Application of AI to Accelerate Formal verification workflow

Liya Liu
Advanced Micro Devices, Inc.
84 Hines Road, Suite #300
Ottawa, ON K2K 3G3
liya.liu@amd.com

ParmarJayeshkumar
Advanced Micro Devices, Inc.
Prestige Technostar, Doddanakundi
Industrial Area 2, Brookefield
Bengaluru, Karnataka 560048
parmarjayeshkumar.parmar@amd.com

Namrata Teggi
Advanced Micro Devices, Inc.
33 Commerce Valley Dr.
Markham ON L3T 7N6
namrata.teggi@amd.com

Shilpa Reddy
Advanced Micro Devices, Inc.
7171 Southwest parkway
Austin TX 78735
United States
shilpa.reddy@amd.com

ABSTRACT

Formal verification is critical for ensuring correctness in safety-critical hardware systems. However, the setup process remains labor-intensive, requiring configuration of the environment manually, RTL integration via Tcl scripts, and binding interfaces to SystemVerilog (SV) assertions. These tasks become increasingly complex with large signal sets or when properties requiring being reused across various configurations. This paper presents a methodology to automate key aspects of the formal verification workflow with AI-assistant. The approach includes Python script generation for RTL interface extraction, environment setup, and regression integration. The major contribution of the work presented in this paper is automatically implementing SystemVerilog properties based on prompts fed in AI tool. These properties are successfully verified using Synopsys VC Formal (VCF) after being reviewed by verification Engineers. With AI-assistance, the formal verification tests can also be launched in regression efficiently and the regression results are automatically sent to test owners. Applying this method, time consumed in formal verification flow has been largely reduced. Although the experiments are conducted using VCF, the methodology is tool-agnostic and applicable to other formal platforms such as JasperGold and Questa Formal.

Keywords: Formal Verification, GitHub Copilot, SystemVerilog Assertions, Finite-State Machine, Synopsys VC Formal

1 Introduction

Formal verification (FV) [1] is a foundational technique in hardware design validation, offering exhaustive coverage and early bug detection compared to simulation-based methods. The popular formal verification tools are VC Formal (VCF Synopsys) [2], JasperGold (Cadence) [3], and Questa Formal (Siemens) [4], etc.

To initiate FV, engineers configure the environment by scripting VCF shell commands, linking RTL designs, and binding SystemVerilog Assertions (SVA) to the design interface. RTL modules often support different features and reusability with signals designed on their interfaces based on system architecture, many companies adopt script-based flows. For example, RTL modules are defined in .v.dpl Perl scripts that generate Verilog files based on configurable parameters during the build process. On the other hand, formal verification tools require assertion checkers being bound directly to the RTL module in Verilog format. Typically, the RTL module to be formally verified is created during the build process applying deperl commands. This interface is typically constructed manually. It is tedious and error-prone for modules with large port lists. Moreover, writing assertion properties is a manual task that depends heavily on the expertise of the design verification engineer.

Recent advances in artificial intelligence (AI) have introduced new opportunities for automating code generation and improving verification efficiency. This paper presents a Python-based framework, assisted by GitHub Copilot [5,6], that automates the setup of the verification environment, interface generation, and regression analysis. It also explores techniques for automatically generating assertion properties for verifying safety-critical hardware blocks. Experimental results show significant reductions in setup time and manual effort, demonstrating the potential of AI-assisted workflows in formal verification.

2 Approach

This section outlines the formal verification flow, identifies automation opportunities, and describes strategies for generating scripts and properties using AI assistance.

2.1 Formal Verification Flow

The formal verification process begins with collecting configuration data that defines the formal verification environment. This includes setting environment variables, selecting the application mode, specifying the top-level RTL module, identifying clocks/resets/wrapper configurations, and specifying directory for saving the report log file. These inputs guide the generation of essential setup files:

- **Blackbox.yml**: Lists RTL modules to be excluded from formal analysis, often using regular expressions [7] to simplify exclusion of third-party IPs or non-critical logic.
- **Module_wrapper.sv**: Optional wrapper for RTL modules supporting multiple configurations.
- **Module_if.sv**: Interface bound to RTL and contains SystemVerilog properties.
- **Module.tcl**: Core script that configures VC Formal shell commands, sets application parameters, and enables reporting results at a specified location in the workspace.

