LLM4SDC: Leveraging Multi-Agent System for Automated SDC Generation and Benchmarking

Peiyi Han
CUHK
Hong Kong, China

Yuntao Lu
CUHK
Hong Kong, China

Haiyang Liu
Tongji Univ.
Shanghai, China

Fangzhou Liu
CUHK
Hong Kong, China

Xufeng Yao
CUHK
Hong Kong, China

Yuyang Ye*
CUHK
Hong Kong, China

Abstract

Logic synthesis quality depends critically on accurate Synopsys design constraints (SDCs), yet manual generation is labor-intensive and error-prone. We construct SDCBench, the first SDC generation benchmark with aligned specification-design-testbench triples, and LLM4SDC, the first end-to-end automated framework for SDC generation. LLM4SDC hierarchically decomposes the task into specialized agents to extract timing-relevant information, conduct domain-specific retrieval-augmented generation, and synthesize static constraints from expert-curated templates, as well as discover dynamic timing exceptions by verification-guided pipelines. Evaluated on SDCBench, LLM4SDC achieves near-perfect syntax correctness, superior semantic fidelity, and high false path accuracy, significantly outperforming state-of-the-art baselines while improving physical design metrics.

1 Introduction

Logic synthesis remains a cornerstone of digital design flows, transforming register transfer level (RTL) descriptions into optimized gate-level netlists under strict timing, area, and power (PPA) constraints [1–3]. The success of the transformation depends critically on the accuracy of Synopsys design constraints (SDCs) files used by synthesis tools, such as Synopsys Design Compiler (DC) [4]. SDCs define clocks, input/output delays, false paths, multi-cycle paths, and other timing exceptions. Despite their centrality, SDC generation remains predominantly manual, requiring expert designers to transfer natural language specifications (Specs) and RTL code into hundreds of precisely formulated tool command language (Tcl) commands [5–8]. The process requires a deep understanding of both architectural intent and the idiosyncrasies of synthesis tools, which is time-consuming, error-prone, and a bottleneck in modern chip development.

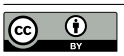
The automation of SDC generation has remained largely unexplored due to fundamental barriers. Despite existing large language models (LLMs) being proficient in code generation, they lack the precision and domain grounding necessary to produce tool-valid constraint scripts consistently. Synthesis tools enforce strict syntactic and semantic rules for SDC Tcl commands, where even minor deviations can lead to synthesis failures or incorrect optimizations [9–11].

There are no publicly available benchmarks for SDC generation tasks, primarily because aligned datasets containing high-level Specs, RTL code, and corresponding testbenches (TBs) are exceedingly rare, hindering both method development and rigorous evaluation. Existing open-source benchmarks in hardware design [12–16] primarily focus on generating modular RTL code, while they exhibit limitations that render them unsuitable for SDCs. Previous benchmarks mainly contain small-scale and isolated modules rather than complete hierarchical designs with realistic clocking architectures and inter-module dependencies [17]. Lacking essential explicit clock and reset generation (CRG) logic degrades the ability to derive accurate timing constraints and reset synchronization paths. A significant limitation is that existing benchmarks do not provide aligned high-level Specs or comprehensive TBs that capture the intended behavior and operational scenarios of the design. Although benchmarks for hardware verifications [18–20] partially focus on TB generation from high-level Specs, they lack a coherent, end-to-end data collection and alignment pipeline that aligns descriptive Specs, raw RTLs, and verification TBs under a unified semantic meaning.

Furthermore, constraint identification beyond the capacities of existing LLMs is a significant challenge. Timing constraints are implicitly encoded across thousands of lines of RTL code and abstract Specs, making direct extraction infeasible for conventional natural language processing (NLP) or rule-based methods [21, 22]. Especially, dynamic constraints require holistic reasoning of control flow, clock domain interactions, and functional behaviors from dynamic verification and simulation feedback [23, 24], which existing LLMs struggle to achieve. For example, false path constraints specify timing exceptions for paths that are structurally present but never sensitized during regular operation, requiring dynamic verification to confirm functional unreachability through simulation analysis of control logic and mode interactions [25–27].

To address the benchmark gap, we propose a multi-agent workflow that automatically synthesizes aligned Spec-RTL-TB triples from open-source repositories to construct the first benchmark **SDCBench**. To improve the capabilities of LLMs for SDC generation, we present **LLM4SDC**, the first end-to-end multi-agent framework that hierarchically decomposes the SDC generation task into specialized agents to handle strict-syntax and long-context requirements, ensure semantic alignment, and identify implicit static and dynamic constraints.

*Corresponding author.
This work is supported by The Research Grants Council of Hong Kong SAR (No. CUHK14211824 and No. CUHK14210723).



DAC '26, July 26–29, 2026, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07…$15.00
https://doi.org/10.1145/3770743.3804364

In LLM4SDC, the parser agent extracts timing-relevant information from Specs and RTL, filtering noise and irrelevant logic. A module of domain-specific retrieval-augmented generation (RAG) with expert-curated SDC templates and tool documentation ensures syntactic compliance and semantic fidelity. Finally, a verification-guided false path discovery pipeline combines simulation-based coverage analysis with LLM-powered semantic reasoning to reliably identify dynamic timing exceptions.

