Saarthi: The First AI Formal Verification Engineer

Aman Kumar[1], Deepak Narayan Gadde[2], Keerthan Kopparam Radhakrishna[2], Djones Lettnin[3]
[1]Infineon Technologies Semiconductor India Private Limited, India
[2]Infineon Technologies Dresden GmbH & Co. KG, Germany
[3]Infineon Technologies AG, Germany

Abstract

Recently, Devin has made a significant buzz in the Artificial Intelligence (AI) community as the world's first fully autonomous AI software engineer, capable of independently developing software code [1] [2]. Devin uses the concept of agentic workflow in Generative AI (GenAI), which empowers AI agents to engage in a more dynamic, iterative, and self-reflective process. In this paper, we present a similar fully autonomous AI formal verification engineer, Saarthi[1], capable of verifying a given RTL design end-to-end using an agentic workflow. With Saarthi, verification engineers can focus on more complex problems, and verification teams can strive for more ambitious goals. The domain-agnostic implementation of Saarthi makes it scalable for use across various domains such as RTL design, UVM-based verification, and others.

Index Terms

Generative AI, Agentic AI, Formal Verification, Saarthi

I. Introduction

Hardware design verification, especially formal verification, entails a methodical and disciplined approach to the planning, development, execution, and sign-off of functionally correct hardware designs. Formal verification uses mathematical methods to prove the correctness of hardware designs against their specifications, ensuring that all possible states and inputs are considered, which complements traditional simulation-based verification techniques that might only cover a subset of possible scenarios due to practical constraints. [3]. The formal verification process encompasses several key roles, such as organizational coordination, task allocation, code development, property proving, analyzing Counter Examples (CEXs), debugging, coverage closure, and documentation preparation. These roles are crucial for managing the complexity and ensuring the thoroughness of the verification process. For instance, analyzing counterexamples involves identifying specific scenarios where the design might fail to meet its specifications, which is critical for debugging and refining the design. This highly intricate activity demands meticulous attention to detail, given its long development cycles and the critical nature of ensuring hardware functionality and reliability [4].

The field of Natural Language Processing (NLP) has undergone a significant transformation with the advent of Large Language Models (LLMs). These powerful models, often referred to as GenAI, have revolutionized how machines understand and generate human language, enabling unprecedented advancements in a wide array of applications [5]. Through extensive training on large datasets using the "next word prediction" approach, LLMs have demonstrated remarkable capabilities in various downstream tasks such as context-sensitive question answering, machine translation, and code generation [6]. Interestingly, the primary components of formal verification – specifically code (assertions as properties) and specification documents – can be considered as forms of "language" or sequences of characters [7]. Various surveys [8] have discussed techniques for improving conventional formal verification; however, we aim to enhance it further using AI. This paper introduces an end-to-end formal verification process driven by LLMs. This process encompasses design specification analysis, code development, verification, and document generation, aiming to establish a unified, efficient, and cost-effective paradigm for hardware design verification. By leveraging the advanced capabilities of LLMs, it is possible to streamline the formal verification process, enhancing both accuracy and productivity.

Like every other semiconductor company, we wanted to investigate the possibilities of using GenAI for dedicated use cases. However, there are known challenges that prevent precise use case definitions [9]. GenAI operates as a stochastic process, meaning it generates non-deterministic output with each regeneration. This is in contrast to the requirements of hardware design verification, which demands precise, deterministic answers, particularly in formal verification where engineers need to make clear pass/fail decisions based on exact criteria. Due to this fundamental mismatch, the non-deterministic output of LLMs is not always suitable for hardware design verification. Additionally, LLMs can suffer from artificial Attention Deficit Hyperactivity Disorder (ADHD), characterized by a tendency to lose focus on the task at hand, and hallucination, where the model generates incorrect or nonsensical information confidently [10]. These issues are very prominent and can lead Generative Pre-trained Transformer (GPT) users to get stuck in iterative loops, repeatedly seeking accurate results without success. Given these challenges, the current state of GenAI may not be well-suited for applications that require the high precision and determinism essential in hardware design verification.

[1]Saarthi is a Sanskrit word that means someone who guides and leads you to your destination.

To overcome the aforementioned challenges and use GenAI for problem-solving, we introduce a fully autonomous AI formal verification engineer, Saarthi, capable of verifying a given RTL design end-to-end using an agentic workflow. Saarthi stands for ScalAble ARTificial and Human Intelligence that uses agentic AI based reasoning patterns and human intelligence to produce sensible results. Saarthi can run formal verification on several complex IPs with coverage closure and find bugs similar to a human verification engineer. Our contributions to this work are summarized below:

- We propose Saarthi, an agentic AI-based formal verification engineer. By providing the design specification, Saarthi sequentially handles verification planning, generating SystemVerilog Assertions (SVAs), proving the properties, analyzing CEXs, and analyzing the formal coverage for sign-off.
- To address the issue of ADHD and code hallucination, we use the approach of agentic (few-shot) workflow as opposed to the non-agentic (zero-shot) workflow.
- To further alleviate potential challenges related to LLMs getting stuck in iterative loops, we use the concept of "human-in-the-loop" AI to ensure uninterrupted end-to-end formal verification.

Section II summarises the related work and introduces agentic workflow design patterns. Section III discusses the details about the framework of AI agents and how they interact with each other to perform formal verification. Section IV presents our results from evaluating formal verification of different RTL designs with varied complexity. Section V concludes the paper with an outlook on possible future research opportunities.

II. Background

Recent work in [10] found that LLMs tend to generate incorrect designs and are vulnerable to security flaws as the authors observed around $60\,\%$ failure rate for the generated RTL designs. Our findings indicated that the expectations of authors from AI were too high in terms of achieving first-time-right designs. Additionally, their approach of benchmarking the LLM-generated designs may have been overly pessimistic, possibly not accounting for the iterative improvement potential of these models. Moving forward, we want to focus on using GenAI as a problem-solving tool and to use the existing capabilities of LLMs to generate better results. There are two types of AI-based workflows:

- Non-agentic workflow (zero-shot)
- Agentic workflow (few-shot)

A. Non-Agentic Workflow

The first productive uses of LLMs involved non-agentic workflows, where we type a prompt and the model generates an answer in one go. This is akin to asking a person to write an essay on a topic and saying "please sit down to the keyboard and type the essay from start to finish without ever using backspace". Despite how hard this is, LLMs do it remarkably well; however, the quality of the generated content is often relatively lower due to the lack of iterative refinement. This approach is termed a zero-shot or non-agentic workflow. In a zero-shot scenario, the model attempts to generate a response without any prior specific examples or iterations tailored to the task at hand.