The formal verification workflow depicted in Figure 1 outlines a modular and repeatable process that begins with collecting configuration input from the user. With the collected information, the script automates file generation, which includes the creation of interface files, wrapper modules, blackbox lists, and Tcl. Once these files are generated, the verification engineer runs VCF to verify the RTL design based on properties. When applied mode is Automatically Extracted Properties (AEP) or Formal X-Propagation application (FXP), the properties are given by VCF. If Formal Property Verification (FPV) or another formal application is applied, then verification engineer should design the required assert properties along with the necessary constraints (assume properties) to verify the features implemented in RTL module.

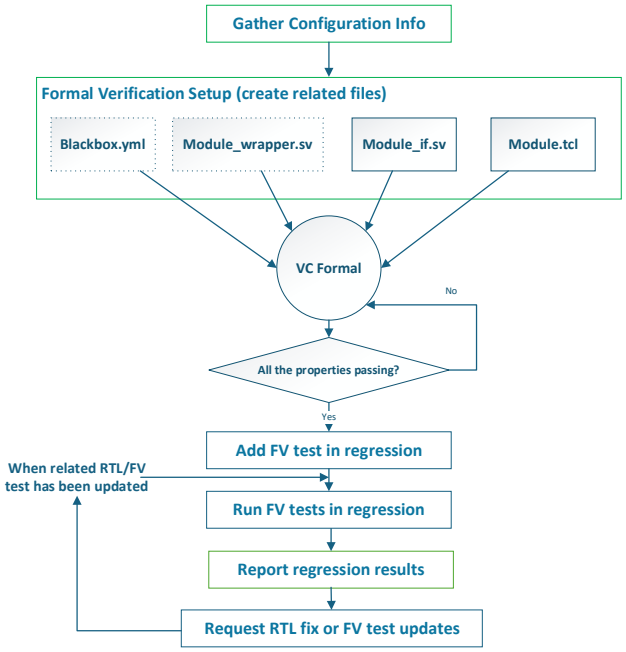

Figure 1. Formal Verification Flow.

After running VCF to verify RTL module, the design verification (DV) engineer has to debug the failed properties by analyzing the definition of the property as well as the constraints to identify the potential RTL issues or the constraint/check problems. Upon successful validation of all assertions, the test is integrated into a regression depot.

The regression is configured to automatically launch FV tests whenever associated RTL modules are updated. This ensures continuous formal verification as the design evolves. Regression results are monitored to confirm that RTL changes are formally verified. If new functionality is introduced, RTL designs are expected to be formally verified on the newest version. The existing properties are to be reviewed or revised with considerations on the constraints update. Once all properties pass, the test remains dormant until further changes are detected. Any failed assertion activates test debug and issue resolution.

To reduce manual effort and improve scalability, we developed Python scripts to automate the generation of setup files and streamline regression reporting. These scripts target the tasks highlighted in the green blocks in Figure 1. Using the script, the required files can be generated within 2 minutes and some applications, such as AEP or FXP, can be run directly after the files generated. Note that the script is applicable not only to FPV but also to other formal applications such as Formal Register Verification (FRV) and Sequential Equivalence Checking (SEQ) [8], etc.

This structured flow ensures consistency, scalability, and traceability across multiple verification cycles and design iterations. Detailed information on the automation strategies will be presented in the next section.

2.2 Automate FV Setup

GitHub Copilot integrated with Visual Studio Code (VS Code) [9] as an extension provides AI-powered code completion across multiple programming languages, especially Python. In VS Code, with the powerful AI assistance, a Python script is developed to automate the tasks highlighted in the "Formal Verification Setup" block in Figure 1. Figure 2 illustrates the control flow of the script developed to automate the formal verification setup.

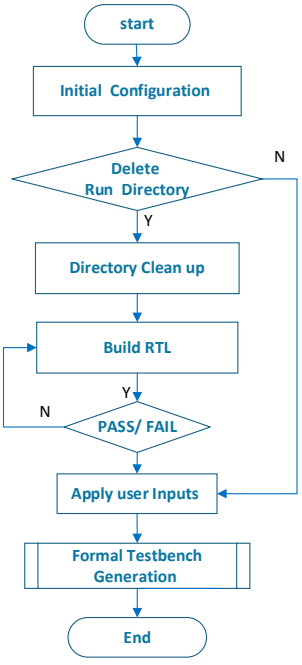

Figure 2. Script Flow.