We evaluate LLM4SDC using three novel metrics tailored to SDC generation: syntax correctness, semantic fidelity, and false path accuracy. Syntax correctness measures the proportion of generated constraints accepted by the synthesis tool without error. Semantic fidelity quantifies the coverage and accuracy of timing intent captured from specifications. False path accuracy evaluates the precision of identifying *workload-dependent* timing exceptions. LLM4SDC achieves near-perfect syntax correctness with approximately 1.00, superior semantic fidelity of 0.93, and high false path accuracy of 0.67, significantly outperforming state-of-the-art (SOTA) LLM baselines including GPT 4.1 [28, 29] and Claude 4.5 [30]. Furthermore, the generated constraints yield measurable physical benefits on industrial-scale designs, significantly improving slack by 29.4%, reducing power by 1.15%, and increasing area by only 0.27% on average.

Contributions are summarized as follows:

- To the best of our knowledge, we construct the first benchmark for SDC generation, comprising aligned Spec-RTL-TB triples automatically synthesized from open-source repositories with verified ground-truth constraints.
- The proposed LLM4SDC framework is the first end-to-end multi-agent framework for automated SDC generation from Spec and RTL, establishing a new paradigm for AI-driven constraint generation in electronic design automation (EDA).
- The hierarchical multi-agent system enables high-fidelity constraint generation by integrating domain-enhanced RAG for static constraint synthesis and verification-guided discovery for dynamic timing exceptions.
- With three novel evaluation metrics, experimental results demonstrate that LLM4SDC achieves substantial improvements over SOTA baselines across all dimensions.

2 Background

Synopsys Design Constraint SDC is the de facto standard expression for design intent, covering key constraints on PPA. It is the common constraint language across EDA tools, such as the synthesis tool DC, IC Compiler, and PrimeTime. It propagates intent consistently along the flow, preserving constraint alignment from logic synthesis to physical implementation [9, 10].

SDCs fall into two categories: static constraints, derivable from structural and timing information in the Specs and RTLs, and dynamic constraints, which capture timing or logic exceptions tied to operational semantics and workload behavior, and are not directly exposed in the design sources. In this work, we adopt a representative subset of industry-recommended commands and, for dynamic constraints, focus on `set_false_path`. A false path is a timing path that is never sensitized under any legal operating condition and should be excluded to avoid spurious violations and misleading optimizations. Identifying false paths requires reasoning about functional inactivity and mode exclusivity beyond RTL or specifications, making the automated generation of test cases particularly challenging. We list the selected Tcl commands in Table 1.

Table 1: Used Tcl commands to construct SDCs, highlighting dynamic constraints in gray.

Type of Information	Commands
System interface	`set_input_transition` `set_load`

Design rule constraints	`set_max_capacitance`
	`set_max_fanout`
	`set_max_transition`
	`set_min_capacitance`

Timing constraints	`create_clock`
	`create_generated_clock`
	`set_clock_groups`
	`set_clock_latency`
	`set_clock_transition`
	`set_clock_uncertainty`
	`set_input_delay`
	`set_output_delay`

Timing exceptions	`set_multicycle_path` `set_false_path`

Agent System for EDA Agent-based methodologies in EDA strive to manage the high complexity of hardware design workflows [31–33]. Recent research highlights the use of multi-agent frameworks powered by LLMs to automate critical tasks, for example, streamlining the complete RTL-to-GDSII design flow via automated task decomposition and script execution [31], and automatically producing syntactically and functionally correct RTL code from natural language Specs [32], and generating high-quality, domain-specific EDA scripts [33].

3 Construction of SDCBench

To bridge this gap, we construct **SDCBench**, the first benchmark for automated SDC generation, by developing a multi-agent workflow that automatically synthesizes aligned Spec-RTL-TB triples from open-source repositories. As illustrated in Figure 1, our pipeline addresses three fundamental challenges. Since raw open-source RTL designs typically lack explicit CRG modules, making clock domain identification infeasible, we augment each design with an LLM-synthesized CRG module to explicitly expose timing domains. To understand the long-context, noise-intensive, and timing-sparse raw RTL files that overwhelm LLMs with syntactic details, we extract compact, timing-aware representations through specialized agents. To ensure semantic consistency across heterogeneous natural-language Specs, executable TBs, and RTL code, we enforce alignment by conditioning both the Spec and TB generators on the same structural design information. To finalize the quality of Specs and TBs, we perform a thorough review process by human experts.

3.1 Heterogeneous Information Extractor

Two cooperative agents extract design and library information from RTL source files and standard cell libraries, using inputs critical to downstream agents for timing semantics. Timing constraints are implicitly encoded across thousands of lines of RTL code, making direct LLM processing infeasible. The RTL extractor addresses this by parsing the augmented design with the synthesized CRG module and isolating timing-critical structures, including top-level module interfaces, inter-module connectivity, clock and reset domains, from CRG instantiation, pipeline stages, and candidate multi-cycle paths.