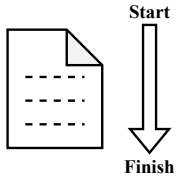

Fig. 1: Non-agentic workflow

In the ReFormAI dataset paper [10], the authors used a similar non-agentic workflow and benchmarked the LLM generated RTL codes that resulted in a relatively higher failure rate. The results suggested that the failure rate was significant due to the one-pass, zero-shot nature of the generation process. The results would likely have been better if a feedback loop had been added to the generation part. A feedback loop would allow for iterative refinement, where the model could receive feedback on its initial outputs and make adjustments to improve accuracy and quality. This approach would enable the LLM to correct errors, incorporate additional context, and ultimately produce higher-quality RTL designs.

B. Agentic Workflow

In contrast to the zero-shot workflow, the agentic or few-shot workflow uses iterative loops and feedback to produce better results. This approach is very similar to how humans think and approach a given task. For the task of writing an essay, a human would typically start by outlining the essay on topic X, conducting web research, preparing a first draft, considering what parts of the essay need revision, revising the draft, and finally producing the final version. Similarly, if LLMs employ this iterative approach to address a prompt, they deliver remarkably better results. In a few-shot workflow, the model is initially provided with a few examples to guide its responses. As it generates outputs, it receives feedback, which it uses to refine and improve its responses iteratively. This process allows for error correction and the incorporation of additional context, leading to higher quality and more accurate results compared to the zero-shot approach.

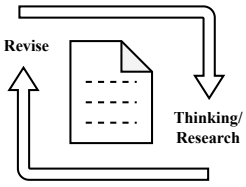

Fig. 2: Agentic workflow

Based on open-source benchmarks, researchers found that using GPT-3.5 with zero-shot prompting, the LLM yields $48\,\%$ correct results. With GPT-4, this accuracy improves to $67\,\%$. However, when using an agentic workflow and wrapping it around

GPT-3.5, it outperformed GPT-4, demonstrating the significant impact of iterative feedback and refinement [11]. Certainly, an agentic workflow wrapped around GPT-4 produced even better results, further enhancing accuracy and performance. We also tried a similar approach to formally verify a synchronous FIFO and achieved a 100 % pass rate using the few-shot approach within minutes. Table I summarizes the result.

TABLE I: Zero-shot and few-shot prompting for formal verification of a synchronous FIFO design

Workflow	Proved Assertions	CEX	Unreachable Covers	Covered Covers
Zero-shot	42.85 %	57.15 %	12.5 %	87.5 %
Few-shot	100 %	0 %	0 %	100 %

Researchers have recently put a lot of effort into defining agentic reasoning design patterns. The most significant ones that facilitate agentic AI-based workflows are:

- Reflection [12] [13]
- Tool use [14] [15]
- Planning [16] [17]
- Multi-agent collaboration [6] [18]

C. Reflection

Reflection uses the concept of a coder agent and a critic agent. For any given task, there would be a coder agent that generates code—in our case, SystemVerilog Assertion (SVA) – and a critic agent that critically analyzes and reviews the output of the coder agent, providing feedback. This feedback loop is iterative, allowing the coder agent to refine its code based on the critic agent's insights, leading to progressively better results.

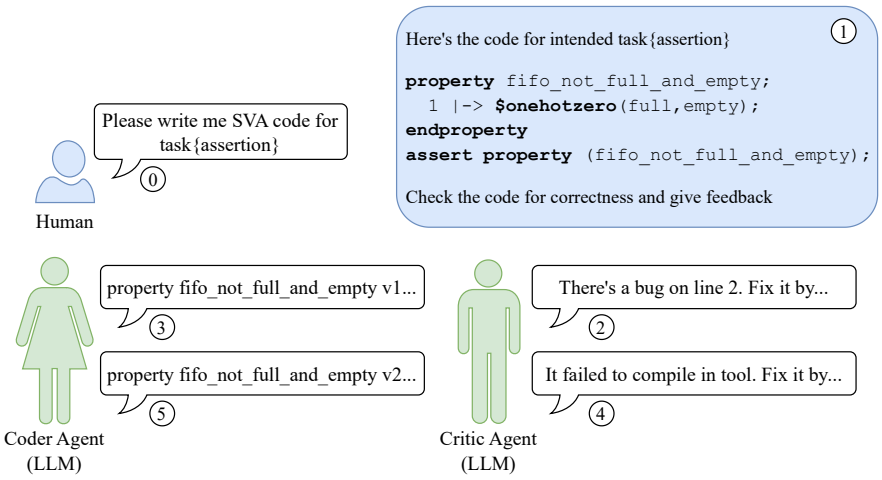

Fig. 3: Coder and critic AI agents for self-reflection (adapted from [11])

Fig. 3 shows a case where a human asked the coder AI agent to write SVA code for a given specification. Once the SVA is generated, the critic agent analyzes the code and provides feedback, identifying a bug in line X. The coder agent then uses this feedback to fix the code and generate version 1 (v1) of the SVA. Next, the critic agent attempts to compile the generated code using a formal verification tool and reports a compilation issue, including the error message from the tool, to the coder agent. The coder agent analyzes this feedback and produces a corrected SVA as version 2 (v2). Using this iterative approach, the human is able to obtain a significantly better SVA by leveraging the capabilities of existing LLMs. The human's role includes initiating the process, reviewing the iterations, and making use of the final, refined SVA code.

LLM agents are increasingly being used to interact with external environments as goal-driven agents. However, these language agents face difficulties in rapidly and effectively learning through trial-and-error, since conventional reinforcement learning techniques necessitate a large number of training samples and expensive model fine-tuning. The authors in [13] propose a novel framework, Reflexion, that uses verbal reinforcement to help agents learn from previous failures. Creating valuable reflective feedback is difficult because it involves accurately identifying where the model went wrong (known as the credit assignment problem [19]) and being able to produce a summary that offers actionable recommendations for improvement. The authors also propose several mitigation techniques such as simple binary environment feedback, pre-defined heuristics for

common failure cases, and self-evaluation such as binary classification using LLMs (decision-making) or self-written unit tests (programming).

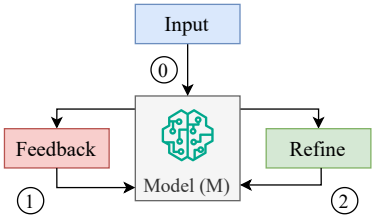

Fig. 4: Iterative feedback with self-refinement [12]

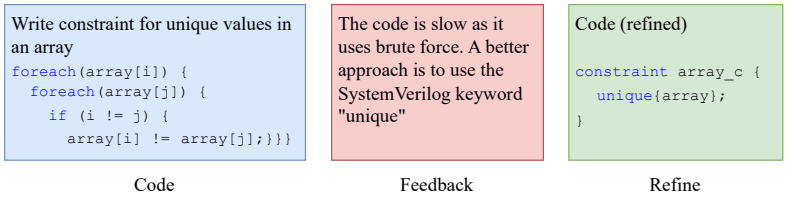

Fig. 5: Example of self-refinement: an initial output is generated by the base LLM and then passed back to the same LLM to receive feedback and sent to the same LLM to refine the output.

