SynAct: A Reasoning-Acting Large Language Model Agent for Adaptive Synthesis Optimization

Fangzhou Liu, Peiyi Han, Jiawei Liu, Yuan Pu, Zhuolun He, Rongliang Fu, Tsung-Yi Ho, Bei Yu

***Abstract*—Logic synthesis transforms RTL designs into gate-level netlists, where PPA results are highly sensitive to the choice of optimization commands, making synthesis tuning both high-dimensional and expensive. Previous approaches fall into two categories: automated methods, which perform black-box search over fixed action spaces with limited decision-level interpretability, and LLM-based methods, which typically generate static scripts upfront and cannot adapt to evolving circuit states. We present SynAct, an adaptive closed-loop LLM reasoning–acting agent that iteratively diagnoses live synthesis reports and reasons over the current circuit state, retrieved tool knowledge, and historical optimization experience to issue targeted commands. SynAct focuses on improving timing, particularly worst negative slack (WNS), while maintaining balanced area and power trade-offs. Experiments on a commercial synthesis tool across 14 designs show that SynAct reduces average WNS to 27% of that from bootstrap synthesis.**

***Index Terms*—Electronic design automation, large language models, logic synthesis, retrieval-augmented generation.**

I. Introduction

LOGIC synthesis transforms RTL designs into gate-level netlists. As shown in Fig. 1, a typical flow comprises translation, logic optimization, and technology mapping, where the latter two stages involve extensive algorithmic choices that commercial tools encapsulate into large command sets with diverse configurable parameters. Prior studies show that the choice and ordering of commands significantly affect PPA outcomes [1], making synthesis tuning a high-dimensional and high-cost design-space exploration challenge. Among the PPA metrics, timing is particularly critical, as violations can necessitate flow re-tuning and additional place-and-route iterations, thereby substantially prolonging the design cycle [2].

Prior work on logic synthesis flow optimization falls into two paradigms, as shown in the upper right of Fig. 1. The first treats synthesis tuning as combinatorial search over AIG-level operator sequences on ABC [3], with methods spanning manual tuning, reinforcement learning (RL) [4], Bayesian optimization (BO) [5], [6], Monte Carlo Tree Search (MCTS) [7], and bandits [8]. Although effective, these methods are confined to fixed action spaces, rely on black-box rewards, and provide limited interpretability regarding how their decisions relate to the current circuit state. These limitations become more pronounced on commercial tools with far richer command vocabularies. The second paradigm emerges from the growing use of large language models (LLMs) in EDA. Works such as ChatEDA [9], ChipNeMo [10], and ChatLS [11] generate complete Tcl scripts or command sequences for commercial tools directly from user intent and tool documentation, thereby avoiding a fixed action space. However, these end-to-end methods rely on "one-shot" generation and produce complete static scripts upfront. As a result, later commands cannot adapt to evolving circuit states after each command, potentially leading to suboptimal results.

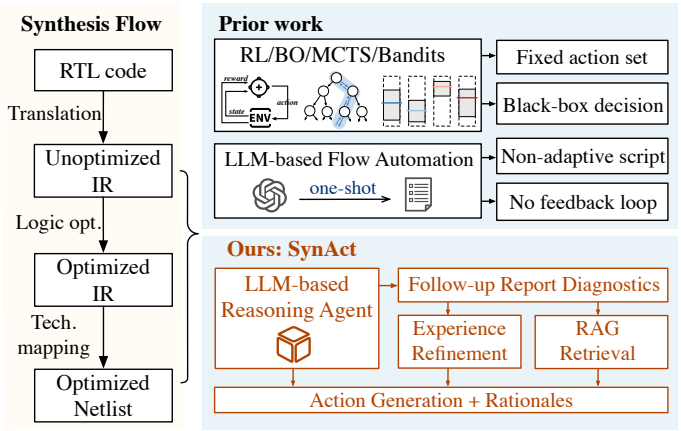

Fig. 1 Traditional synthesis flow vs. our LLM-based agent SynAct. "IR" denotes intermediate representation.