The extraction schema yields a structural design information representation $\mathcal{I}_{\text{RTL}}$ that captures architectural timing intent without overwhelming downstream agents with irrelevant syntactic details, enabling consistent downstream generation.

To avoid flooding the LLM with vast amounts of low-level data from standard cell libraries that are irrelevant to high-level constraint reasoning, the library extractor filters and summarizes constraint-relevant parameters from given technology libraries, including clock transition limits, cell drive strengths, setup/hold characteristics of sequential elements, and pin-level timing constraints. The extractor produces a compact library information representation, $\mathcal{I}_{\text{Lib}}$, that grounds timing specifications in physical technology parameters, ensuring that the generated constraints respect tool-specific timing requirements.

3.2 Aligned TB and Spec Generation

With distilled design and library information, we instantiate a dual-agent system that generates semantically aligned Spec-TB pairs. Both the TB and Spec generators use the same structural abstraction, $\mathcal{I}_{\text{RTL}}$, ensuring consistency in clock domains, design interfaces, and timing assumptions across all Spec, RTL, and TB artifacts. Shared dependency enables supervised alignment for rigorous evaluation of SDC generation.

TB Generator. The TB generator constructs a simulation environment by instantiating the top-level design, wiring clocks and resets according to the synthesized CRG module. Leveraging $\mathcal{I}_{\text{RTL}}$ with module hierarchy, inter-module connectivity, and pipeline structures, it generates stimulus patterns and assertions that exercise timing-critical scenarios, such as mode transitions, pipeline fills and drains, and handshake sequences. The resulting TB provides concrete, executable behaviors that enable the verification-guided false path discovery described in Section 4.3, grounding abstract timing intents through simulation-based coverage analysis.

Spec Generator. In parallel, the Spec generator produces a natural-language description of design functionality, operating modes, and timing assumptions. Unlike the executable TB, the Spec synthesizes high-level timing intent by consuming both $\mathcal{I}_{\text{RTL}}$ and $\mathcal{I}_{\text{Lib}}$. It describes interface protocols, submodule roles, pipeline depths, and multi-cycle behaviors while embedding technology-specific constraints derived from library parameters, such as "clock X transition must not exceed Y ps at register bank Z inputs". This dual conditioning ensures the Spec reflects both architectural intent and physical timing requirements, providing supervision for static constraint generation in Section 4.2.

Cross-Artifact Alignment. To enforce semantic consistency across heterogeneous artifacts of code-based TBs and natural-language Specs, we architect both generators to condition on the same design information $\mathcal{I}_{\text{RTL}}$ while incorporating their respective private inputs. Formally, we express the generation process as Equation (1).

$$\text{TB} = \mathcal{G}_{\text{TB}}(\mathcal{I}_{\text{RTL}},\ \text{RTL}), \quad \text{Spec} = \mathcal{G}_{\text{Spec}}(\mathcal{I}_{\text{RTL}},\ \mathcal{I}_{\text{Lib}}), \tag{1}$$

where $\mathcal{G}_{\text{TB}}$ and $\mathcal{G}_{\text{Spec}}$ denote the TB and Spec generators, respectively, and RTL represents the synthesized raw RTL code. The shared dependency on $\mathcal{I}_{\text{RTL}}$ ensures that timing intents expressed in the Spec are grounded in executable behavior in the TB. For example, when the Spec describes a multi-cycle path assumption, the TB instantiates corresponding stimulus sequences that exercise that path in simulation to discover unreachable paths.

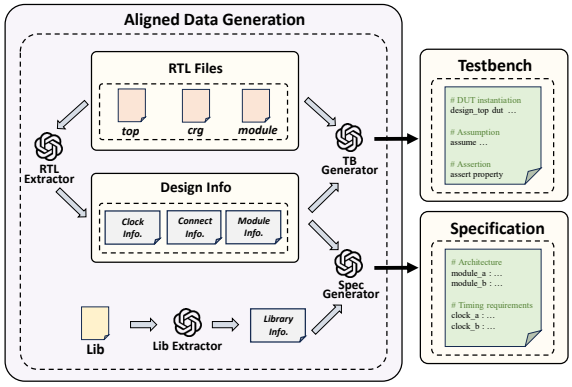


Figure 1: Multi-agent workflow for aligned data generation.

4 Design of LLM4SDC Framework

Automating SDC generation faces barriers due to strict syntactic and semantic rules, as well as implicit static and dynamic reasoning constraints that existing LLMs struggle to overcome. To overcome these challenges, we present **LLM4SDC**, which hierarchically decomposes the task into specialized agents that address specific aspects of constraint generation.

4.1 Multi-Agent Architecture

SDC generation exhibits a fundamental dichotomy between static and dynamic constraints. Static constraints encode structural, mode-agnostic timing intent, such as clock definitions, I/O delays, clock groups, and multi-cycle paths.In contrast, dynamic constraints capture behavior-dependent timing exceptions, such as false paths, that are structurally present but never sensitized under any regular operations. These constraints require verification-guided analysis to confirm functional unreachability through simulation of control logic and mode interactions.