The authors in [12] introduce iterative self-refinement, a fundamental characteristic of human problem-solving that involves creating an initial draft and subsequently refining it based on self-provided feedback. A similar approach can be applied to LLM agents as shown in Fig. 4. Given an input (0), self-refinement starts by generating an output and passing it back to the same model *M* to get feedback (1). Feedback is passed back to *M*, which refines the previously generated output (2). Steps (1) and (2) iterate until a stopping condition is met. An example of such a self-refinement is highlighted in Fig. 5. Using self-feedback, the LLM agent could modify the previously generated code for a constraint in SystemVerilog to generate unique values in a random array to an optimized version.

D. Tool Use

Tool use is another agentic reasoning design pattern that leverages AI-based tools to generate meaningful results. Web search tools such as Copilot, which assists with coding by providing intelligent code suggestions, and code execution tools such as [20] and [15], which help automate and streamline the coding process, significantly increase productivity gains. These tools enable users to accomplish tasks more efficiently by harnessing the power of AI to provide relevant information, automate repetitive processes, and enhance overall workflow. Some examples of areas of application for tool use are mentioned in Table II.

TABLE II: Tool use in agentic workflow [11]

Analysis	**Research**	**Productivity**	**Images**
Code Execution	Search engine	Email	Imager generation (e.g., Dall-E)
Wolfram Alpha	Web browsing	Calendar	Image captioning
Bearly Code Interpreter	Wikipedia	Cloud Storage	Object detection

The authors in [21] used GPT tools to analyze inductive-step failure CEXs in formal verification to generate helper assertions. A similar approach could be employed to analyze the CEXs and request an explanation of the failure. Based on this analysis, the user can decide whether the RTL design needs a fix or if the property should be adjusted in the case of a false-positive. An example of such a conversation between a human and the LLM is highlighted in Fig. 6, demonstrating how the LLM provides insights that aid the user in making these decisions.

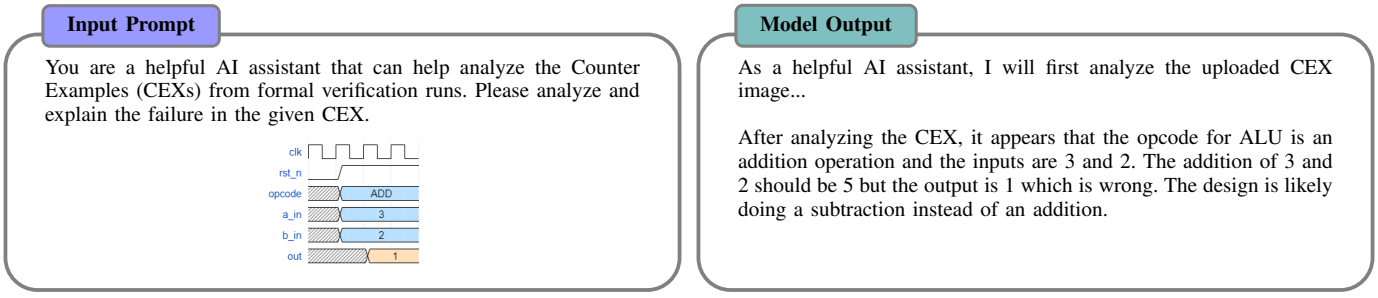

Fig. 6: Using GenAI tool to analyze a CEX

E. Planning

Planning is one of the key workflows to generate quality output from LLMs using agentic reasoning. Even though LLMs are quite good at generating responses for a given prompt, they are often criticized for their lack of commonsense reasoning. This deficiency can impact the accuracy and reliability of the generated responses. Fig. 7 illustrates this issue through a standard prompting example where the model's output is incorrect for a simple reasoning question. By incorporating planning into the workflow, LLMs can be guided to consider broader context and logical steps, which helps mitigate the lack of commonsense reasoning and leads to higher quality outputs.

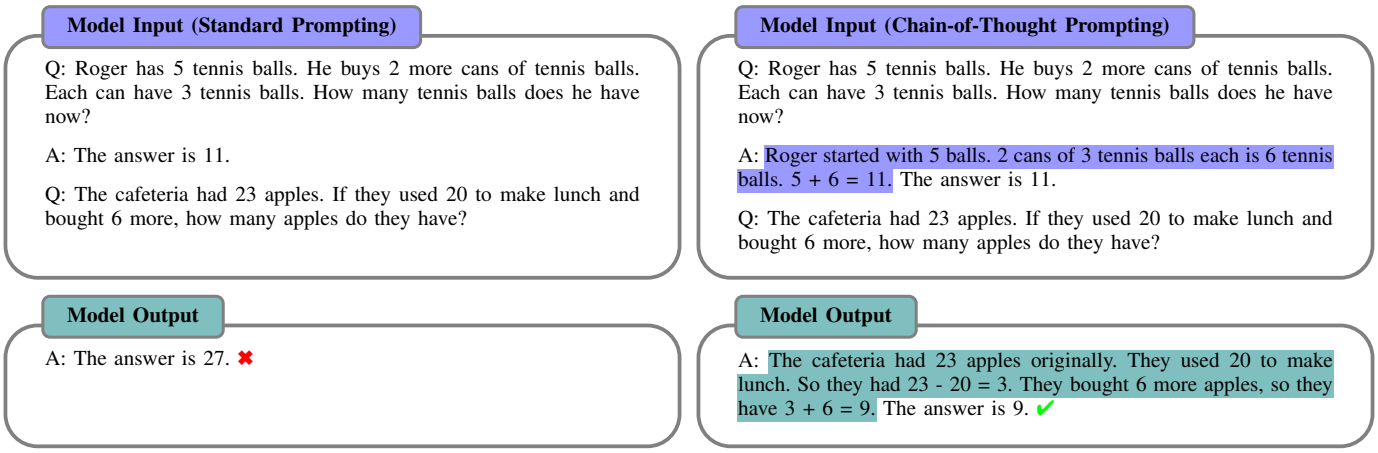

Fig. 7: Standard prompting vs. chain-of-thought prompting. Chain-of-thought reasoning processes are highlighted [16].

The authors in [16] explore generating a Chain-of-Thought (CoT) – a series of intermediate reasoning steps that enable LLMs to tackle complex arithmetic, commonsense, and symbolic reasoning tasks. CoT, in principle, allows models to decompose multi-step problems into intermediate steps, which means that additional computation can be allocated to problems that require more reasoning steps. For many reasoning tasks where standard prompting has a flat scaling curve, CoT prompting leads to dramatically increasing scaling curves. An example of CoT prompting is shown in Fig. 7 that elicits reasoning in LLMs.