The automation process begins with the collection of user-defined configuration parameters, which include the module name, clock and reset signal definitions, wrapper usage, and the designated directory for saving formal verification results. These inputs serve as the foundation for initializing the formal environment and guiding subsequent script operations. Based on user inputs, the Python script first determines whether any previous run directories should

be removed to ensure a clean workspace. It then evaluates whether the RTL design requires rebuilding. This is typically triggered when updates have been made to the source files or configuration parameters. If a rebuild is necessary, the script invokes system commands to regenerate the RTL in Verilog format, ensuring that the latest design version is used for verification. Following this, the script proceeds to generate all required components, including the interface file containing SVA, the wrapper module for RTL encapsulation, the Tcl script for tool configuration, and the black box list for excluding modules that are not necessary to be formally verified. In cases where the RTL remains unchanged, the script bypasses the rebuild step and directly generates the required files. This structured flow promotes repeatability, minimizes manual intervention, and enhances consistency across formal verification tasks.

An example execution of the automation script is illustrated in Figure 3, using the module **modeA_sync_fifo** as a case study. During runtime, the script interactively prompts the user to provide essential configuration details, including clock and reset signal definitions, wrapper usage, and other module-specific parameters. The report log file will be saved in the out directory under the corresponding test folder by default and this default setting is not reflected in this interaction with a user. Once the inputs are collected, the script proceeds to automatically generate the formal verification artifacts required for VC Formal execution. These include the files mentioned in section 2.1. This example demonstrates the script's ability to translate user inputs into a complete and reusable formal testbench, aligning with the setup flow previously outlined in Figure 1.

```
python3 src/verif/vcf/scripts/vcf_formal_tb_builder.py
Do you want to delete $SIM directory? (y/n): y
Do you want to delete vcf_runs directory? (y/n): y
✅ Successfully cleaned up $SIM: /proj/version_1/sim
✅ Successfully cleaned up $SIM: /proj/version_1/vcf_run directory
Enter BUILD target: top_component
================================================================================
                BUILD IS COMPLETED : FORMAL SETUP STARTS
================================================================================
Top level RTL module to verify (e.g. mpART): sync_fifo
Has wrapper (Y/N): Y
The BUILD test configuration used for the build command (e.g. from 'make quick'): modeA_sync_fifo
Directory to hold TCL config file (default is moudle_top) -- hit enter to skip:

Enter as many clocks as necessary, starting with the main clock.
The period doesn't have to match reality but the relative period ratios between clocks must be respected.
Clock name (hit enter to skip only if have at least one clock already): clk
Period in ns (number only, no units): 10
Clock name (hit enter to skip only if have at least one clock already):

Enter as many resets as necessary, starting with the main reset. We need at least one.
Reset name (hit enter to skip only if have at least one reset already): reset
Which edge are we using [low/high] ? : high
Reset name (hit enter to skip only if have at least one reset already):
Name of TCL configuration file (default is module_top.tcl)  -- hit enter to skip :
✅ Found module file: /proj/version_1/src/rtl/design_1_0/fifos/sync_fifo.v

Generating interface from context 'sync_fifo'
================================================================================================================
✅ Generated: src/verif/vcf/sync_fifo_top_if.sv
================================================================================================================
Creating /proj/version_1/src/verif/vcf/sync_fifo/sync_fifo.tcl
Creating /proj/version_1/src/verif/vcf/sync_fifo/sync_fifo_if.sv
Creating /proj/version_1/src/verif/vcf/sync_fifo/cc_connectivity.csv
Creating /proj/version_1/src/verif/vcf/sync_fifo/sync_fifo_wrapper.sv
Creating /proj/version_1/src/verif/vcf/sync_fifo/blackbox.yml
```

Figure 3. An Example of Generating FV Test.

2.2.1 Generate Main Function

To implement the flow depicted in Figure 3, we structure the Python script around a **Config()** class containing two key methods: **get_user_input()** and **generate_code()**. The first method gathers project-specific configuration interactively, while the second orchestrates file generation through modular sub-functions. Each sub-function is developed individually using targeted prompts in Copilot, allowing for isolated debugging and efficient integration into the main script. This script (**vcf_formal_tb_builder.py**) creates formal verification environments for RTL modules. Here's a breakdown of the core classes and the methods developed:

- **Config Class**
  - Init (): Initializes configuration with default values for formal verification setup
  - get_input_file_path(): Searches for RTL module files in the build output directory
  - replace_lines_and_generate_module(): Parses RTL files to create SystemVerilog interface files
- **File Generation Functions**
  - get_tcl_cfg_str(): Generates TCL configuration for formal verification tool
  - get_sv_if_str(): Creates SV interface with property placeholders
  - get_wrapper_str(): Produces optional wrapper module for parameter modification
- **Environment Setup Functions**
  - generate_build (): Runs build commands to prepare RTL for verification
  - clean_up (): Removes previous build artifacts and run directories
  - generate_code(): Orchestrates creation of all formal verification files
- **User Interaction Functions**
  - get_user_input(): Collects module details, clocks, resets and configuration preferences
  - get_user_input_clock(): For Multiple clocks
  - get_user_input_reset (): Gathers timing specifications from user
  - write_str (): Creates output files in appropriate directory structure

