the order they were generated. This mode is appropriate when compute resources are constrained or when debugging individual simulation failures. In parallel mode, multiple simulation processes are launched simultaneously using Python's subprocess module with non-blocking Popen calls. Each process runs independently, and the framework collects their return codes after all have completed. Parallel mode is the preferred mode for large regressions, as it can reduce the total wall-clock time proportionally to the number of available compute nodes.

5.8 AI-ASSISTED TEST SEQUENCE GENERATION

The test sequence generation component represents the most novel aspect of the AMS Verification Automation Framework. While all other modules produce structural or configuration files whose content follows deterministic patterns, test sequences are fundamentally non-deterministic — their content depends on the specific verification intent, the design's functional requirements, and the engineer's understanding of what stimulus is needed to exercise each design condition. This is the domain where a large language model agent becomes valuable.

5.8.1 VS Code Agent Architecture

The AI agent is implemented as a Visual Studio Code extension that operates through the VS Code chat interface. It uses Claude Sonnet 4.6 (Anthropic) as its backend language model, accessed via the Anthropic API. The agent is not part of the Tkinter GUI — it runs as a separate, independently accessible component that the engineer uses after the testbench template has been generated by the scripted stages.

The agent's most important architectural feature is its context-reading capability. Before generating any output, the agent reads the currently open testbench file in the VS Code editor. This reading step serves two purposes. First, it identifies which protocol is instantiated in the testbench by scanning the localparam declarations near the top of the file — the presence of PROTO_UART, PROTO_I2C, or PROTO_ONE_WIRE tells the agent which communication protocol the design uses and therefore what address, data, and timing structure the test sequences must follow. Second, the agent reads the DUT and driver port declarations to extract

the correct signal names, which are then used in the generated code rather than generic placeholders.

5.8.2 Prompt-Driven Sequence Generation

The engineer writes a natural language prompt in the VS Code chat panel describing the desired stimulus in functional terms. A typical prompt might read: “ramp the supply from 0 to 1.8 V over 50 ns, wait for 10 ns, then send the byte 0xA5 to address 0x04 using the selected protocol.” The agent interprets this description, resolves the protocol-specific transaction format from the testbench context it has read, and generates the corresponding SystemVerilog test sequence code.

The generated code includes correct timing constructs — #delay statements for ramp steps, repeat blocks for protocol transactions, and @(posedge clk) synchronisation where needed. Address and data values specified in the prompt are translated into the correct bit width and formatting for the protocol in use. For One-Wire protocol, the agent generates the appropriate reset-presence and read/write slot sequences. For UART, it constructs bit-level pulse sequences using the localparam timing values already present in the testbench. The generated test sequence is written directly into the testbench file at the designated stimulus placeholder section, and the full output — including the agent’s reasoning and the generated code — is visible in the VS Code chat window for review.

The use of Claude Sonnet 4.6 as the backend model provides sufficient language understanding to handle informally phrased prompts and to infer intent from abbreviated or shorthand descriptions, which is common in engineering communication. The testbench file context — provided as input alongside the prompt — grounds the model’s generation in the specific signal names and protocol characteristics of the design at hand, compensating for the model’s lack of prior exposure to this specific design and producing output that is syntactically and semantically compatible with the existing testbench structure.

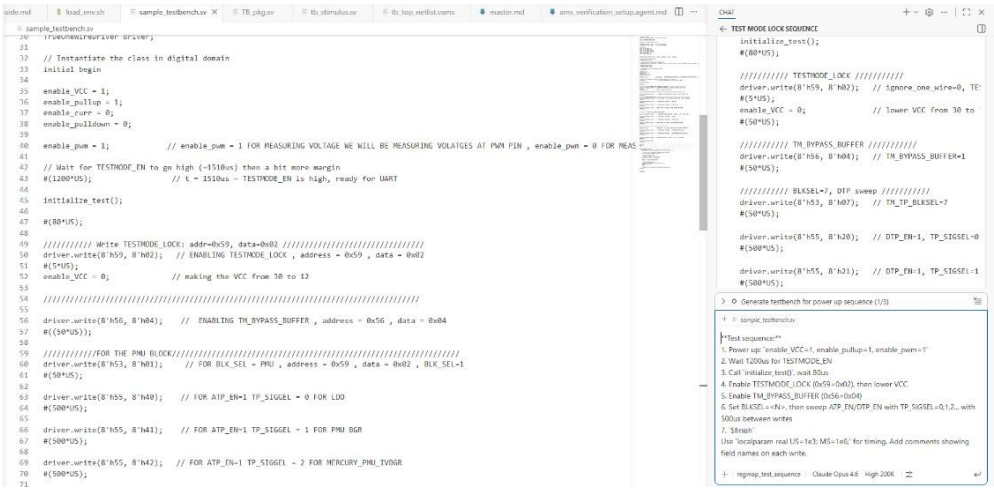


Figure 5.11 VS Code agent chat interface illustrating prompt-driven test sequence generation

Figure 5.11 describes the VS Code agent's chat interface during prompt-driven test sequence generation, illustrating how a functional stimulus description becomes SystemVerilog code.

5.9 SUMMARY

This chapter has provided a detailed account of the implementation logic underlying the five core automation modules of the AMS Verification Automation Framework. The netlist parsing engine was described as a two-pass streaming processor that uses set subtraction for top module identification and handles both ANSI and non-ANSI port declaration styles. The Verilog-AMS driver generator was shown to apply a systematic port direction reversal and to construct real-variable bridges between the digital initial block and the analog V() access functions. The testbench template generator was explained as a string-based line-by-line constructor that injects protocol-specific content when a communication protocol is selected from the package. The AMSCF and PVT corner file generator was described as parsing technology model files to discover process sections and voltage supplies, computing the full Cartesian product of user-selected PVT parameters, and writing one correctly formatted Spectre-syntax corner file per combination. The regression script generator was shown to compute the T × C test-corner cross-combination and produce a shell script with properly formatted Cadence Xcelium commands, sleep intervals, and automatic deployment to the simulation directory. The chapter concluded with the AI-assisted test sequence generation agent, which uses Claude Sonnet 4.6 to produce protocol-aware stimulus code from natural language prompts, grounded in the context of the open testbench file.

Chapter 6 follows with an analysis of the results obtained from the framework, including the time savings measured during testing, the accuracy of the generated files across multiple corner configurations, and the quality of the AI-generated test sequences.