Although CoT emulates the thought process of human reasoners, this does not necessarily indicate that the neural network is actually "reasoning". CoT typically involves few-shot prompting, where the model is provided with a few examples to guide its responses. This approach can be expensive, especially when using paid LLMs. In contrast, using zero-shot prompting with a more generalized prompt could be more cost-effective. Furthermore, there is no guarantee that CoT will follow correct reasoning paths, which can lead to both correct and incorrect answers. The variability and uncertainty in the reasoning process mean that while CoT can help generate more logically structured responses, it can also propagate errors if the initial reasoning path is flawed.

F. Multi-Agent Collaboration

AI agents can collaborate to solve tasks given by a human. These agents can leverage several LLMs to handle different responsibilities within a complex task. The authors in [6] introduce communicative agents for software development, which are designed to interact and share information to improve task-solving efficiency, and present an open-source alternative to Devin [1]. Research done in [22] supports the notion that a multi-agent system performs better than a single agent when solving complex tasks. Table III summarizes the results of a multi-agent debate for different tasks, such as the Massive Multitask Language Understanding (MMLU) benchmark and chess moves, demonstrating the improved performance of multi-agent systems in these diverse scenarios.

TABLE III: Multi-agent debate [22]

Task	Single Agent	Multi-Agent
Biographies	66.0 %	73.8 %
MMLU	63.9 %	71.1 %
Chess move	29.3 %	45.2 %

A classic example of multi-agent collaboration is depicted in Fig. 3, where the coder and critic agents work together to solve a given task that includes reasoning and feedback. In this scenario, the coder agent generates the code, while the critic

agent reviews the output and provides feedback to improve accuracy. To compensate for problems such as code hallucination – where an AI generates plausible but incorrect code – it is usually better to divide a complex task into simpler tasks and have one agent solve each of them [23]. This approach not only reduces the risk of hallucination but also mitigates the risk of an agent getting stuck in an iterative loop while attempting to solve a complex task. By breaking down the task, each agent can focus on a specific aspect, leading to more efficient and accurate problem-solving.

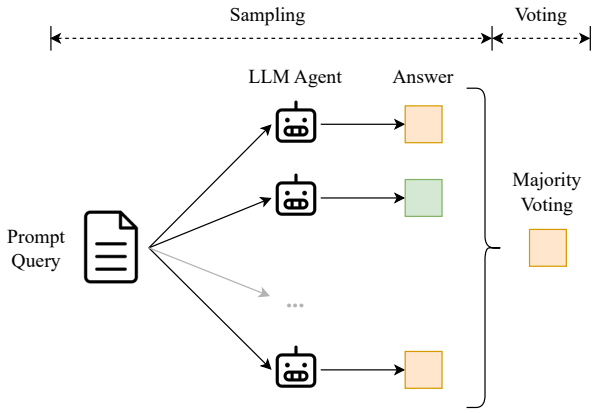

Fig. 8: Sampling-and-voting method [23]

The authors in [23] suggest the so-called "sampling-and-voting" method to improve results from multiple LLM agents. This approach involves generating results from multiple LLM agents for the same prompt (sampling) and then voting on the majority result to obtain the best possible outcome. This method leverages the diversity of responses to enhance accuracy and reliability. The performance of LLMs scales with the number of agents instantiated, meaning that as more agents are used, the overall accuracy and quality of the results improve. This scaling effect is highlighted in Fig. 8, which demonstrates how increasing the number of agents leads to better performance metrics such as accuracy, consistency, and robustness of the outputs.

III. Saarthi: Agentic AI-Based Formal Verification

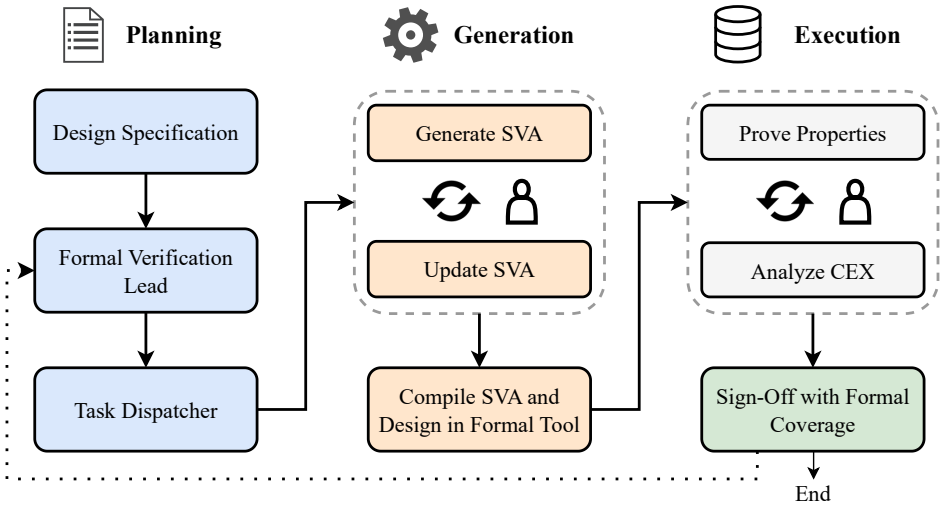

Fig. 9: Saarthi: Agentic AI based formal verification using multi-agent collaboration

To realize our contributions and conduct our experiments, we prepared a flow as illustrated in Fig. 9 where AI agents are in the driver's seat as soon as a task is given to solve. Saarthi is designed to facilitate formal verification through a sophisticated agentic AI-based approach that leverages multi-agent collaboration. Saarthi integrates several design patterns, including agentic reasoning and techniques to mitigate issues such as attention deficits, hallucinations, and repetitive looping. It is built using

three open-source frameworks – CrewAI, AutoGen, and LangGraph – enabling the user to select any of them for formal verification. The architecture implements a configurable agent orchestration system that can be adapted to varying verification requirements while maintaining consistency and reliability in the verification process.