To address the dichotomy, LLM4SDC decomposes SDC generation into two cooperative yet specialized pipelines for static constraint generation and dynamic constraint generation, as shown in Figure 2. Each pipeline employs multiple specialized agents to ensure global consistency. By hierarchically decomposing the task, the framework systematically addresses strict-syntax requirements through domain-specific retrieval, handles long-context challenges by information extraction, and identifies implicit constraints through verification-guided discovery. Critically, both pipelines leverage the aligned Spec-RTL-TB triples from SDCBench.

4.2 Static Constraint Generation

Static constraints capture structural timing intent independent of specific workloads, including primary and generated clocks, clock groups, input/output delays, and multi-cycle paths. Existing LLMs lack the precision and domain grounding necessary to consistently produce tool-valid constraint scripts, due to strict syntactic rules and limited exposure to SDC-specific syntax. Our static generation pipeline addresses these challenges through specialized extraction agents and domain-enhanced RAG.

RTL and Spec Extractors. Implicit timing constraints span thousands of lines of RTL code and abstract Specs. The static pipeline

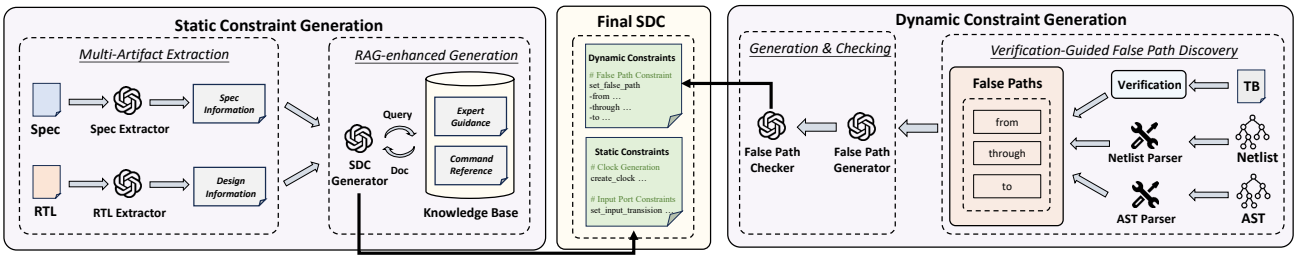


Figure 2: Overview of LLM4SDC multi-agent architecture.

begins with two complementary extraction agents that mirror the design described in Section 3 but focus on constraint-relevant design information.

SDC Generator with Domain-Specific RAG. Even with extracted design information, standard LLMs fail to consistently produce tool-acceptable SDC scripts due to limited exposure to SDC-specific syntax and the strict syntactic rigidity of EDA tools. We develop a domain-specific RAG module that grounds the SDC generator agent in authoritative SDC documentation, utilizing expert-curated templates and tool documentation.

The retrieval corpus from SDC user manuals and reference guides, including full syntax templates, parameter definitions, and usage examples. However, naive semantic-similarity retrieval is insufficient for Tcl commands because of their narrow, overlapping vocabularies. Commands with distinct semantics often have similar textual descriptions and are rarely applicable under specific design conditions that are not obvious from their descriptions. A purely embedding-based retriever over-retrieves superficially similar commands, leading to incorrect or overly aggressive SDC suggestions.

To address this limitation, we adopt an expert-prioritized hybrid retrieval strategy. For a candidate SDC command document C and a query Q constructed from $\mathcal{I}_{\text{static}}$, we define the retrieval score as a combination of expert priors and document similarity signals:

$$S(C, Q) = \alpha \cdot \text{sim}(Q, C) + (1 - \alpha) \cdot \pi(C), \tag{2}$$

where $\text{sim}(Q, C) \in [0, 1]$ is the cosine similarity between query and document embeddings, and $\pi(C) \in [0, 1]$ is an expert-assigned priority weight reflecting how likely command C should be used in typical design scenarios. The mixing coefficient $\alpha \in [0, 1]$ balances data-guided relevance and expert guidance, and both terms are normalized to the range $[0, 1]$, eliminating the need for additional scaling. We set $\alpha = 0.6$ based on validation-set tuning, prioritizing semantic alignment while preserving strong expert priors.

The score formulation ensures essential commands remain retrievable when $\pi(C)$ is high, even if the semantic match is modest, while preventing overuse of niche or unsafe commands with low priors. Furthermore, we augment each document with expert-authored usage guidelines capturing engineering conventions beyond raw syntax—recommended constraint grouping, naming conventions, and tool-specific requirements.

Conditioned on top-k retrieved documents and structured outputs from extractors, the SDC generator agent synthesizes clock, delay, and related commands. The domain-specific RAG ensures that generated constraints not only parse correctly under strict EDA tool requirements but also align with industry best practices, directly addressing the syntactic compliance challenge.