2.2.2 Generate Files

In VS Code, GitHub Copilot can be guided to generate Python functions using two primary prompt strategies: (1) explicitly describing the required functionality in the prompt, or (2) providing a reference implementation accompanied by a concise prompt. The first strategy is particularly effective for generating structured configuration files such as **Blackbox.yml**. To reduce the complexity of the list in this file, regular expressions are often employed to define exclusion patterns. In this case, only a framework of the YAML file is needed, and the prompt specifies the desired structure and commenting style. For example, we provide a prompt to generate this file:

> **write a python function to generate a blackbox.yml file in which involves following lines:**
> **"# The blackbox collection holds a list of RTL modules we wish to omit**
> **# from the RTL file list. When VC Formal cannot find an RTL definition for**
> **# a module instance it simply blackboxes it.**
> **# We list all RTL files to be commented out, with one file per line. Note**
> **# that the line must start with a hyphen and that there is a space between the**
> **# hyphen and the RTL file name. "**

Copilot then generates a Python function that produces the required file format. This approach minimizes manual effort and ensures consistency, especially when the

excluded modules vary across projects or configurations. Apply this prompt, in VS Code chat window, the function is created by the selected model **Claude Sonnet 4** shown below.

```
def generate_blackbox_yml(output_file_path, rtl_files_to_blackbox=None):
    """
    Generate a blackbox.yml file with the specified header comments and blackbox section.

    Args:
        output_file_path (str): Path where the blackbox.yml file will be created
        rtl_files_to_blackbox (list, optional): List of RTL file names to be blackboxed
    """
    # Header comments for the blackbox.yml file
    header_comments = [
        "# The blackbox collection holds a list of RTL modules we wish to omit\n",
        "# from the RTL file list. When VC Formal cannot find an RTL definition for\n",
        "# a module instance it simply blackboxes it.\n",
        "# We list all RTL files to be commented out, with one file per line. Note\n",
        "# that the line must start with a hyphen and that there is a space between the\n",
        "# hyphen and the RTL file name.\n",
        "blackbox:\n"
    ]

    # Create the blackbox.yml file
    with open(output_file_path, 'w') as blackbox_file:
        # Write the header comments
        blackbox_file.writelines(header_comments)

        # If RTL files are provided, add them to the blackbox section
        if rtl_files_to_blackbox:
            for rtl_file in rtl_files_to_blackbox:
                blackbox_file.write(f"  - {rtl_file}\n")
        else:
            # Add a comment indicating no files are currently blackboxed
            blackbox_file.write("  # Add RTL files to blackbox here, one per line with '- filename' fo

    print(f"Blackbox.yml file generated successfully at: {output_file_path}")
```

Figure 4. An Example Script to generate Blackbox.yml.

Running this Python function, the target blackbox.yml file is generated as depicted in Figure 5.

```
1 # The blackbox collection holds a list of RTL modules we wish to omit
2 # from the RTL file list. When VC Formal cannot find an RTL definition for
3 # a module instance it simply blackboxes it.
4 # We list all RTL files to be commented out, with one file per line. Note
5 # that the line must start with a hyphen and that there is a space between the
6 # hyphen and the RTL file name.
7 # i.e. - ati_*.v
8 blackbox:
9
```

Figure 5. The Created Blackbox.yml.

After the initial generation of the Blackbox.yml file, users can manually append the list of RTL modules to be excluded from formal analysis. This is typically done after line 8, following the example provided on line 7. Completing this list interactively during script execution is inefficient, especially for designs with numerous third-party IPs. Alternatively, blackboxing can be specified directly in the Tcl script using commands such as **set_blackbox -design {cells_name}**. This flexibility allows engineers to tailor exclusion criteria based on project-specific verification requirements.

The second strategy leverages reference-based prompting where an existing .sv file is used as a template. It is applied in generating wrapper and Tcl files. For example, the Module_wrapper.sv file is created by opening a wrapper module and applying a concise prompt (shown below) :

> **Provide a python function to create a general "wrapper" module in SystemVerilog referring to the wrapper.sv.**

This approach enables GitHub Copilot to infer structure and syntax from the reference, producing a reusable wrapper module tailored to the specific RTL configuration.