A. *Flow Architecture*

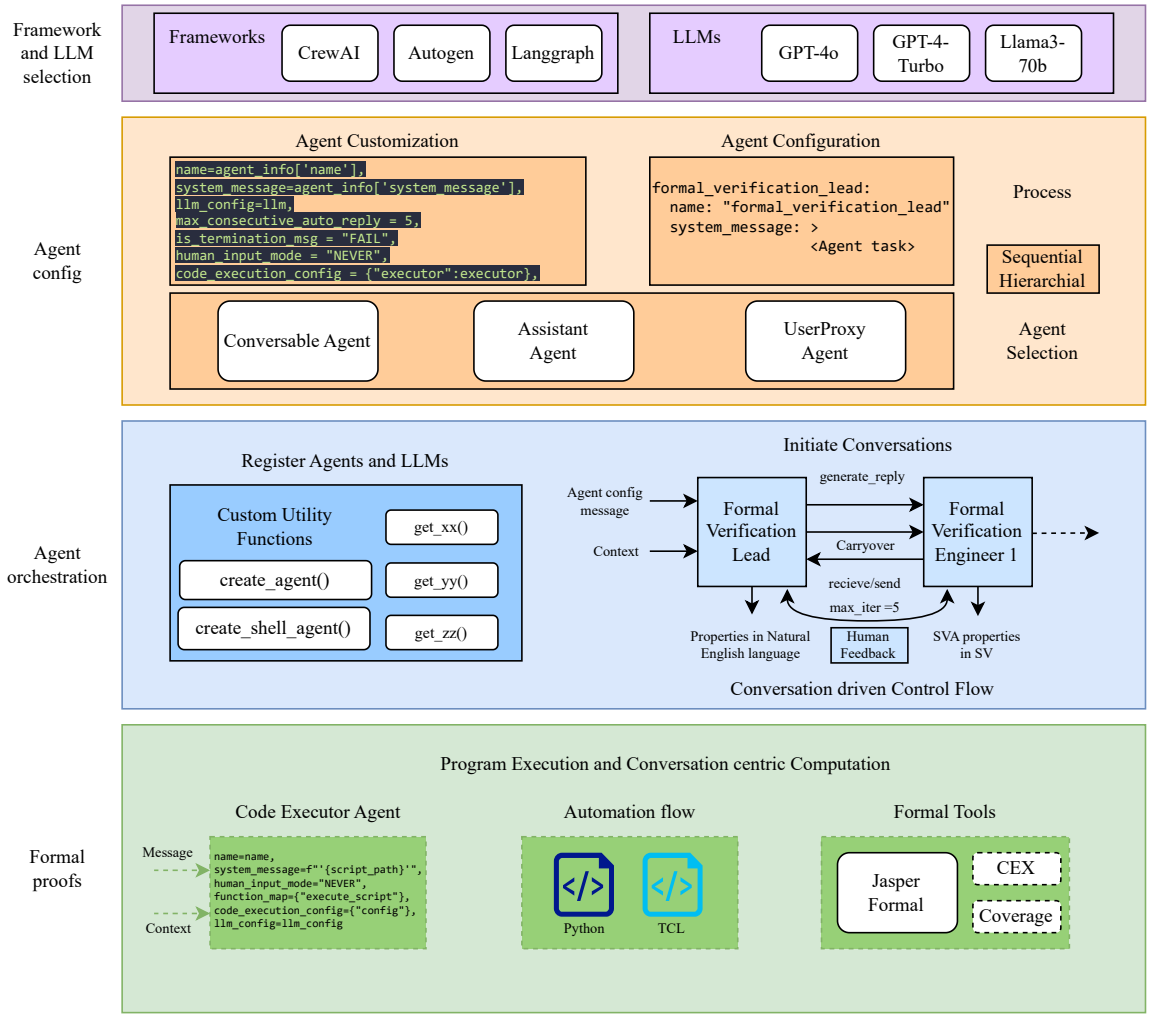

Fig. 10: Example usage of Saarthi for formal verification using multi-agent conversation

The system's entry point is the main function, which implements an argument parsing system supporting multiple configuration domains, including framework selection mechanisms and LLM model specifications. The framework selection encompasses implementations such as CrewAI, AutoGen, and LangGraph, while the LLM configuration supports models including GPT-4o, GPT-4-Turbo, and Llama3-70B. This modular architecture enables seamless switching between different implementation frameworks while maintaining consistent verification workflows. This main function also processes input files, which consist of a specification file for the input design. Each framework has an agent configuration file that defines the agents, including their roles and descriptions. The collaborative formal verification process is realized through a structured crew of agents, each specializing in a distinct aspect of verification.

The AutoGen implementation features a sequential processing pipeline that manages inter-agent communication through structured message-passing protocols. The system implements comprehensive logging mechanisms that capture detailed interactions and logs. The CrewAI implementation utilizes decorator patterns for task definition, implementing a structured approach to workflow management. The system incorporates file-based tool integration mechanisms and implements a comprehensive logging system that captures execution details at multiple granularity levels.

The framework also implements an error management system incorporating multiple feedback loops for continuous improvement. The system utilizes critic agents that perform property evaluations, providing detailed feedback for assertion improvement. The iteration control system uses threshold-based monitoring to prevent infinite loops, automatically triggering human intervention when resolution cannot be achieved autonomously. The error-handling system implements structured exception management through try-catch hierarchies. Saarthi also generates verification artifacts through template engines and formatting systems. The logging system implements timestamp-based organization and multi-level capture, ensuring complete traceability of the verification process. Fig. 10 shows an example usage of Saarthi to accomplish a task using multi-agent collaboration.

Fig. 9 is the high-level overview of the flow we have defined for formal verification where AI agents are in the driver's seat as soon as a task is given to solve. Every block in the flow chart is executed by an LLM, except the first, the design specification that a human provides. The first LLM agent is the so-called "formal verification lead" who is responsible for generating a Verification Plan (vPlan) (i.e., a list of the properties necessary to verify the given design written in the natural English language) based on the given specification. The verification lead divides the follow-up tasks to other LLM agents. An example of such an agent in the AutoGen framework with its role, goal and backstory is highlighted in Listing 1. The subsequent steps in the flow involve several LLMs acting as formal verification engineers to analyze the vPlan and generate SVA for each corresponding element. An example of the tasks defined in the AutoGen framework for vPlan generation and SVA generation is highlighted in Listing 2. The generated properties are evaluated for correctness by the critic agents and feedback is provided to improve SVA. This iterative process continues until a threshold is reached without a conclusion on the SVA. This is when human intervention is required to decide the correct SVA and continue the overall process. The generated properties are then proven in a formal tool, and the CEXs are analyzed if any and fixed by the LLM agents. Once all properties are proven, another LLM agent takes over to analyze the formal coverage, an important verification sign-off criteria. Based on missing coverage, feedback is provided to the formal verification lead to add missing properties.