CHAPTER 6

RESULTS AND DISCUSSION

6.1 INTRODUCTION

The preceding chapters described in detail what the AMS Verification Automation Framework does and how each of its modules is implemented. This chapter now shifts focus to what was actually observed during testing — what the tool produced, whether those outputs were correct, and how much time was saved compared to the manual approach. The results are discussed across five areas: netlist parsing and hierarchy extraction, testbench template generation, AMSCF and PVT corner file generation, AI-assisted test sequence generation, and overall productivity impact. The chapter concludes with an honest discussion of the limitations encountered during validation and their implications for future development.

All results reported here were obtained by testing the framework against the netlist of the design under test (DUT) used in the internship project. The test environment used Cadence Xcelium as the simulation engine, and validations were conducted for the One-Wire protocol flow. Where time comparisons are made, the manual effort estimate reflects the typical time observed or reported before the automation tool was available.

6.2 EXPERIMENTAL SETUP

The framework was tested on a Linux-based workstation running Python 3. The design under test is a mixed-signal analog IP block whose netlist was provided in Verilog-AMS (.vams) and SystemVerilog (.sv) formats. The testing covered all four automated stages of the framework: netlist parsing, testbench template generation, AMSCF corner file generation, and regression script generation. Additionally, the VS Code agent was evaluated for test sequence generation on the same design.

The PVT corner configuration used for testing consisted of three process sections, three voltage levels, and three temperature values, yielding a total of 27 corner combinations. The One-Wire protocol was selected for the testbench generation tests. The AI agent was evaluated by providing it with a multi-step natural language prompt describing a realistic AMS stimulus scenario, and the quality and correctness of the generated output was assessed manually.

6.3 NETLIST PARSING AND HIERARCHY EXTRACTION RESULTS

6.3.1 Top Module Identification

The netlist parser was applied to the DUT netlist file, which contains module definitions in both SystemVerilog and Verilog-AMS format. The set subtraction algorithm described in Chapter 5 successfully identified the correct top-level module on the final version of the parser. It is worth noting that during the development phase, an early version of the parser incorrectly identified multiple candidate modules as potential top-level modules. This occurred because some sub-modules in the design had no instantiations pointing to them within the parsed file segment, causing them to pass the set subtraction test as false positives. This issue was rectified by refining the candidate selection logic to choose the module with the greatest number of direct child instantiations, after which the correct top module was identified reliably.

6.3.2 Pin List Extraction

Port extraction from the identified top module was tested against the actual DUT interface. An early version of the extraction logic encountered difficulties with the specific port declaration style used in the DUT netlist, which resulted in some ports being missed or their directions being misclassified. After iterative refinement of the regex patterns to correctly handle both ANSI and non-ANSI declaration styles, as well as multi-line port declarations, the final version of the parser extracted the complete port list correctly. All input, output, and inout ports were identified with the correct directions, and wreal-typed analog ports were correctly distinguished from logic-typed digital ports.

The parsed port information was written to the netlist analysis report file and displayed in the GUI, giving a clear, structured summary of the DUT interface that served as the input to all subsequent automation stages.

6.3.3 Hierarchy Tree and Duplicate Detection

The module hierarchy traversal correctly built the parent-child relationship tree for the DUT netlist, identifying the top module and all its sub-modules with their respective instance names. The duplicate module detection logic scanned the full file and reported any module names that appeared more than once. This feature proved directly useful during testing — one of the sub-modules in the DUT netlist had a redefinition that could have caused elaboration conflicts in Xcelium if undetected. The tool flagged this and commented out the duplicate definition, allowing the simulation to proceed without manual intervention.

6.3.4 Netlist Format Limitation

An important limitation identified during testing is that the parser operates correctly only on SystemVerilog (.sv) and Verilog-AMS (.vams) format netlists. When tested against SPICE-syntax (.scs) or layout-extracted (.tit) netlists, the module definition patterns used by the regex engine did not match the subcircuit declaration syntax of those formats, and the parser returned no results. This is a known limitation of the current implementation and is discussed further in Section 6.7.

```
Analyzing DUT: netlist_05_may_with_GAN.vams
Running: Parsing DUT file

Top Module: MERCURY_PRIME_MCM
Total Ports: 6

Extracted Ports:
    1. CS                                          (output, wire)
    2. DH                                          (inout , wire)
    3. PWM                                         (input , wire)
    4. S                                           (input , wire)
    5. SL                                          (inout , wire)
    6. VCC                                         (inout , wire)

Submodule Instantiations (3):
  - MERCURY_PRIME_PKG IPAR
  - MERCURY_TOP ISI
  - LS_GAN_TOP IGaN
✓ Generated prompt file: prompts/stage2_dut_analysis_prompt.txt
```

Figure 6.1 Netlist analysis report showing extracted top module, categorised pin list, and submodule hierarchy

Figure 6.1 depicts the netlist analysis report produced for the validation DUT, showing the extracted top module, categorised pin list, and submodule hierarchy used for testing.

6.4 TESTBENCH TEMPLATE GENERATION RESULTS

The testbench template generator was evaluated using the One-Wire protocol selection. The generated template contained all the expected structural elements: wire and real declarations for every DUT port, DUT instantiation with named port connections, driver instantiation, protocol master module instantiation with correct One-Wire timing localparams, and the always blocks bridging the protocol outputs to the driver internal variables. The generated file was passed directly to the Cadence Xcelium compiler without any manual edits, and it compiled without errors on the first attempt.

This result is significant because testbench compilation errors are common when files are written manually — a mismatched port width, a missing wire declaration, or a type mismatch between a wreal and a logic signal can each cause elaboration failures that take time to diagnose. The automated generation of the structural skeleton, guided by the correctly parsed

port information, avoided all of these issues. The protocol injection logic correctly instantiated the One-Wire protocol module and tied off unused UART and SPI ports to constants, as required by the Xcelium elaborator.

The directory structure created by the tool during this stage — including the testbench directory, AMSCF directory, and protocol source directory — was verified to match the expected project layout. All files were placed in the correct locations without any path errors.

Figure 6.2 Generated testbench template for the DUT with UART protocol instantiated

Figure 6.2 explains the generated testbench template with the UART protocol instantiated, showing the structure that compiled in Cadence Xcelium without any manual correction.

6.5 AMSCF AND PVT CORNER FILE GENERATION RESULTS

The AMSCF generation module was evaluated with a configuration of three process sections, three voltage levels, and three temperature values. The Cartesian product of these three dimensions produced 27 unique corner combinations, and the tool generated 27 corresponding Spectre-format corner files. All 27 files were generated with syntactically correct content — verified by inspection of the include statements, vsource definitions, temperature option settings, and portmap directives in each file.