The generated wrapper module, as an example shown in Figure 6, where variables **DUT_NAME** and **DUT_INST_NAME** are to be defined in Module.tcl, in which the parameter **OP_WIDTH** can be overwritten by a tcl command in a proc **get_vcs_opts{}** in Module.tcl as the following example:

-pvalue rtl_wrapper.OP_WIDTH=12

where parameter rtl_wrapper.OP_WIDTH is obtained from an argument of a proc in Module.tcl. This enabled the reusability of the initiated files. The rest signals on the RTL interface (described in next section) can be added by another function.

```
1 /* This file instantiates the DUT so the instantiation parameters
2  * is configurable.
3  *
4  * File generated by $STEM/src/verif/vcf/scripts/vcf_formal_tb_builder.py.
5  */
6
7 module rtl_wrapper #(parameter OP_WIDTH  = 4) (
8   input                clk,
9   input                reset
10  );
11
12  `DUT_NAME #(.OP_WIDTH (OP_WIDTH)) `DUT_INST_NAME(
13    .clk(clk),
14    .reset(reset)
15    );
16
17 endmodule
```

Figure 6. The Reference for Generating Module_wrapper.sv.

Similarly, a reference file is applied with a short prompt to generate another function for creating Module.tcl file.

2.2.3 Generate Module_if.sv file

Because the interface template is not as simple as the files described above, a Python script is required to be generated. As previously mentioned, RTL modules are scripted using .v.dpl files, which are executed via system commands to generate the corresponding Verilog (.v) files. The creation of the Module_if.sv file requires parsing the Verilog output of the top-level RTL module designated for formal verification. From the parsed outputs, the script extracts key interface elements including the module name, input/output signal declarations, and parameter definitions, and incorporates them into the assertion interface file. Configuration parameters provided during setup are also embedded to ensure alignment with the verification context. In addition to these automatically extracted signals, internal signals, such as control flags or protocol indicators specific to verification, must be manually added when required in property implementation, and the manual work is unavoidable, as it depends on the specific RTL design. This hybrid approach balances automation with flexibility.

```
1  /*****************************************************************************************************************
2  This interface file is auto-generated from rtl_top.v from build $OUT_HOME
3  *****************************************************************************************************************/
4
5  interface rtl_top_if #(
6      parameter DATA_WIDTH = 8,
7      parameter ADDR_WIDTH = 4
8  )(
9      input                   clk,
10     input                   reset,
11     input                   wr_en,
12     input                   rd_en,
13     input [DATA_WIDTH-1:0]  wr_data,
14     input [DATA_WIDTH-1:0]  rd_data,
15     input                   full,
16     input                   empty,
17     input [ADDR_WIDTH:0]    fifo_count
18 );
19     default clocking props_clocking @(posedge clk); endclocking
20     default disable iff(reset);
21
22     //Paramater updated from RTL
23     localparam [1:0] //FSM states
24         ST_A  = 2'b00,
25         ST_B  = 2'b01,
26         ST_C  = 2'b10;
27
28 endinterface
29
30 bind rtl_top  rtl_top_if #(.DATA_WIDTH(PARAM_DATA_WIDTH),
31                            .ADDR_WIDTH(PARAM_ADDR_WIDTH))
32                           rtl_top_if_inst(.*);
```

Figure 7. The Reference for Generating Module_if.sv.

As shown in Figure 7, the example of an interface file generated by the Python script involves the signals on the module port and related parameters, and it is bound with the corresponding RTL module.

2.2.4 Generate other required files

In our design system, the registers are defined in Extensible Markup Language (XML) [10] file, and the exclusions of the registers or some fields are listed in YAML [11] files. The XML file is a register description file which contains the register addresses, fields, size, offset, reset value. Hence, in addition to the files discussed earlier, to formally check registers and construct FRV environment, it requires YAML and XML configuration files. Generating these files manually for each project is repetitive and time intensive. To streamline this process, a Python script is developed to automate the generation/modification of these configuration files by parsing key-value pairs from a database formatted as JavaScript Object Notation (JSON) [12] file containing register names, XML file paths, Register Abstraction Layer (RAL) [13] package locations, and exclusion criteria for specific registers. The sample YAML file and a tcl is given as an input to the script which contains the keys. The script dynamically updates placeholders in YAML and Tcl files with values from the JSON data, handles complex cases like keys with multiple values, and generates context-specific file names for seamless integration. By reducing manual effort and ensuring consistency across projects, the script streamlines the setup process for verification workflows, making it reusable, scalable and efficient for large signal sets or multi-project environments

2.3 Generate Properties

In applying FPV, properties are typically derived from specifications, which are expressed in natural language. AI tools, especially Large Language Models (LLMs) [14] are supposed to have the capability to implement SVAs that are intended to be formally verified. However, based on our best knowledge, majority of AI applications are focusing on the accelerating algorithms implemented underlying the formal tools themselves. For example, a recent work in [15] contributes on formal verification of Common Weakness Enumerations (CWEs) in a dataset of hardware designs written in SV from an AI tool. Another work in [16] proposed a framework to automate program verification.