Listing 1: Formal verification lead agent example

```
formal_verification_lead:
  role: >
    Formal Verification Lead
  goal: >
    Gather all the necessary information regarding the given Register Transfer Level (RTL) design and
    its specification to define a set of formal properties to verify its functionality.
  backstory: >
    An expert formal verification engineer, who spends all day and night thinking about how to verify
    the given Design Under Verification (DUV) based on its specification. It analyzes the design
    specification and defines the set of SystemVerilog Assertion (SVA) properties required to verify
    the functionality of the given design. The properties are described in the natural English
    language.
  allow_delegation: false
  verbose: true
  max_iter: 5
```

Listing 2: Example of tasks defined for formal verification lead and the SVA generation responsible engineer

```
vplan_gen :
  description: >
    Analyze the specification of the Register Transfer Level (RTL) design to come up with a set of
    properties written in the natural English language that would be used to verify the functionality
    of the design using formal verification.
  expected_output: >
    SystemVerilog Assertion (SVA) properties defined in the natural English language that will be used
    to verify the functionality of the given design using formal verification.

property_gen :
  description: >
    Analyze the given property description and write the SystemVerilog Assertion (SVA) property.
  expected_output: >
   Correct SystemVerilog Assertion (SVA) for each of the properties.
```

Since ADHD, hallucination and being stuck in iterative loops are the usual downsides of LLMs, we circumvent these issues by dividing complete verification into smaller tasks. We also make sure that if there is a feedback loop between two agents, say, to generate and update the SVA, and the loop cannot decide on the correct SVA, we have set a threshold of 5 iterations, after which the human would be asked to intervene and provide the correct SVA to move forward. We call this approach "human-in-the-loop AI" that uses human feedback to ensure the entire flow is exercised and does not get stuck at any point. It plays a key

role in making the models more truthful and reducing hallucination errors and allows human feedback to steer agents in the right direction, specify goals, and others. The human-in-the-loop component sits in front of the auto-reply components. It can intercept the incoming messages and decide whether to pass them to the auto-reply components or to provide human feedback based on customization in the agentic AI frameworks. Fig. 11 illustrates such a design. The algorithm for human-in-the-loop AI is highlighted in algorithm 1.

Algorithm 1 Human-in-the-loop message processing in Saarthi

Require: M: Messages, $mode \in \{NEVER, TERMINATE\}$, $max\_replies \in N^+$

Initialize $counter \leftarrow 0$, $conversation\_active \leftarrow true$
while conversation_active **do**
  $m_t \leftarrow$ ReceiveMessage()
  if IsTerminationMessage(m_t) **then**
    conversation_active $\leftarrow false$
  else if mode = NEVER **then**
    ProcessAutoReply(m_t)
  else if mode = TERMINATE **then**
    if counter $\geq$ max_replies **then**
      $h_c \leftarrow$ RequestHumanInput(m_t)
      if h_c = TERMINATE **then**
        conversation_active $\leftarrow false$
      else if h_c = SKIP **then**
        ProcessAutoReply(m_t)
      else if h_c = INTERCEPT **then**
        ProcessHumanReply(m_t)
        counter $\leftarrow 0$
      end if
    else
      ProcessAutoReply(m_t)
      counter $\leftarrow$ counter + 1
    end if
  end if
end while
Ensure: Counter resets on INTERCEPT; One reply per message

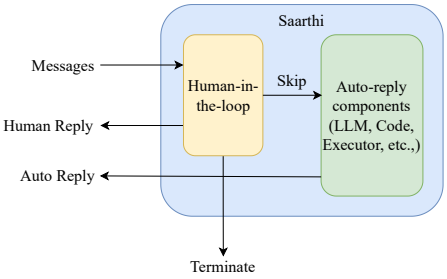

Fig. 11: Allowing human feedback in agents [18]

The algorithm starts by setting the conversation to active and the counter to 0. When the mode is set to NEVER, it doesn't wait for human input and terminates automatically. However, when the mode is set to TERMINATE, it requests humans to give some input h_c. If h_c is terminated, then the human-in-the-loop process terminates. If h_c is skipped, then the human-in-the-loop process proceeds forward (e.g., to other agents) without terminating the process. If h_c is intercepted, the algorithm considers and processes the human input.

B. Agent Orchestration

Once the agents are configured, they are passed as keys to an orchestration setup. This orchestration mechanism is capable of managing agents in both a sequential and hierarchical manner. For this formal verification process, the agents are arranged in a sequential order. After orchestration, the selected framework's main module initiates the verification process, invoking the agents sequentially.

During this process, the agents generate key artifacts such as vPlan and properties, logging their interactions and collaborations as they proceed. The generated properties are also subjected to evaluation by critic agents, who provide feedback to improve the quality and correctness of SVAs. This iterative process continues until a stable threshold is achieved. If the agents are unable to finalize the SVAs within a predefined maximum number of iterations, human intervention is triggered for further assessment. Once the SVAs are finalized, they undergo formal verification, with any CEXs identified and resolved by the agents.

IV. Benchmarking and Results

To evaluate the performance and benchmark its capabilities, we used Saarthi to verify RTL designs of varied complexity. Tables IV, V and VI underline the performance of Saarthi on basic, intermediate, and advanced design complexity levels. We chose three Key Performance Indicators (KPIs) to benchmark the results with different LLMs. The first is "success rate" which determines the number of successful runs (i.e., end-to-end formal verification) out of the total runs. The second KPI is the coverage rate (formal coverage after end-to-end formal verification). The third KPI indicates the pass rate of the assertions generated from Saarthi with each LLM. The results are highlighted in Fig. 12.

TABLE IV: Performance of Saarthi on basic difficulty level designs

Design	Metric	Pass@1			Pass@2			Pass@3		
		GPT-4o	GPT-4-Turbo	Llama3	GPT-4o	GPT-4-Turbo	Llama3	GPT-4o	GPT-4-Turbo	Llama3
Accumulator	# Properties	11	4	9	13	10	0	12	6	0
	% Proven	45.45%	50%	44.44%	53.85%	70%	0%	33.33%	16.67%	0%
	% Coverage	81.82%	80.95%	77.78%	80%	84%	0%	80.77%	83.58%	0%
8-bit ALU	# Properties	10	5	18	13	13	0	16	14	0
	% Proven	20%	20%	27.78%	7.69%	92.31%	0%	37.50%	14.29%	0%
	% Coverage	93.29%	94.33%	90.53%	95.54%	87.59%	0%	92.68%	93.13%	0%
Edge detector	# Properties	6	8	7	5	7	8	7	9	10
	% Proven	33.33%	62.50%	14.29%	40%	71.43%	75%	28.57%	44.44%	40%
	% Coverage	60%	60%	53.85%	50%	72.73%	75%	50%	73.33%	70.59%
4-state FSM	# Properties	6	5	9	9	9	7	9	8	8
	% Proven	100%	100%	88.89%	100%	77.78%	57.14%	100%	100%	75%
	% Coverage	82.22%	80.95%	65.23%	52.94%	83.33%	78.74%	75%	75.59%	71.25%
Up-down counter	# Properties	6	8	8	9	8	8	9	0	8
	% Proven	33%	75%	88%	33%	25%	25%	26%	0%	75%
	% Coverage	56%	69%	0%	53%	71%	79%	81%	0%	7%