Table 6.1 PVT corner configuration used for AMSCF generation validation

PVT Dimension	Values Selected	Count
Process Corners	nom, slow, fast	3
Voltage Levels	Low, Nominal, High	3
Temperature (°C)	-40, 27, 125	3
Total Combinations	**3 × 3 × 3**	**27**

Each generated file followed the naming convention described in Chapter 5, encoding the process section, voltage string, and temperature in the filename. The filenames were verified to be unique across all 27 combinations, ensuring that no corner file would accidentally overwrite another. The technology file parsing correctly discovered the process section names and voltage supply names from the PDK model file provided, without requiring the user to enter these manually.

The time taken to generate all 27 corner files from the point of confirming the PVT selection in the GUI was less than five minutes, including the technology file parsing step. Prior to the availability of this tool, preparing even a single AMSCF file manually required looking up the technology file structure, constructing the include and vsource statements by hand, and verifying the syntax — a process that could take several hours for a full corner set. The automation effectively reduced what was previously a multi-hour manual task to an operation requiring only a few user interactions in the GUI.

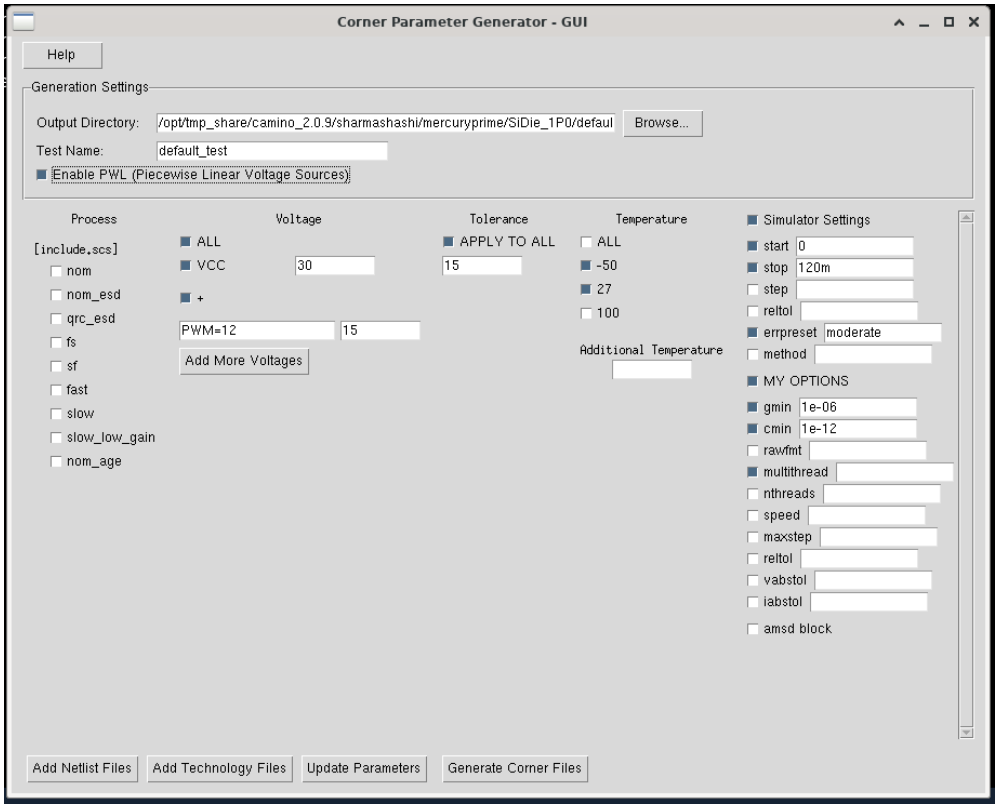


Figure 6.3 AMSCF configuration sub-window with PVT selections and list of 27 generated corner files

Figure 6.3 describes the AMSCF configuration sub-window populated with the validation PVT selections, listing all 27 corner files generated for the regression run.

6.6 TIME SAVINGS ANALYSIS

One of the primary motivations for developing the AMS Verification Automation Framework was to reduce the time spent on repetitive setup tasks in the AMS verification flow. The results of the validation testing provided concrete evidence that this goal was achieved. Table 6.2 summarises the estimated time for each major setup task before and after the introduction of the tool.

Table 6.2 Comparison of manual versus automated setup time for key verification tasks

Verification Setup Task	Manual Effort (Before Tool)	Automated (After Tool)
Full test environment setup	1 to 2 working days	Less than 5 minutes
AMSCF files for all 27 corners	Several hours	Less than 5 minutes
Testbench template generation	Hours of manual coding	Less than 5 minutes
Regression script preparation	Manual per-corner command writing	Automated T × C generation

The most striking improvement is in the overall test environment setup time. Before the tool, a verification engineer setting up a new AMS simulation environment from scratch would

typically spend one to two full working days on the task — parsing the netlist manually to understand the interface, writing the testbench by hand, constructing AMSCF files for each corner, setting up directory structures, and writing the regression commands. With the framework in place, the same setup is completed in well under five minutes from the command-line invocation to having a complete set of ready-to-simulate files.

Think of it like building furniture: manually, you measure every piece, cut by hand, and assemble step by step — which takes all day. The automation tool is like having a CNC machine that reads the blueprint and cuts every piece correctly in minutes, leaving you only to assemble and refine. The engineer's time is now focused on the intellectually demanding parts of verification — designing meaningful test scenarios and analysing simulation results — rather than on the mechanical setup work that preceded them.

It should be noted that the time savings figures in Table 6.2 are based on observation and comparison with typical manual practice, and not on a controlled formal time study with multiple participants. However, even allowing for generous variability in individual working speeds, the order-of-magnitude difference between the manual and automated durations is large enough to be practically significant regardless of measurement method.

6.7 AI-ASSISTED TEST SEQUENCE GENERATION RESULTS

6.7.1 Test Scenario Used for Evaluation

The VS Code agent, powered by Claude Sonnet 4.6, was evaluated on a multi-step test scenario representative of a real AMS verification task. The prompt provided to the agent described the following sequence of operations: first, ramp the VCC supply voltage from zero to its nominal value; second, wait until the testmode enable signal transitions high; third, initiate UART synchronisation; and fourth, write four successive address-data pairs over the UART interface, where each transaction enables or disables a specific internal signal within the DUT.

This scenario was deliberately chosen to test the agent's ability to handle several different categories of stimulus in a single generation task: an analog ramp (requiring timed real-valued assignments), a conditional wait (requiring event-based synchronisation), a protocol handshake (requiring UART-specific timing), and multiple repeated transactions (requiring a structured loop or sequential stimulus blocks). Generating all of this manually would require the engineer to look up the timing localparams in the testbench, construct the ramp sequence with correct delay increments, and write four correctly formatted UART write transactions with the right address and data values.