4.3 Dynamic Constraint Generation

Dynamic constraints describe timing exceptions whose validity depends on design functional behavior and operational modes. Identifying dynamic constraints requires holistic reasoning across control flow, clock-domain interactions, and functional behaviors, drawing on dynamic verification and simulation feedback—capabilities that existing LLMs lack. The most prominent example is the false path constraint, a timing path structurally present but never sensitized under any legal input sequence, thus requiring no timing closure. Automatically discovering false paths is challenging because it demands reasoning about both structure and behavior. LLM4SDC addresses it through a verification-guided false path discovery pipeline that synergizes simulation-based coverage analysis with semantic reasoning.

Verification-Guided Path Discovery. The dynamic pipeline leverages the aligned TBs from SDCBench to enable simulation-based unreachability analysis. We use commercial verification tools to simulate the RTL design using the generated TB and generate coverage reports. Statements and branches that remain unexecuted across all test scenarios indicate operations that were never functionally activated, forming candidates for false-path analysis.

The verification report parser ingests coverage reports and extracts unreachable assignment statements and control constructs, normalizing them into a list of RTL operations annotated with source file, line number, and surrounding control context, enclosing always blocks and conditional guards. The list of RTL operations serves as the entry point for reconstructing candidate timing paths, grounding the structural analysis in concrete simulation evidence.

Structural Path Reconstruction via AST and Netlist Parsing. Given an unreachable assignment statement identified by verification, we reconstruct its corresponding timing path through dual-level structural analysis. The AST parser operates on the abstract syntax tree to perform register-to-register path tracing. It decomposes each statement into a left-hand side (LHS), a right-hand side (RHS), and a conditional expression, then executes the following backward and forward traversals. From the RHS, follows signal definitions and module boundaries backward to locate the driving register output (`Reg/Q`) or primary input port, yielding the path source `from`. From the LHS, follow assignments and module hierarchies forward to identify the destination register input (`Reg/D`) or primary output port, yielding the path sink `to`. The AST-level tracing respects RTL structural semantics, correctly handling nested conditionals, hierarchical

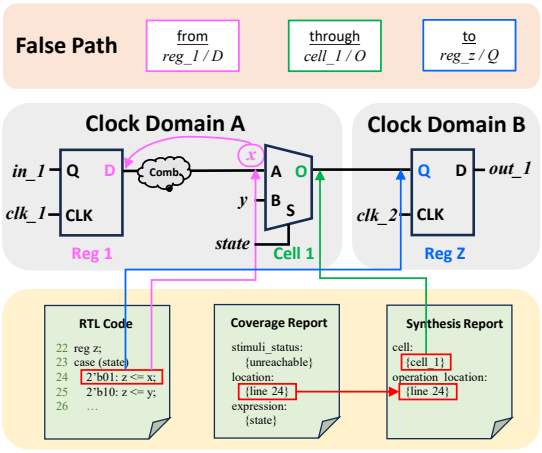


Figure 3: Pipeline of verification-guided false path discovery.

references, and generate constructs, producing candidate register-to-register path pairs.

While AST tracing reveals timing endpoints, it does not capture how control logic turns off paths at the gate level. The netlist parser addresses this by leveraging the synthesized design and previously generated static SDC to obtain a gate-level netlist and synthesis report that maps RTL signals to cell instances and pins. For each unreachable assignment, the parser examines conditional expressions (enable or mode signals) and maps these RTL signals to physical cell pins via the synthesis report. Pins form the `through` component, while AST-derived endpoints of `from` and `to` components. Special handling treats constant-driven paths, such as `reset` signal, by starting directly from the conditional control signal.

Figure 3 illustrates the process for verification-guided discovery. RTL contains an assignment gated by an active-low reset signal, making it unreachable during functional operation. Simulation coverage flags the statement, triggering AST traversal to identify sink `reg_z.Q` and source `reg_1.D`. Conditional expression `state` is mapped to the corresponding cell pin in the synthesized netlist. Components compose into a semantically correct SDC constraint, demonstrating guidelines of simulation evidence for structural reasoning to produce valid dynamic constraints.

False Path SDC Generator and Semantic Validation. The structural analysis produces a set $\mathcal{F}$ of high-recall candidates in the form of (`from`, `through`, `to`) triples. The false path SDC generator converts structural descriptors into concrete SDC Tcl commands by canonicalizing endpoints and through-points according to the synthesized netlist, resolving hierarchical instance names, and mapping signals to pins. Overlapping or identical triples are grouped and merged into minimal, non-redundant exception sets. For each surviving triple, the generator instantiates a template for `set_false_path` with appropriate clauses from `-from`, `-to`, and `-through`, yielding draft false path SDC commands structurally consistent with the design.

Although the verification-guided method provides high recall for structurally unreachable paths, it remains incomplete and may include spurious candidates. Deep or rare false paths may not be triggered within a finite time, and purely structural analysis cannot capture semantic infeasibility arising from protocol-level invariants. Conversely, while LLM-based agents can hypothesize plausible false paths from high-level Spec descriptions using learned design patterns, they lack formal guarantees and may propose unsafe constraints.