In this section, two strategies of generating properties automatically with AI-assistance for formal verification purposes are presented.

2.3.1 Properties Generated Based on Diagram

Finite State Machine (FSM) is the best candidate in applying the formal method. The general specification of a design based on the FSM, which serves as a primary format for defining control logic designs, is a diagram. An example of a FSM diagram is shown in Figure 8. This FSM is designed for a sub-module involved in a coprocessor IP. It comprises three states **ST_A**, **ST_B** and **ST_C**. The transitions between these states are governed by combinations of four input signals **a**, **b**, **c**, and **d**. The FSM reacts to these inputs on each clock cycle and moves to the appropriate next state.

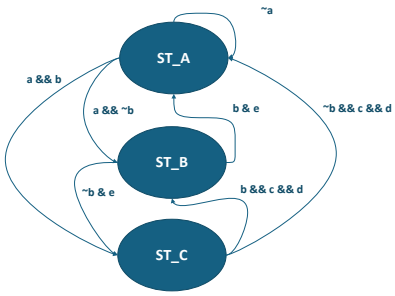

Figure 8. Finite State Machine Example.

Before requesting the SV property code, we feed such a prompt to inform that AI should have a prerequisite of FV concept to indicate the background of the following requests before it starts generating SVA.

> **What kind of properties (written in SystemVerilog) should be designed for formally verifying a finite state machine?**

Copilot uses cutting-edge language models to answer the above question. It lists 6 types of properties that should be designed for formally verifying a general FSM design.

- **State Transition Properties**
  These ensure the FSM transitions correctly from one state to another based on inputs.
- **State Reachability**
  Verify that all states in the FSM can be reached under some input conditions.
- **Illegal State Detection**
  Ensure the FSM never enters undefined or illegal states.
- **Mutual Exclusivity**
  Ensure only one state is active at a time (especially in one-hot encoded FSMs).
- **Timing Constraints**
  Verify that certain states are held for a specific number of cycles or that transitions occur within a time window.
- **Output Behavior**
  Check that outputs behave correctly based on the current state and inputs.

In the AI response, it also proposed that cover properties should be planned for ensuring the formal tool exploring all relevant paths. Based on this answer, we request generating properties for this FSM using the following prompt:

> **Based on what you told me above, generate the SystemVerilog properties for formally verifying the FSM shown in the attached diagram with the state defined as "reg [1:0] stateCur".**

It generates 10 assert properties and 3 cover properties within 1 minute. Note that not all types of properties are generated and the AI-generated properties should be reviewed by engineers before they are employed in the actual verification activities. In our experiments, the generated properties applying mode GPT4-o look to be accurate pretty much. Majority of the properties can be directly used, however, some defined macro in the RTL source file should be manually replaced. This tells us that if the prompts provide enough information, then manual work may not be required.

2.3.2 *Properties Generated Based on Examples*

In practical design environments, most specifications are documented in natural language, often spanning multiple sections or entire documents. This presents a challenge when attempting to use AI tools to generate formal properties, as isolated sentences or short paragraphs rarely provide sufficient context. For example, it is common that an RTL module designed using AXI bus for communication between blocks. Hence, RTL behavior governed by the AMBA AXI and ACE protocol standards [17], which span over 300 pages. The requirements for AXI read and write transactions, including timing, ordering, and security constraints, are distributed across several chapters, making it difficult to encapsulate them in a single prompt.