6.7.2 Quality of Generated Output

The agent generated the complete test sequence in a single prompt interaction. The generated code correctly included the supply ramp section with incremental voltage assignments, the event-based wait for the testmode enable signal, the UART synchronisation call, and all four address-data transaction sequences. The structural logic of the generated sequence was correct — the order of operations, the use of protocol task calls, and the signal names were all consistent with the testbench context that the agent had read.

However, manual corrections were needed in two areas. First, the delay syntax used in the supply ramp section was occasionally incorrect — the agent sometimes used a delay format that is valid in SystemVerilog but incompatible with the specific simulation mode configured in the Xcelium environment. Second, in some of the ramp steps, intermediate voltage values were omitted, producing a step-change rather than a smooth ramp. Both issues required the engineer to make targeted edits to the generated code before it was ready for simulation.

For more complex scenarios involving multiple interdependent conditions or non-standard protocol timing, the agent required follow-up prompts to refine the generated sequence. This is consistent with the published literature on LLM-based testbench generation — as discussed in Chapter 2, even dedicated tools like CorrectBench and AutoBench report pass rates below 100%, reflecting the inherent complexity of generating syntactically and functionally correct HDL code from natural language descriptions in a single attempt.

6.7.3 Overall Assessment of Agent Capability

Despite the need for occasional manual correction, the AI agent component provides a meaningful productivity benefit. Generating the structural scaffold of a complex multi-step test sequence from a single natural language prompt — even one that needs some post-editing — is substantially faster than writing the entire sequence from scratch. The agent correctly inferred the protocol in use from the testbench context, used the correct signal names throughout, and produced syntactically valid SystemVerilog code that required only targeted fixes rather than complete rewrites. As the quality of the underlying language model improves over time, the frequency of the observed correction cases is expected to reduce.

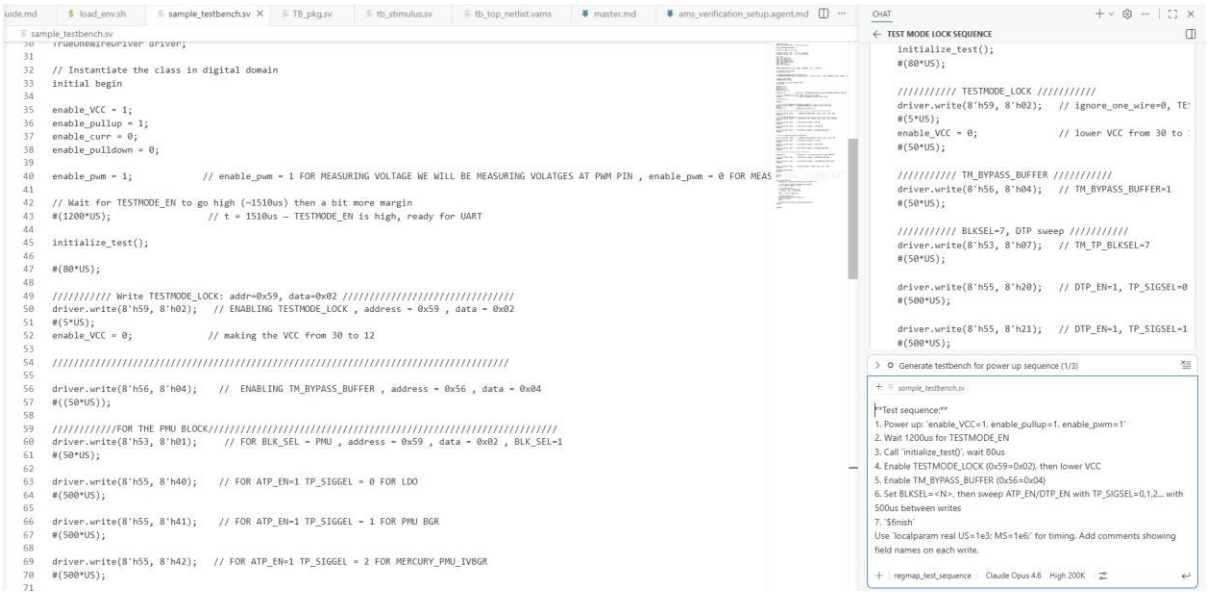


Figure 6.4 VS Code agent output for the VCC ramp and UART write sequence prompt

Figure 6.4 depicts the VS Code agent's output for the VCC ramp and UART write-sequence prompt, showing the multi-step stimulus generated from a single instruction.

6.8 OBSERVED LIMITATIONS

A balanced evaluation of any engineering tool must include an honest account of its limitations, and the AMS Verification Automation Framework is no exception. Several limitations were identified during the testing and development phases, and these are documented here both for completeness and to provide clear direction for future work.

6.8.1 Netlist Format Support

The most significant functional limitation of the current implementation is its restricted netlist format support. The parser handles SystemVerilog (.sv) and Verilog-AMS (.vams) formats correctly but does not support SPICE-syntax (.scs) or extracted-layout (.tit) netlists. In practice, many analog IP blocks at the transistor level are described in SPICE format, and layout-extracted netlists in .tit format are commonly used for post-layout simulation. The absence of support for these formats means the tool cannot be used in its current form for transistor-level or post-layout AMS verification flows, which limits its applicability to the behavioural and RNM abstraction levels.

6.8.2 Workflow Fragmentation Between GUI and Agent

The current architecture requires the engineer to work across two separate environments: the Tkinter GUI for the scripted automation stages, and the VS Code agent for test sequence

generation. Switching between these two tools interrupts the workflow and increases cognitive overhead. In an ideal implementation, the agent would be accessible directly from within the GUI, so the engineer never needs to leave the framework environment to complete the full verification setup. This integration is identified as a key improvement for future development.

6.8.3 Absence of In-GUI Simulation Launch

The regression execution in the current implementation is performed by running the generated run.sh shell script from the command line, outside the GUI. A “Launch Simulation” button within the GUI itself was identified as a desired feature but was not implemented within the internship project timeline. Its absence means the engineer must manually switch to a terminal to execute the regression after the GUI has completed all setup steps, which partially breaks the otherwise seamless GUI-driven workflow.

6.8.4 Residual Manual Intervention Points

While the framework automates a large proportion of the setup effort, several points in the workflow still require manual input from the engineer. These include the pin mapping step in the protocol integration stage, the entry of technology file paths and voltage nominal values in the AMSCF configuration sub-window, and the editing of AI-generated test sequences where delay syntax or ramp step values need correction. The cumulative effect of these manual touchpoints means the workflow is not yet fully end-to-end automated, and the degree of smoothness between stages could be improved in future iterations.

6.9 DISCUSSION

The results presented in this chapter demonstrate that the AMS Verification Automation Framework achieves its primary objective: substantially reducing the time and manual effort required to set up an AMS verification environment. The reduction from one to two working days to less than five minutes for the full test environment setup is an outcome that has direct practical value in any industrial verification schedule, where reducing setup overhead frees engineering time for the more critical tasks of test planning, coverage analysis, and result interpretation.