To synergize these complementary strengths, we introduce a false-path-checker agent that combines simulation evidence with LLM-powered semantic reasoning. We prompt agents with (i) expert-curated guidelines on common false path patterns, (ii) the candidate path set $\mathcal{F}$, and (iii) intermediate analysis results from parsers of control-flow annotations and coverage statistics. It reasons over this input to refine the candidate list, filtering out spurious paths and proposing additional high-likelihood paths that were not directly observed in simulation.

Crucially, for each accepted false path, the agent generates a natural-language justification, enabling designers to quickly validate proposed exceptions. The finalized output is a vetted set of `set_false_path` commands that are both structurally consistent and semantically well-motivated.

5 Experiments

5.1 Experimental Setup

Benchmark. The evaluation benchmark, SDCBench, is based on the IWLS benchmark [34] of RTL designs. Following the aligned data generation pipeline described in Section 3, we automatically synthesize for each design a corresponding high-level Spec and a functional testbench. The resulting benchmark consists of 22 designs spanning diverse application domains. All designs are synthesized using the ASAP7 PDK [35], and Synopsys DC [4] is the reference synthesis engine. For RTL parsing and AST construction, we leverage the PyVerilog library [36].

Model. LLM4SDC instantiates each agent using GPT 4.1 as the backbone and accesses the model via the official agent APIs. All agents share the same model and utilize specialized system prompts and retrieval contexts.

Baselines. LLM4SDC compares with SOTA commercial GPT-4.1 and Claude Sonnet 4.5 models, both of which achieve strong performance on programming tasks. Prompting models with truncated Specs and RTLs to the context window, we instruct them to generate a complete SDC file. Baseline models do not use retrieval augmentation or multi-agent coordination, representing a strong single-shot LLM approach.

Metrics. We define syntax correctness, semantic fidelity, and fals path accuracy metrics to evaluate the quality of generated SDC constraints, each targeting a distinct aspect of correctness. Syntax correctness ($\mathcal{A}_{\text{syn}}$) is the fraction of all generated commands accepted by DC without syntax errors, which applies to both static and dynamic constraints. We compute the ratio of the number of syntax-valid commands accepted by DC over the total number of generated commands, and formulate the score as

$$\mathcal{A}_{\text{syn}} = \frac{|\{c \in C_{\text{gen}} \mid \text{SyntaxCheck}(c) = \text{true}\}|}{|C_{\text{gen}}|}, \tag{3}$$

Semantic fidelity ($\mathcal{A}_{\text{sem}}$) is a measure of whether static constraints accurately reflect timing intent explicitly stated in the Spec and RTL. Since SDCs are not unique, we consider only those that clearly contradict the Spec to be incorrect. The ratio of the number of semantically consistent static constraints to the total number of static constraints is

$$\mathcal{A}_{\text{sem}} = \frac{|\{c \in C_{\text{static}} \mid \text{ExpertVerify}(c) = \text{correct}\}|}{|C_{\text{static}}|}, \tag{4}$$

Table 2: End-to-end SDC generation performance.

Design	Syntax Correctness (↑)			Semantic Fidelity (↑)			False Path Accuracy (↑)		
	Ours	GPT	Claude	Ours	GPT	Claude	Ours	GPT	Claude
AC97	**1.00**	0.95	0.90	**0.96**	0.88	0.75	**0.56**	0.36	0.28
AES	**1.00**	0.93	0.88	**0.94**	0.86	0.72	**1.00**	0.34	0.25
DES	**1.00**	0.90	0.86	**0.92**	0.84	0.70	**0.38**	0.29	0.22
FPU	**1.00**	0.96	0.91	**0.95**	0.87	0.74	**0.80**	0.33	0.26
I2C	**1.00**	0.89	0.85	**0.93**	0.83	0.69	**0.59**	0.27	0.21
SPI	**1.00**	0.91	0.87	**0.91**	0.85	0.71	**1.00**	0.30	0.24
TV80	**1.00**	0.94	0.89	**0.94**	0.86	0.73	**0.43**	0.24	0.19
USB	**1.00**	0.88	0.84	**0.92**	0.82	0.68	**0.62**	0.32	0.27
VGA	**1.00**	0.97	0.92	**0.90**	0.89	0.76	**0.48**	0.26	0.23
WB	**1.00**	0.90	0.83	**0.93**	0.84	0.67	**0.81**	0.29	0.25
Average	**1.00**	0.92	0.88	**0.93**	0.85	0.72	**0.67**	0.30	0.24

Table 3: PPA comparison *w* and *w/o* dynamic constraints (false paths).

Design	Area (μm^2) (↓)		Power (uW) (↓)		Slack (ns) (↑)	
	w/o FP	w/ FP	w/o FP	w/ FP	w/o FP	w/ FP
AC97	**1017.5**	1018.6	**60.4**	60.7	8.76	**15.55**
FPU	2017.3	**2017.2**	183.8	**183.0**	1.79	**1.98**
I2C	**83.5**	84.1	25.6	**24.7**	**32.1**	**37.1**
Avg. ↓	—	**0.27%**	—	**-1.15%**	—	**29.4%**

Table 4: Ablation study showing the incremental impact of each component in LLM4SDC.