Rather than describing the full specification, we provide a property as a reference that captures the intended behavior described in the header comments, as shown in Figure 9. The illustrated reference property is designed to verify whether a memory read transaction complies with a defined security policy, where the response originates from an AXI slave. The goal of this assertion described in the comments between line 238 to 242 is supplied for AI understanding the property **req_neg_id_prop**. This example serves as a template for AI-assisted generation of related properties, enabling more accurate and context-aware generation of the required assert properties.

```
238     // ================     req_neg_id property     ================
239     // The goal of this property is the AXI read access security check based on Read Address ID.
240     // The specification requires that ARID[2:0] must be 2, ARID[10:4] must be the defined INIT_ID, and
241     // ARID[20:11] must be the defined UNIT_ID.
242     // ============================================================
243
244     property req_neg_id_prop (AxADDR, AxREADY, AxVALID, AxID, AxUSER, AxLEN, min_addr, max_addr,
245                               xREADY, xVALID, xID, xRESP);
246       int my_txn_id;
247         ((AxREADY && AxVALID &&
248           ((AxID[2:0]!= 3'h2) || (AxID[20:11] != INIT_ID)) &&
249           (AxADDR inside {[min_addr:max_addr]})), my_txn_id = AxID)
250           |=> !((xID == my_txn_id) && xVALID && xREADY)[*0:5] ##1 ((xID == my_txn_id) && xVALID && xREADY) && (xRESP == SLVERR);
251     endproperty // req_neg_id_prop
252
253     // ======== Place your checker instantiations here ======
254     req_neg_id : assert property (req_neg_id_prop (
255                                   .AxADDR  (XBAR__s_axi_araddr  ),
256                                   .AxREADY (s_XBAR_axi_arready ),
257                                   .AxVALID (XBAR__s_axi_arvalid ),
258                                   .AxID    (XBAR__s_axi_arid),
259                                   .AxUSER  (XBAR__s_axi_aruser),
260                                   .AxLEN   (XBAR__s_axi_arlen   ),
261                                   .min_addr(UNIT_ID_START),
262                                   .max_addr(UNIT_ID_END),
263                                   .xREADY  (XBAR__s_axi_rready  ),
264                                   .xVALID  (s_XBAR_axi_rvalid   ),
265                                   .xID     (s_XBAR_axi_rid      ),
266                                   .xRESP   (s_XBAR_axi_rresp    )
267                                                                  ));
```

Figure 9. Example Property Provided in the Prompts.

Providing prompts with this example makes it easier to generate properties related to AXI behaviors than attempting to describe complex requirements directly. More strategies for generating SV code can be found in [18].

2.4 Report FV Test Status

Unlike simulation-based regression, FV tests are only rerun when RTL or specification changes occur. To automate this, a Python script maps each test to its sensitive RTL files via a JSON configuration. Using Perforce [19], the script detects file updates and triggers corresponding FV tests before regression execution, to ensure efficient and targeted verification.

VCF supports logging of test results via commands specified in the Tcl script, typically directing output to a file named vcf.log within a designated directory (e.g., out/). This log includes the number of passed, failed, and inconclusive properties. To eliminate manual review, a Python script was developed—using GitHub Copilot—to parse the log file and report detailed failure information. These results can be automatically emailed to verification engineers and program managers, enabling efficient tracking and debugging across formal verification workflows.

3 Observations

To evaluate the effectiveness of the proposed AI-assisted formal verification workflow, we compared manual and automated efforts across five representative RTL modules: Module A, Module B, Module C, Module D, and Module E. These modules vary in the functionality complexity, signal count, and verification scope, providing a balanced view of the methodology's scalability. Effort estimates across five RTL modules highlight the impact of automation in formal verification workflows. The "Before" and "After" rows in Table 1 represent manual and AI-assisted approaches, respectively.

RTL Block	Number of signals on i/f	Number of properties	Approach	FV setup	Property Development	Estimated Regression Status Report
Module A	15	38	Before	2 hours	2 hours	10 minutes
			After	5 minutes	5 minutes	2 minutes
Module B	27	11	Before	2.5 hours	3.5 hours	10 minutes
			After	5 minutes	1.5 hours	2 minutes
Module C	40	22	Before	3 hours	3.5 hours	10 minutes
			After	5 minutes	10 minutes	2 minutes
Module D	29	13	Before	2 hours	4 hours	10 minutes
			After	5 minutes	10 minutes	2 minutes
Module E	298	22	Before	4 hours	9 hours	10 minutes
			After	5 minutes	10 minutes	2 minutes

Table 1. Comparison of Efforts in Formal Verification.

Manual FV setup involves writing configuration files, debugging syntax, and executing builds. Obviously, effort is scaled with the interface complexity. Property development is time-intensive, especially for modules with protocol-driven behavior (i.e. AXI). The estimated time in the table above is based on the performance of a general junior engineer, who has formal verification concept and is familiar with SV property development skills. The complexity and number of assertion properties significantly influence the effort required for their development. In verifying module E case, which involves more complex AXI behavior with numerous signals and logical considerations, the time required for human development surpasses even that of simpler cases like module E. Moreover, properties written manually by engineers are susceptible to syntax errors and typos, whereas AI-generated code generally avoids such issues. Consequently, the debugging time associated with AI-generated properties may be lower compared to manually written code.

Regression reporting requires manual log inspection and status communication. Note that the time required for analyzing FV test status by reviewing summary messages in the vcf.log file remains consistent, regardless of the RTL block size.