The correctness of the generated outputs — 27 syntactically valid AMSCF files, a compilation-error-free testbench template, and a correctly structured regression script — validates the implementation approach taken in each module. The fact that the testbench compiled without errors on the first attempt is particularly noteworthy, given that type mismatches and port

connection errors are among the most common sources of testbench compilation failures in AMS flows.

The AI agent results highlight both the promise and the current limitations of LLM-based test generation in analog mixed-signal contexts. The agent's ability to produce a structurally correct multi-step stimulus sequence from a single natural language prompt represents a genuine capability that did not exist in the manual flow at all. The need for post-editing corrections in timing syntax and ramp values is a limitation, but not a disqualifying one — it is consistent with the state of the art in the published literature, and the edited output is still produced far faster than a hand-written equivalent would be.

The limitations identified in Section 6.8 are real and significant, but they are also well-defined and actionable. None of them represent fundamental architectural problems with the framework — they are all addressable through incremental development. The netlist format limitation, for example, requires extending the parser to handle SPICE subcircuit syntax, which is a well-understood text processing problem. The GUI-agent integration limitation requires an API connection between the Tkinter application and the VS Code extension, which is technically feasible. These directions are discussed further in Chapter 7.

6.10 SUMMARY

This chapter presented the results of validating the AMS Verification Automation Framework against the DUT netlist used in the internship project. The netlist parser correctly identified the top module and extracted the complete pin list after iterative refinement, with duplicate module detection providing a practically useful error-prevention capability. The testbench template generator produced a compilation-ready file for the One-Wire protocol flow on the first attempt. The AMSCF generator produced all 27 PVT corner files with correct syntax, reducing what was previously a several-hour manual task to an operation taking less than five minutes. The AI agent demonstrated the ability to generate complex multi-step AMS test sequences from a single natural language prompt, though human review and correction of timing details was found to be necessary in some cases. Four specific limitations were identified — restricted netlist format support, GUI-agent fragmentation, the absence of an in-GUI simulation launch, and residual manual intervention points — which collectively define the scope of future development described in Chapter 7.

CHAPTER 7

CONCLUSION AND FUTURE SCOPE

7.1 INTRODUCTION

This thesis set out to address a clearly defined and practically significant problem: the manual overhead involved in setting up AMS verification environments, which consumes a disproportionate amount of a verification engineer's time on tasks that are repetitive, structured, and well-suited to automation. The AMS Verification Automation Framework developed during the course of this internship project demonstrates that it is possible to automate the most time-intensive parts of this setup — from netlist parsing and testbench generation through to multi-corner AMSCF file creation and regression script preparation — and to complement that automation with an AI agent capable of generating test sequences from natural language prompts. This concluding chapter summarises the work carried out, highlights the key contributions, states the conclusions drawn from the results, and outlines the directions identified for future development.

7.2 SUMMARY OF WORK

The work described in this thesis began with an analysis of the AMS verification flow as practised in an industrial environment, using Cadence Xcelium and Spectre as the primary simulation tools. The analysis identified five specific areas of manual effort that were both significant in time cost and amenable to automation: netlist parsing and design hierarchy extraction, testbench template construction, AMSCF file generation across PVT corners, regression script preparation, and test sequence creation.

To address these areas, a Python-based automation framework was designed and implemented. The framework is structured around a stage-driven graphical user interface built with the Tkinter library, which guides the verification engineer through four sequential processing stages without requiring knowledge of the underlying Python implementation. The first stage parses the design netlist using a two-pass streaming algorithm that automatically identifies the top-level module, extracts the complete port list with direction and type information, builds the submodule hierarchy, and detects duplicate module declarations. The second stage generates a complete SystemVerilog testbench template from the parsed port information, with automatic injection of the selected communication protocol — UART, SPI, or One-Wire — from a built-in reusable protocol package. The third stage uses a dedicated sub-window to collect PVT

corner specifications and generates the full set of Spectre-format AMSCF files through a Cartesian product computation across process, voltage, and temperature dimensions. The fourth stage prepares the regression shell script by computing the complete test-corner cross-combination matrix and formatting the corresponding Cadence Xcelium invocation commands.

Complementing the GUI-based scripted stages, a VS Code-integrated AI agent was developed using Claude Sonnet 4.6 as the backend language model. This agent reads the open testbench file to determine the protocol and signal context, and generates test sequences from natural language prompts. The output — including supply ramp sequences, conditional wait constructs, and protocol transaction blocks — is written directly into the testbench file, accelerating the most creatively demanding part of the verification setup process.

7.3 KEY CONTRIBUTIONS

The following are the principal contributions of this work:

1. **Automated Netlist Parsing Engine —** A two-pass streaming parser was designed and implemented in Python that automatically identifies the top-level module in a hierarchical AMS netlist using a set subtraction algorithm, extracts the complete port list with direction and type classification, constructs the submodule hierarchy tree, and detects duplicate module declarations. The parser handles both ANSI-style and non-ANSI-style port declarations and correctly distinguishes wreal-typed analog ports from logic-typed digital ports, a distinction that is critical for correct testbench generation in mixed-signal designs.
2. **End-to-End GUI-Driven Verification Setup Workflow —** A complete, stage-based graphical user interface was developed that guides a verification engineer from netlist input through to a simulation-ready environment without requiring Python scripting knowledge. The GUI encapsulates parsing, testbench generation, AMSCF configuration, and regression preparation in a single cohesive workflow, reducing the cognitive burden of managing multiple disconnected scripts and manual steps. This end-to-end workflow represents a practically deployable tool for AMS verification teams working with Cadence Xcelium.
3. **Multi-Corner AMSCF Generation —** An automated AMSCF generation module was implemented that parses technology model files to auto-discover process sections and voltage supply names, computes the Cartesian product of user-selected PVT parameters, and generates a complete set of correctly structured Spectre-format corner files with consistent naming conventions. A configuration requiring 27 corner files

was completed in under five minutes, with all generated files verified to be syntactically correct.

4. **AI-Assisted Test Sequence Generation via LLM Agent —** A VS Code-based AI agent was developed using Claude Sonnet 4.6 that generates protocol-aware test sequences from natural language prompts. The agent reads the testbench file context to infer the protocol and signal names, and produces SystemVerilog stimulus code including supply ramp sequences, event-based waits, and protocol transaction blocks. The agent significantly reduces the time needed to produce multi-step test stimulus, though human review is recommended for timing-critical parameters.