Configuration	Syntax ↑	Semantic ↑	False Path ↑
GPT-4.1 (Baseline)	0.92	0.85	0.30
+ Multi-Agent	0.95	0.90	0.42
+ RAG Module	1.00	0.89	0.43
+ Verification (Ours)	**1.00**	**0.93**	**0.67**

False path accuracy ($\mathcal{A}_{\text{fp}}$ is the fraction of proposed false path constraints confirmed correct by human experts through manual review. The score measures the practical requirement that false path constraints are safe and never turn off a sensitizable path. The ratio of the number of valid false path directives to the total number of generated false path constraints is

$$\mathcal{A}_{\text{fp}} = \frac{|\{f \in F_{\text{gen}} \mid \text{ExpertValidate}(f) = \text{correct}\}|}{|F_{\text{gen}}|}. \tag{5}$$

5.2 Experimental Results

We evaluate LLM4SDC on the benchmark of 22 representative designs to compare its performance with our method and two SOTA baselines. Due to space limitations, we present results of ten representative designs. As shown in Table 2, LLM4SDC achieves near-perfect syntax correctness across all designs, scoring an average value of **1.00**, significantly outperforming 0.93 of GPT 4.1 and 0.67 of Claude 4.5, which demonstrates the effectiveness of our domain-specific RAG module for tool-compliant SDC generation. In terms of semantic fidelity of static constraints, LLM4SDC achieves an approximate score of **0.93**, surpassing baselines by over 9.4% and 29.2% respectively, indicating superior ability to extract and represent constraint intent from Specs and RTLs.

For most challenging dynamic constraints, LLM4SDC achieves an average false path accuracy of **0.67**, compared to 0.30 and 0.24 of GPT 4.1 and Claude 4.5. The accuracy of baselines collapses on false path identification, whereas our system grounds each candidate in verification evidence and yields reliable results. Overall, LLM4SDC consistently outperforms SOTA LLMs across metrics and designs, demonstrating robust generalization and reliability. The extremely few syntax errors further suggest that the proposed framework can serve as a trustworthy assistant in real-world synthesis flows, reducing the burden of manual constraint authoring.

5.3 PPA Analysis

To demonstrate the impact of dynamic constraint generation, we evaluate the PPA of synthesized designs in two scenarios, which are (1) static constraints only, and (2) both static and dynamic constraints (with generated false paths). We select three representative designs from SDCBench that contain complex control logic and multi-mode operation.

Table 3 demonstrates that the automatically generated false path constraints lead to consistent improvements across PPA metrics. Here, the "slack" is positive. Since the designs in SDCBench are multi-clock-domain, we report results for the clock domain most significantly impacted by the generated constraints. The introduction of dynamic constraints significantly improves timing by 29.4%, reduces power by 1.15%, and increases area by only 0.27%. The PPA improvement results from false path constraints preventing the synthesis tool from over-optimizing infeasible timing paths, allowing more aggressive optimization on absolute critical paths.

5.4 Ablation Study

To understand the individual contributions of each component in LLM4SDC, we conduct an ablation study by incrementally adding key modules to a baseline large language model. We start with GPT-4.1, prompted to generate complete SDC files directly from Spec and RTL. We then progressively introduce our innovations: (i) the multi-agent decomposition framework for structured information extraction, (ii) the domain-specific RAG module to enhance syntactic correctness, and (iii) the verification-guided false path discovery mechanism for dynamic constraint generation.

Table 4 summarizes the performance across all three metrics: syntax correctness, semantic fidelity, and false path accuracy averaged over the 22 benchmark designs. Starting from the baseline, the introduction of the multi-agent architecture significantly improves syntax correctness from 0.92 to 0.95 and semantic fidelity from 0.85 to 0.90, which stems from isolating monolithic parsing tasks and reducing context noise. The RAG module further boosts syntax correctness to 1.00, demonstrating the grounding fact of precise domain knowledge. Finally, integrating the verification-guided false path discovery pipeline elevates false path accuracy to **0.92**. This underscores the indispensability of verification for false path discovery.

6 Conclusion and Future Work

To address the labor-intensive and data-starved SDC generation task, we present LLM4SDC, the first end-to-end multi-agent system for automated SDC generation from Specs and RTLs, achieving near-perfect syntax correctness, high semantic fidelity, and reliable false path accuracy, significantly outperforming state-of-the-art LLMs. The current framework covers 17 essential SDC commands, including core static and false path constraints. In the future, we plan to expand our directives to include dynamic commands and collaborate with industry experts to enrich our RAG knowledge continually.

References

[1] R. L. Rudell, *Logic Synthesis for VLSI Design*. University of California, Berkeley, 1989.

[2] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Springer, 1996.

[3] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, 2002.

[4] I. Synopsys, "Design Compiler: Concurrent Timing, Area, Power, and Test Optimization," https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler.html, 2025.