TABLE V: Performance of Saarthi on intermediate difficulty level designs

Design	Metric	Pass@1			Pass@2			Pass@3		
		GPT-4o	GPT-4-Turbo	Llama3	GPT-4o	GPT-4-Turbo	Llama3	GPT-4o	GPT-4-Turbo	Llama3
Sync. FIFO	# Properties	14	12	12	15	13	7	6	9	13
	% Proven	7.14%	41.67%	8.33%	33.33%	23.08%	28.57%	50%	33.33%	30.77%
	% Coverage	90.48%	58.71%	91.66%	50%	73.08%	71.43%	50%	87.50%	84.62%
FSM controller	# Properties	29	4	0	30	6	0	31	10	0
	% Proven	48.28%	75%	0%	43.33%	33.33%	0%	52.38%	20%	0%
	% Coverage	72.41%	42.86%	0%	68.97%	84.62%	0%	70%	73.68%	0%
Priority encoder	# Properties	11	11	12	7	12	16	4	10	11
	% Proven	18%	55%	17%	71%	33%	38%	50%	40%	18%
	% Coverage	82%	82%	83%	25%	82%	87%	50%	70%	78%
16-bit LFSR	# Properties	6	5	6	6	5	6	5	5	8
	% Proven	83%	20%	67%	50%	40%	33%	40%	40%	40%
	% Coverage	0%	0%	44%	40%	43%	67%	40%	71%	78%
CRC generator	# Properties	8	6	9	4	7	5	8	6	10
	% Proven	25%	33%	100%	50%	43%	40%	38%	50%	60%
	% Coverage	80%	67%	0%	50%	57%	63%	67%	67%	75%

Saarthi performs the best with the GPT-4o model and the worst with the Llama3-70B model. GPT-4o has a consistent proof rate and the most consistent coverage metrics. Even though the proven rate of the assertions is lower for GPT-4o, it yields a higher overall coverage. GPT-4-Turbo has the highest proven rate variability (i.e., less consistent results with varied design complexity). It has a higher average assertion proven rate but lower overall coverage. Llama3-70B has consistently lower success rates with the highest number of attempts needed to complete the overall end-to-end formal verification. It is worth noting that this model often generates more assertions per run compared to the other two.

LLMs have context length and input token restrictions that can lead to truncated critical information in large RTL designs, resulting in inaccurate assertions. The quality of LLM-generated outputs depends on the prompts, yet models may still deviate from guidelines, producing incorrect assertions. Selecting the optimal temperature parameter for assertions varies across designs; incorrect settings cause overly deterministic or random outputs. Robust models like GPT-4o can err by introducing implicit clocks and resets in smaller circuits. LLMs often misinterpret intricate RTL constructs, generate syntactically incorrect or

TABLE VI: Performance of Saarthi on advanced difficulty level designs

Design	Metric	Pass@1			Pass@2			Pass@3		
		GPT-4o	GPT-4-Turbo	Llama3	GPT-4o	GPT-4-Turbo	Llama3	GPT-4o	GPT-4-Turbo	Llama3
Booth multiplier	# Properties	0	9	9	0	4	10	0	6	9
	% Proven	0%	44.44%	22.22%	0%	50%	50%	0%	33.33%	22.22%
	% Coverage	0%	77.78%	77.78%	0%	42.86%	90%	0%	66.67%	88.89%
Pipelined adder	# Properties	0	7	7	0	10	0	0	7	0
	% Proven	0%	42.86%	28.57%	0%	40%	0%	0%	85.71%	0%
	% Coverage	0%	57.14%	83.33%	0%	70%	0%	0%	13.33%	0%
RV32I core	# Properties	12	14	10	14	14	12	14	14	10
	% Proven	50%	44.43%	30.02%	55%	54.24%	35%	70%	75%	44.32%
	% Coverage	80%	82%	50%	82%	82%	51%	82%	82%	53%
PID Controller	# Properties	5	2	0	1	4	0	5	2	0
	% Proven	80.30%	50%	0%	100%	25%	0%	20%	50%	0%
	% Coverage	46.77%	13.60%	0%	9.40%	17.69%	0%	44.53%	42.97%	0%
Round robin arbiter	# Properties	0	7	7	0	6	0	0	4	0
	% Proven	0%	57%	29%	0%	33%	0%	0%	25%	0%
	% Coverage	0%	50%	67%	0%	89%	0%	0%	78%	0%