7.4 CONCLUSIONS

The results reported in Chapter 6 provide concrete evidence that the AMS Verification Automation Framework achieves its intended objective. The reduction in full test environment setup time from one to two working days to under five minutes is practically significant by any measure. Across all tested scenarios, the generated outputs were syntactically correct, and the testbench template compiled without errors in Cadence Xcelium on the first attempt. These outcomes validate both the correctness of the underlying algorithms and the practical usefulness of the tool as a whole.

Perhaps the most important conclusion to draw from this work is that the AMS verification setup problem is not fundamentally a knowledge problem — it is a repetition problem. The structure of a testbench, a corner file, or a regression script is well understood by any experienced verification engineer. The time consumed in producing these files manually is not time spent thinking; it is time spent transcribing the same patterns over and over again. Automating this transcription does not require sophisticated intelligence — it requires careful engineering of structured text processing and template generation, which Python is well suited to. The framework demonstrates this principle clearly.

The AI agent component leads to a complementary conclusion. Test sequence generation does involve creative judgement — choosing what stimulus to apply, in what order, and with what parameters. This is where language model assistance provides genuine value, because the agent can translate a functional description into structured code faster than a human can write it from scratch. The limitation observed — occasional errors in delay syntax and ramp step values — reflects the current state of LLM capability for domain-specific HDL generation rather than a fundamental problem with the approach itself.

From an industrial applicability standpoint, the framework makes a specific and distinctive contribution by enabling corner-based regression execution through Cadence Xcelium. Prior to this work, running multi-corner regressions in the team's environment required a separate and more complex tooling setup. The framework integrates this capability directly into the GUI-driven workflow, making it accessible to any engineer on the team without additional infrastructure overhead. This is the contribution of most immediate practical value for an AMS verification team working with the Cadence Xcelium simulation infrastructure.

7.5 FUTURE SCOPE

The limitations identified in Chapter 6 point to clear and well-defined directions for future development, each of which would meaningfully extend the tool's capability and completeness.

7.5.1 Extended Netlist Format Support

The most significant functional limitation of the current implementation is its restriction to SystemVerilog (.sv) and Verilog-AMS (.vams) netlist formats. In an industrial AMS verification environment, transistor-level blocks are commonly described in SPICE-syntax (.scs) format, and post-layout extracted netlists often use proprietary formats. Extending the parser to handle .scs and .vhdl formats would substantially broaden the tool's applicability and enable its use across different abstraction levels of the design hierarchy. This is considered the highest-priority future development task, since netlist format diversity is a practical constraint encountered in nearly every industrial AMS project.

7.5.2 In-GUI Simulation Launch

The current flow requires the engineer to leave the GUI and run the generated regression script from the command line. Adding a Launch Simulation button that invokes the script through a Python subprocess call and streams progress output to a log panel within the GUI would complete the end-to-end workflow within a single interface. This is a relatively straightforward implementation and would significantly improve usability, particularly for engineers who are less accustomed to terminal-based operation.

7.5.3 Direct Integration of the AI Agent into the GUI

Integrating the AI agent directly into the Tkinter GUI as an embedded chat panel would eliminate the context switch currently required between the GUI and VS Code. Technically, this involves connecting the GUI to the language model API directly from Python. Such integration would also allow the agent to automatically receive the parsed port list and selected

protocol from the GUI state, making its context awareness tighter and reducing the potential for mismatch between what the GUI knows and what the agent reads from the open file.

7.5.4 End-to-End Flow Automation

Several stages of the current workflow retain manual touchpoints — including technology file path entry, pin mapping, and occasional correction of AI-generated timing values. Future development should target a reduction in these manual steps through smarter default detection, automatic validation of generated files, and richer in-GUI error feedback. Incorporating a dry-run compilation check — where the generated testbench is automatically compiled in a test mode after generation and any errors reported within the GUI — would catch structural issues before the engineer ever needs to open the file manually.

7.5.5 Expanded Protocol Package

The current protocol package supports UART, SPI, and One-Wire. Many AMS designs in the power management and automotive domains also use protocols such as I2C, PMBus, and LIN. Expanding the protocol package to include these would broaden the framework's coverage across different IP families. The package architecture, with its single-parameter protocol selection mechanism, is specifically designed to accommodate this kind of extension without requiring changes to the core framework code.

7.6 CLOSING REMARKS

The AMS Verification Automation Framework is, at its core, built around a simple but important idea: that a verification engineer's time is too valuable to be spent on mechanical, repetitive setup tasks. The hours saved by automating netlist parsing, testbench construction, and corner file generation are hours that can instead be spent understanding the design, thinking through edge cases, and interpreting simulation results — the parts of verification that genuinely require human expertise and cannot be automated away.

What makes this work particularly meaningful in an industrial context is its accessibility. Any engineer joining an AMS verification team — regardless of their prior experience with the simulation infrastructure or with the Cadence Xcelium environment — can set up a complete multi-corner regression environment for a new design in minutes, using only the netlist as input. There is no prerequisite knowledge of AMSCF syntax, no need to understand the protocol package internals, and no requirement to write HDL manually just to get the simulation environment operational. A new team member can be productive from day one

rather than spending their first week learning setup procedures. This accessibility is what gives the tool its broader and lasting value.

The integration of an LLM-based agent for test sequence generation points toward a future where the gap between what an engineer wants to verify and what the simulator actually receives as input becomes increasingly narrow. The framework developed in this thesis is a practical and grounded step in that direction — one that is already useful today, and well positioned to become more capable as both the underlying language models and the broader AMS verification toolchain continue to evolve.

REFERENCES

[1] G. E. Gielen and R. A. Rutenbar, “Computer-aided design of analog and mixed-signal integrated circuits,” Proceedings of the IEEE, vol. 88, no. 12, pp. 1825–1854, Dec. 2000.

[2] K. Kundert, The Designer’s Guide to SPICE and Spectre. Boston, MA: Kluwer Academic Publishers, 1995.

[3] Siemens EDA, “Dawn of the new mixed-signal verification era and the need for revolution,” Verification Horizons Blog, Feb. 2019. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons. [Accessed: Jun. 2026].

[4] N. Georgoulopoulos and A. Hatzopoulos, “Real number modeling of a Flash ADC using SystemVerilog,” ResearchGate, Nov. 2017. [Online]. Available: https://www.researchgate.net/publication/321278387. [Accessed: Jun. 2026].

[5] N. Georgoulopoulos, K. Siozos, and A. Hatzopoulos, “Design and verification of a SAR ADC SystemVerilog real number model,” Journal of Electronic Testing, vol. 40, Jul. 2024.

[6] B. Brennan and M. Testorf, “Real number modeling,” in Proc. Design Verification Conference (DVCon), San Jose, CA, 2013. [Online]. Available: https://dvcon-proceedings.org. [Accessed: Jun. 2026].