[5] J. K. Ousterhout *et al.*, *Tcl: An Embeddable Command Language*. University of California, Berkeley, 1989.

[6] H. Bhatnagar, *Advanced ASIC Chip Synthesis using Synopsys® Design Compiler™ Physical Compiler™ and Primetime®*. Springer, 2002.

[7] J. Cong and Z. Zhang, "An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation," in *ACM/IEEE Design Automation Conference (DAC)*, 2006, pp. 433–438.

[8] Z. Zhang and B. Liu, "SDC-Based Modulo Scheduling for Pipeline Synthesis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2013, pp. 211–218.

[9] Synopsys, Inc., *Design Compiler User Guide*, Synopsys, Inc., March 2019, version P-2019.03.

[10] ——, *Using the Synopsys Design Constraints Format Application Note*, Synopsys, Inc., June 2007, version Z-2007.03.

[11] ——, *Synopsys Timing Constraints and Optimization User Guide*, Synopsys, Inc., September 2019, version P-2019.03-SP4.

[12] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An Open-Source Benchmark for Design RTL Generation with Large Language Model," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*. IEEE, 2024, pp. 722–727.

[13] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating Large Language Models for Verilog Code Generation," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[14] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking Large Language Models for Automated Verilog RTL Code Generation," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*. IEEE, 2023, pp. 1–6.

[15] S. Liu, Y. Lu, W. Fang, M. Li, and Z. Xie, "OpenLLM-RTL: Open Dataset and Benchmark for LLM-Aided design RTL Generation," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2024, pp. 1–9.

[16] B. Nadimi, G. O. Boutaib, and H. Zheng, "Pyranet: A Multi-Layered Hierarchical Dataset for Verilog," in *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2025, pp. 1–7.

[17] L. Benini, G. De Micheli, E. Macii, M. Poncino, and R. Scarsi, "Symbolic Synthesis of Clock-Gating Logic for Power Optimization of Synchronous Controllers," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 4, no. 4, pp. 351–375, 1999.

[18] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "AutoBench: Automatic Testbench Generation and Evaluation Using LLMs for HDL Design," in *ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, 2024, pp. 1–10.

[19] ——, "CorrectBench: Automatic Testbench Generation with Functional Self-Correction Using LLMs for HDL Design," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*. IEEE, 2025, pp. 1–7.

[20] N. S. Murthy, E. Nelson, S. S. Sapatnekar, and J. Sartori, "VerifLLMBench: An Open-Source Benchmark for Testbenches Generated with Large Language Models," *DVCON*, pp. 1–13, 2025.

[21] S. Gangadharan and S. Churiwala, *Constraining Designs for Synthesis and Timing Analysis*. Springer, 2013.

[22] M. Wainberg and V. Betz, "Robust Optimization of Multiple Timing Constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 34, no. 12, pp. 1942–1953, 2015.

[23] O. Coudert, "An Efficient Algorithm to Verify Generalized False Paths," in *ACM/IEEE Design Automation Conference (DAC)*, 2010, pp. 188–193.

[24] P. C. McGeer and R. K. Brayton, *Integrating Functional and Temporal Domains in Logic Design: The False Path Problem and Its Implications*. Springer Science & Business Media, 2012, vol. 139.

[25] D. Du, S. H. Yen, and S. Ghanta, "On The General False Path Problem in Timing Analysis," in *ACM/IEEE Design Automation Conference (DAC)*, 1989, pp. 555–560.

[26] R. A. Bergamaschi, "The Effects of False Paths in High-Level Synthesis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1991, pp. 80–83.

[27] M. Nourani and C. A. Papachristou, "False Path Exclusion in Delay Analysis of RTL Structures," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 10, no. 1, pp. 30–43, 2002.

[28] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "GPT-4 Technical Report," *arXiv preprint*, 2023.

[29] K. Sanderson, "Gpt-4 is here: What scientists think," *Nature*, vol. 615, no. 7954, p. 773, 2023.

[30] Anthropic, "Introducing Claude Sonnet 4.5," https://www.anthropic.com/news/claude-sonnet-4-5, 2025.

[31] Z. He, H. Wu, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "ChatEDA: A Large Language Model Powered Autonomous Agent for EDA," in *ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, 2023, pp. 1–7.

[32] Y. Zhao, H. Zhang, H. Huang, Z. Yu, and J. Zhao, "MAGE: A Multi-Agent Engine for Automated RTL Code Generation," in *ACM/IEEE Design Automation Conference (DAC)*, 2025, pp. 1–7.

[33] G. Pasandi, K. Kunal, V. Tej, K. Shah, H. Sun, S. Jain, C. Li, C. Deng, T.-D. Ene, H. Ren *et al.*, "JARVIS: A Multi-Agent Code Assistant for High-Quality EDA Script Generation," *arXiv preprint*, 2025.

[34] C. Albrecht, "IWLS 2005 benchmarks," in *IEEE/ACM International Workshop on Logic Synthesis*, vol. 9, 2005.

[35] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm FinFET Predictive Process Design Kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.

[36] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer, 2015, pp. 451–460.