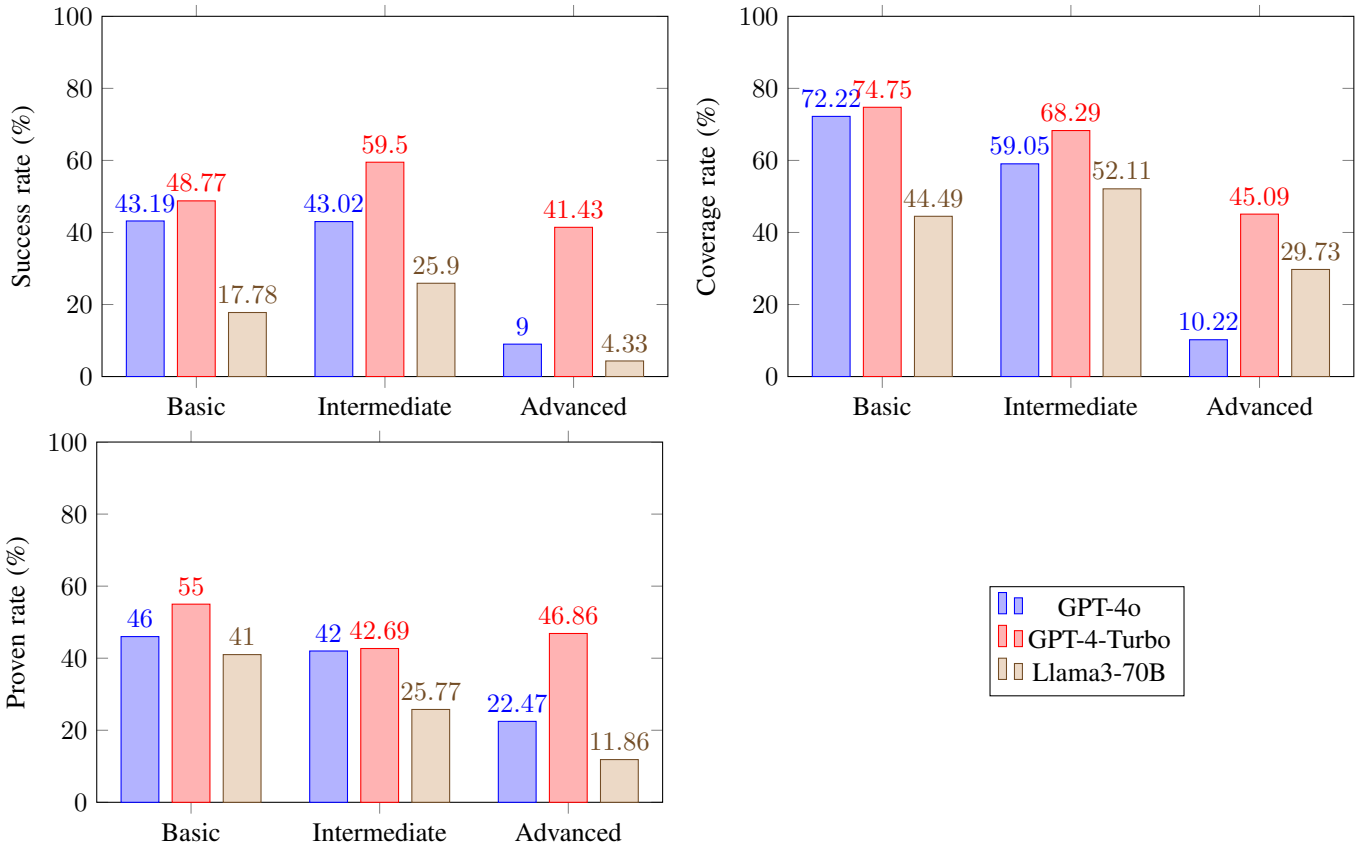

Fig. 12: KPIs for Saarthi benchmarking

semantically irrelevant assertions, and produce vacuous passes in formal properties. GPT-4o can create overly complex assertions with redundant conditions, while Llama models frequently have syntax errors. Additionally, GPT-4 models may introduce unnecessary assertions and signals not present in the original RTL specification.

V. Conclusion

The paper outlines a fully autonomous AI-based formal verification engineer, Saarthi. Saarthi understands design specifications, creates a verification plan, assigns tasks to several AI verification engineers, and communicates with formal verification tools such as Cadence Jasper to prove properties. It also analyzes CEXs, assesses formal coverage, and reacts to improve it by adding missing properties for the final sign-off. Although the results for end-to-end formal verification do not yield a $100\,\%$ guarantee with every run, Saarthi performs significantly well most of the time, with an overall efficacy of around $40\,\%$. The quality of the results also depends on the LLM used, with GPT-4o outperforming other models. As predicted in [24] and [25], achieving Artificial General Intelligence (AGI) by 2027 is strikingly plausible, and we believe that Saarthi could be pivotal in reaching that milestone within the hardware design verification domain. Saarthi is based on the agentic workflow, which makes it highly scalable and adaptable to other domains such as RTL design, UVM testbench generation, and more. To expand its capabilities, we need to define the agents and their responsibilities to ensure they work together to solve given tasks. In the future, we will continue our experiments to extend Saarthi's applications beyond formal verification, exploring its potential in additional usecases.

References

[1] Cognition AI. *Introducing Devin, the first AI software engineer*. https://www.cognition.ai/blog/introducing-devin. Accessed: 2024-07-17.

[2] OpenDevin Team. *OpenDevin: An Open Platform for AI Software Developers as Generalist Agents*. https://github.com/OpenDevin/OpenDevin. Version v1.0. 2024.

[3] Erik Seligman, Tom Schubert, and M V Achutha Kiran Kumar. *Formal Verification, An Essential Toolkit for Modern VLSI Design*. Morgan Kaufmann Publishers, 2015.

[4] Harry Foster. *2022 Wilson Research Group Functional Verification Study*. Tech. rep. Mentor, A Siemens Business, Oct. 2022.

[5] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[6] Chen Qian et al. *ChatDev: Communicative Agents for Software Development*. 2024. arXiv: 2307.07924 [cs.SE]. URL: https://arxiv.org/abs/2307.07924.

[7] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[8] Moez Krichen. "A Survey on Formal Verification and Validation Techniques for Internet of Things". In: *Applied Sciences* 13.14 (2023). ISSN: 2076-3417. DOI: 10.3390/app13148122. URL: https://www.mdpi.com/2076-3417/13/14/8122.

[9] Erik Berg. "What ChatGPT and Generative AI means for Semiconductor Design and Development". In: *60th Design Automation Conference*. 2023.

[10] Deepak Narayan Gadde et al. *All Artificial, Less Intelligence: GenAI through the Lens of Formal Verification*. 2024. arXiv: 2403.16750 [cs.AI]. URL: https://arxiv.org/abs/2403.16750.

[11] Andrew Ng. "What's next for AI agentic workflows". In: *Sequoia Capital's AI Ascent*. 2024.

[12] Aman Madaan et al. *Self-Refine: Iterative Refinement with Self-Feedback*. 2023. arXiv: 2303.17651 [cs.CL]. URL: https://arxiv.org/abs/2303.17651.

[13] Noah Shinn et al. *Reflexion: Language Agents with Verbal Reinforcement Learning*. 2023. arXiv: 2303.11366 [cs.AI]. URL: https://arxiv.org/abs/2303.11366.

[14] Shishir G. Patil et al. *Gorilla: Large Language Model Connected with Massive APIs*. 2023. arXiv: 2305.15334 [cs.CL]. URL: https://arxiv.org/abs/2305.15334.

[15] Zhengyuan Yang et al. *MM-REACT: Prompting ChatGPT for Multimodal Reasoning and Action*. 2023. arXiv: 2303.11381 [cs.CV]. URL: https://arxiv.org/abs/2303.11381.

[16] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[17] Yongliang Shen et al. *HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face*. 2023. arXiv: 2303.17580 [cs.CL]. URL: https://arxiv.org/abs/2303.17580.

[18] Qingyun Wu et al. *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation*. 2023. arXiv: 2308.08155 [cs.AI]. URL: https://arxiv.org/abs/2308.08155.

[19] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[20] Shishir G. Patil et al. *Gorilla: Large Language Model Connected with Massive APIs*. 2023. arXiv: 2305.15334 [cs.CL]. URL: https://arxiv.org/abs/2305.15334.

[21] Aman Kumar and Deepak Narayan Gadde. *Generative AI Augmented Induction-based Formal Verification*. 2024. arXiv: 2407.18965 [cs.AI]. URL: https://arxiv.org/abs/2407.18965.

[22] Yilun Du et al. *Improving Factuality and Reasoning in Language Models through Multiagent Debate*. 2023. arXiv: 2305.14325 [cs.CL]. URL: https://arxiv.org/abs/2305.14325.
[23] Junyou Li et al. *More Agents Is All You Need*. 2024. arXiv: 2402.05120 [cs.CL]. URL: https://arxiv.org/abs/2402.05120.
[24] Leopold Aschenbrenner. *Situational Awareness: The Decade Ahead*. https://situational-awareness.ai/wp-content/uploads/2024/06/situationalawareness.pdf. June 2024.
[25] Sébastien Bubeck et al. *Sparks of Artificial General Intelligence: Early experiments with GPT-4*. 2023. arXiv: 2303.12712 [cs.CL]. URL: https://arxiv.org/abs/2303.12712.