[7] Cadence Design Systems, “Demystifying mixed-signal simulation for digital verification engineers,” Cadence White Paper, San Jose, CA. [Online]. Available: https://www.cadence.com. [Accessed: Jun. 2026].

[8] D. Frey and J. O’Riordan, “Verilog-AMS: Mixed-signal simulation and cross domain connect modules,” in Proc. IEEE Custom Integrated Circuits Conference (CICC), 2000.

[9] Renesas Electronics Corporation, “Mixed signal design and verification methodology for complex SoCs,” Renesas White Paper, Dec. 2021. [Online]. Available: https://www.renesas.com. [Accessed: Jun. 2026].

[10] Cadence Design Systems, “Start your engines: The innovation behind Universal Connect Modules (UCM),” Cadence Community Blog, Aug. 2024. [Online]. Available: https://community.cadence.com. [Accessed: Jun. 2026].

[11] National University of Singapore, Mixed Signal Design Simulation Manual, Dept. of Electrical and Computer Engineering, Singapore, 2005.

[12] Cadence Design Systems, Spectre AMS Designer and Xcelium Simulator Mixed-Signal User Guide, Release 24.03, San Jose, CA, 2024.

[13] Cadence Design Systems, Xcelium Logic Simulator Datasheet, San Jose, CA, 2024.

[14] Cadence Design Systems, “New Cadence Xcelium Apps accelerate simulation-based verification for automotive, mobile and hyperscale designs,” Cadence Press Release, Jun. 2022. [Online]. Available: https://www.cadence.com. [Accessed: Jun. 2026].

[15] Accellera Systems Initiative, Universal Verification Methodology (UVM) 1.2 User's Guide, Napa, CA, 2015. [Online]. Available: https://www.accellera.org. [Accessed: Jun. 2026].

[16] IEEE Standard for Universal Verification Methodology (UVM), IEEE Std 1800.2-2020, New York, NY, 2020.

[17] Accellera Systems Initiative, UVM Mixed-Signal (UVM-MS) 1.0 Standard, Napa, CA, Jan. 2025. [Online]. Available: https://www.accellera.org. [Accessed: Jun. 2026].

[18] N. Georgoulopoulos et al., "UVM-based verification of a mixed-signal design using SystemVerilog," in Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2018.

[19] H. Lin, Z. Ye, and A. M. Khan, "Machine learning-based PVT space coverage and worst case exploration in analog and mixed-signal design verification," in Proc. Design Verification Conference (DVCon), San Jose, CA, Mar. 2017.

[20] Keysight Technologies, "Python APIs for EDA workflows," Keysight EDA Resource Centre, Sep. 2024. [Online]. Available: https://www.keysight.com. [Accessed: Jun. 2026].

[21] Python Software Foundation, "Tkinter — Python interface to Tcl/Tk," Python Documentation, 2024.[Online]. Available: https://docs.python.org/3/library/tkinter.html. [Accessed: Jun. 2026].

[22] R. Edelman and S. Bhutada, "UVM sans UVM: An approach to automating UVM testbench writing," in Proc. Design Verification Conference (DVCon), San Jose, CA, Feb. 2014.

[23] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "AutoBench: Automatic testbench generation and evaluation using LLMs for HDL design," in Proc. ACM/IEEE Int. Symp. on Machine Learning for CAD (MLCAD), Salt Lake City, UT, Sep. 2024.

[24] R. Qiu et al., "CorrectBench: Automatic testbench generation with functional self-correction using LLMs for HDL design," in Proc. Design, Automation and Test in Europe (DATE), 2025.

[25] M. Liu, T.-D. Ene, R. Kirby et al., "ChipNeMo: Domain-adapted LLMs for chip design," arXiv preprint arXiv:2311.00176, Oct. 2023. [Online]. Available: https://arxiv.org/abs/2311.00176. [Accessed: Jun. 2026].

[26] W. Liu, F. Hou, J. Zhang, H. Liu, and A. Lei, "A multi-agent generative AI framework for IC module-level verification automation," arXiv preprint arXiv:2507.21694, Jul. 2025. [Online]. Available: https://arxiv.org/abs/2507.21694. [Accessed: Jun. 2026].

[27] NVIDIA Corporation, "Building AI agents to automate software test case creation," NVIDIA Developer Blog, Oct. 2024. [Online]. Available: https://developer.nvidia.com. [Accessed: Jun. 2026].

[28] A. Garg et al., "DV UVM based AMS co-simulation and verification methodology for mixed signal," in Proc. Design Verification Conference (DVCon), 2023. [Online]. Available: https://dvcon-proceedings.org. [Accessed: Jun. 2026].

Shashikant

ORIGINALITY REPORT

6%	5%	3%	3%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	tudr.thapar.edu Internet Source	
2	arxiv.org Internet Source	
3	dvcon-proceedings.org Internet Source	
4	digitalcollection.utem.edu.my Internet Source	
5	Rashed Al Amin, Md Golam Rassel Lincoln, Roman Obermaisser. "Towards LLM-Assiste HDL Generation and Verification", 2025 14th Mediterranean Conference on Embedded Computing (MECO), 2025 Publication	
6	fenix.tecnico.ulisboa.pt Internet Source	
7	www.cadence.com Internet Source	
8	Submitted to Thapar University, Patiala Student Paper	
9	Submitted to University Politehnica of Bucharest Student Paper	
10	Dimple Vijay Kochar, Nathaniel Pinckney, Guan-Ting Liu, Chia-Tung Ho, Chenhui Deng, Haoxing Ren, Brucek Khailany. "GRPO with State Mutations: Improving LLM-Based Hardware Test Plan Generation", 2026 27th International Symposium on Quality Electronic Design (ISQED), 2026 Publication	
11	iris.univr.it Internet Source	
12	pdfs.semanticscholar.org Internet Source	
13	essay.utwente.nl Internet Source	

14 repository.iutoic-dhaka.edu
Internet Source

15 scholars.hkbu.edu.hk
Internet Source

16 Submitted to K L University
Student Paper

17 Submitted to University of Edinburgh
Student Paper

18 Submitted to University of Canterbury
Student Paper

19 mediatum.ub.tum.de
Internet Source

20 elib.uni-stuttgart.de
Internet Source

21 objectstorage.ap-dcc-gazipur-1.oraclecloud15.com
Internet Source

22 Submitted to London School of Science & Technology
Student Paper

23 Submitted to Morgan State University
Student Paper

24 Submitted to Rochester Institute of Technology
Student Paper

25 Sarbishaei, Hassan. "Time-varying Volterra analysis of nonlinear circuits", Proquest, 20111109
Publication

26 Dylan Muñoz Cornejo, Marlon José Loria Guiltres, Christopher Quiros Cisneros, Geovan Mejia Gaitán. "Design of a mixed signal test environment for an edge detector", 2025 IEEE Central America and Panama Student Conference (CONESCAPAN), 2025
Publication