In contrast, the automated flow uses Python scripts and AI-generated code to streamline these tasks. Setup time is reduced to approximately 5 minutes per module, property generation is accelerated via prompt-driven AI-assistant, and regression results are parsed and reported automatically.

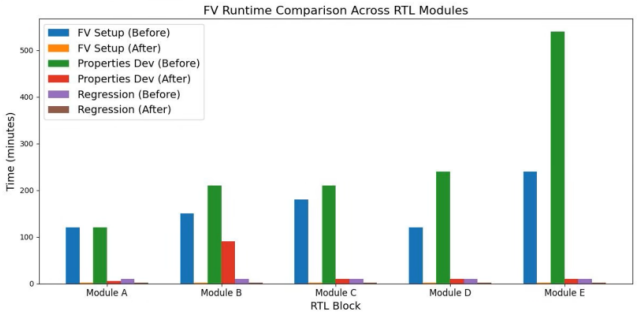

Figure 10. FV Runtime comparison across RTL modules

To highlight the reduced work time/efforts, in Figure 10, the impact of applying the proposed framework or not is visualized in a bar chart across the five modules, based on the data shown in Table 1. The chart clearly demonstrates significant reductions (the orange, red and brown colored bars for each module) in verification effort enables faster iteration and improves scalability across diverse design blocks in the project. Recently, the method has been successfully employed in multiple projects.

4 Conclusions

This work demonstrates the AI-assisted automation in formal verification workflows. Supplying prompts in VS Code, we make use of GitHub Copilot to generate efficiently Python script for environment setup, interface construction, and regression reporting tasks that traditionally demand significant manual effort. Additionally, SystemVerilog properties can be automatically derived from FSM diagrams or references with brief prompts. The proposed workflows significantly improve work efficiency in the verification process. Beyond FPV and FRV applications, this approach can extend to initiating a range of VCF tools, including Sequential Equivalence Checking (SEQ), Connectivity Checking (CC), and Data Path Verification (DPV). The strategies presented are also applicable for formal verification using other tools, i.e. JasperGold or Questa, etc.

Our findings highlight the untapped potential of AI tools in formal verification, a domain where automation remains limited despite broader industry adoption. As a forward-looking extension, we aim to extract formal properties directly from contextual specifications embedded in design documents. This direction supports the development of intelligent agents capable of improving assertion quality, accelerating verification cycles, and strengthening design confidence in complex semiconductor systems.

Acknowledgments

The authors thank the AMD VMT team, especially Prashant Desai, Anna Safonov, and Allen Bao, for technical support. We acknowledge Tony Wu and former MP team member Simon Robidas for building the formal verification task library and appreciate Ali Moussa and Gagan Chandra Basavaraju for developing scripts to launch and report FV regressions.

References

[1] Formal Verification, Wikipedia, https://en.wikipedia.org/wiki/Formal_verification, August 2025
[2] Synopsys, VC Formal User Guide, Version W-2024.09-SP2, March 2025.
[3] Jasper Formal Verification Platform, Cadence, https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html, August, 2025
[4] Questa Formal, Siemens, https://eda.sw.siemens.com/en-US/ic/questa/formal-verification, August 2025
[5] GitHub, https://github.com/, July 2025
[6] Microsoft Copilot, https://copilot.microsoft.com/chats/bnj5nzHBpZQujfTXtHyes, July 2025
[7] Regular Expression, https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference, July 2025
[8] VC Formal User Guide, Version W-2024.09-SP2, March 2025
[9] Visual Studio Code, https://code.visualstudio.com/, August 2025.
[10] XML, https://aws.amazon.com/what-is/xml/, August 2025
[11] YAML, https://yaml.org/, August 2025.
[12] JSON, https://www.json.org/json-en.html, August 2025.
[13] RAL, UVM Register Model, https://verificationguide.com/uvm-ral/introduction-to-uvm-ral/, August 2025
[14] Large Language Model, https://aws.amazon.com/what-is/large-language-model/, August 2025
[15] Xujie Si et.al., "Code2Inv: A Deep Learning Framework for Program Verification", CAV, 2020.
[16] Deepak Narayan Gadde et.al., "All Artificial, Less Intelligence: GenAI through the lens of Formal Verification", March 2024
[17] AMBA® AXI™ and ACE™ Protocol Specification, ARM, https://documentation-service.arm.com, August, 2025
[18] Prompt engineering cheat sheet, https://codesignal.com/blog/prompt-engineering/prompt-engineering-cheat-sheet/, August 2025
[19] Perforce https://www.perforce.com/, August 2025.