27 M. N. Saranya, Rathnamala Rao. "Design and Verification of an Asynchronous NoC Rou Architecture for GALS Systems", Journal of Electronic Testing, 2024
Publication

28 Meisam Abdollahi, Seyedeh Faegheh Yeganli, Mohammad (Amir) Baharloo, Amirali Baniasadi. "Hardware Design and Verification with Large Language Models: A Scoping Review, Challenges, and Open Issues", Electronics, 2024
Publication

29 Submitted to University of New South Wales
Student Paper

30 technodocbox.com
Internet Source

31 Submitted to Jabatan Pendidikan Politeknik Dan Kolej Komuniti
Student Paper

32 Lange, Andre, Ihor Harasymiv, Oliver Eisenberger, Frederic Roger, Joachim Haase, and Rainer Minixhofer. "Towards probabilistic analog behavioral modeling", 2015 IEEE International Symposium on Circuits and Systems (ISCAS), 2015.
Publication

33 Nikolaos Georgoulopoulos, Alkiviadis Hatzopoulos. "UVM-based Verification of a Digita PLL Using SystemVerilog", 2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), 2019
Publication

34 Submitted to University of Hull
Student Paper

35 github.com
Internet Source

36 semiengineering.com
Internet Source

37 tudr.thapar.edu:8080
Internet Source

38 Submitted to CSU, San Jose State University
Student Paper

39 Ketki Purushottam Kasaralikar, Vaishali Ingale, Vanita Agarwal, Ashlesha Gokhale. "Des and Verification of a Cache Coherency Verification IP for AMBA ACE Protocol Using UVM 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), 2024
Publication

40 Luciano Lavagno, Grant E. Martin, Igor L. Markov, Louis K. Scheffer. "Electronic Design Automation for IC System Design, Verification, and Testing", CRC Press, 2026
Publication

41 Submitted to Strathmore University (Main Account)
Student Paper

42 Yu Zhao, Zexuan Zhao, Yuanhong Jiao, Xiaomin Wei, Heng Yang, Long Liu, Yongcai Hu. "Design of a Reusable UVM Verification Framework for Pixel Readout Chips", 2025 10th International Conference on Integrated Circuits and Microsystems (ICICM), 2025

Publication

43 publikationen.bibliothek.kit.edu
Internet Source

44 www.dcs.ed.ac.uk
Internet Source

45 Abdelrahman Ehsan El-Gendy, S. E. D. Habib. "Cairo University RISC-V (CURISCV) Processor", 2025 International Conference on Electrical, Communication and Compute Engineering (ICECCE), 2025
Publication

46 Tetsuya Iizuka. "Buffer-ring-based all-digital on-chip monitor for PMOS and NMOS proc variability and aging effects", 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, 04/2010
Publication

47 d197for5662m48.cloudfront.net
Internet Source

48 eprints.utm.my
Internet Source

49 etd.astu.edu.et
Internet Source

50 www.semanticscholar.org
Internet Source

51 "MLCAD 2024 Cover Page", 2024 ACM/IEEE 6th Symposium on Machine Learning for CA (MLCAD), 2024
Publication

52 dr.ntu.edu.sg
Internet Source

53 magendoo.ro
Internet Source

54 www.epicos.com
Internet Source

55 www.sce.carleton.ca
Internet Source

56 "Advances in VLSI, Communication, and Signal Processing", Springer Science and Busin Media LLC, 2025
Publication

57 Wei-Tzer Huang, Yu-Cheng Liao, Wei-Chen Lin. " Deterministic Dual-Role I C Verification Bus Ownership Invariants and Bounded Runtime Switching ", IEEE Access, 2026

Publication

58 doczz.net
Internet Source

59 dokumen.pub
Internet Source

60 www.ijert.org
Internet Source

61 "Formal Aspects of Component Software", Springer Science and Business Media LLC, 2
Publication

62 E Bhawani Eswar Reddy, Sutirtha Bhattacharyya, Ankur Sarmah, Fedrick Nongpoh, Kart Maddala, Chandan Karfa. "LHS: LLM Assisted Efficient High-level Synthesis of Deep Learning Tasks", ACM Transactions on Design Automation of Electronic Systems, 2025
Publication

63 R. Rakhi, R. K. Siddharth, M. Vijay Babu, K. G. Shreeharsha, M. H. Vasantha, Y. B. Nithin Kumar. "A Power-Efficient High-Speed D-Latch Architecture for Enhanced PDP Performance", IEEE Access, 2026
Publication

64 Vasudevan, Shobha. "High level static analysis of system descriptions for taming verification complexity", Proquest, 20111108
Publication

65 d-nb.info
Internet Source

66 dspace.daffodilvarsity.edu.bd:8080
Internet Source

67 ir-library.mmust.ac.ke
Internet Source

68 isjem.com
Internet Source

69 lio5599.github.io
Internet Source

70 picture.iczhiku.com
Internet Source

71 unsworks.unsw.edu.au
Internet Source

72 www.mdpi.com
Internet Source

73 www.verificationacademy.com
Internet Source

74 Kaichang Chen, Geroges Gielen. "A Generalized Constraint Learning and Transfer Methodology with Net-First Graph Neural Network and Selective Topological Search fo Hierarchical Analog / Mixed-Signal Circuit Layout Synthesis", ACM Transactions on Desi Automation of Electronic Systems, 2025
Publication

75 Ying Li, J.D. Cressler, Yuan Lu, Jun Pan, Guofu Niu, R.A. Reed, P.W. Marshall, C. Polar, M. Palmer, A.J. Joseph. "Proton tolerance of multiple-threshold voltage and multiple-breakdown voltage cmos device design points in a 0.18 µm system-on-a-chip cmos technology", IEEE Transactions on Nuclear Science, 2003
Publication

76 Dipayan Saha, Hasan Al Shaikh, Shams Tarek, Farimah Farahmandi. "Special Session: ThreatLens: LLM-guided Threat Modeling and Test Plan Generation for Hardware Secu Verification", 2025 IEEE 43rd VLSI Test Symposium (VTS), 2025
Publication

77 Lihong Zhang. "", IEEE Transactions on Computer-Aided Design of Integrated Circuits a Systems, 5/2008
Publication

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	Off		

Dr. Mayank Kumar Rai

[Professor, ECE Department]

Dr. Arnab Pattanayak

[Assistant Professor, ECE Department]