A key observation is that commercial synthesis tools support a continuous interaction model: the tool session remains open across commands, allowing an agent to issue commands and observe updated PPA reports after each iteration, making synthesis a natural fit for a closed-loop LLM agent. This suggests that rather than generating a script upfront or searching over a fixed action space, the agent should interact directly with the tool, diagnosing the current circuit state and retrieving relevant knowledge at each iteration to guide its next decision. However, two challenges arise: (1) at each iteration, the LLM must locate relevant content within extensive command documentation, where irrelevant information can cause reasoning errors; (2) pure LLM reasoning lacks systematic reuse of prior experience, making it difficult to reuse commands proven

Fangzhou Liu, Peiyi Han, Jiawei Liu, Yuan Pu, Zhuolun He, Rongliang Fu, Tsung-Yi Ho, and Bei Yu are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong SAR.

Bei Yu and Rongliang Fu are the corresponding authors (e-mail: byu@cse.cuhk.edu.hk; rlfu@cse.cuhk.edu.hk).

effective in previous iterations and leading to inefficient long-horizon exploration.

To address these challenges, we present **SynAct**, an adaptive closed-loop LLM reasoning–acting agent that iteratively optimizes PPA on a commercial synthesis tool, with particular emphasis on WNS. At each iteration, SynAct conditions its decision on the latest timing reports and diagnostic outputs, mitigating information overload through a multi-layer GraphRAG module [12] that retrieves scenario-relevant command documentation on demand. To improve exploration efficiency, Bayesian optimization over a GrammarVAE latent space leverages historical optimization experience accumulated in earlier iterations. Built on ReAct [13] for coupled reasoning and action and AutoGen-style multi-agent collaboration [14] for role coordination, SynAct continuously interprets circuit feedback and produces each optimization command with an explicit rationale, as illustrated in Fig. 1. Our main contributions are as follows:

1) A closed-loop LLM agent that iteratively optimizes PPA by adapting decisions to the current circuit state.
2) A multi-layer GraphRAG module for scenario-relevant knowledge retrieval, mitigating information overload.
3) A BO-guided experience refinement mechanism over a GrammarVAE latent space for experience-guided exploration.
4) SynAct reduces average WNS to 27% of that from bootstrap synthesis while maintaining balanced area and power trade-offs.

The remainder of this paper is organized as follows. Section II reviews the necessary background. Section III presents the SynAct framework, and Section IV details its knowledge- and experience-guidance modules. Section V provides a practical example of the framework in operation. Section VI reports the experimental results, and Section VII concludes the paper.

II. Preliminaries

A. Logic Synthesis

Logic optimization and technology mapping are key stages in logic synthesis, respectively restructuring Boolean networks and mapping them to library cells under timing, area, and power constraints. Numerous algorithms have been proposed for these tasks [3]. Commercial synthesis tools offer a much richer command space that includes combinational and sequential optimization, retiming, power-aware optimization, and physically aware synthesis [15]. These capabilities create a larger and more context-dependent action space that must be explored effectively to achieve high-quality PPA results.

Because each command alters the circuit state and subsequent actions rely on the updated netlist and PPA reports, synthesis optimization can be formulated as a Markov Decision Process (MDP) $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \mathcal{F}, \mathcal{R})$, forming the basis of our agent design.

- $\mathcal{X}$ is the state space, where each state x_t includes the current netlist, PPA summary, and detailed timing, area, and power reports.
- $\mathcal{A}$ is the action space of executable synthesis commands.
- $\mathcal{F} : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$ is the transition function realized by the synthesis tool under fixed settings.
- $\mathcal{R} : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ is the reward function, which uses a weighted aggregation of WNS and other PPA metrics to capture timing, area, and power trade-offs.

Bootstrap synthesis is the initialization flow that configures the technology library, reads and elaborates RTL, applies timing constraints, and runs the initial optimization. It excludes recipe-level tuning and yields the common initial state x_0 and reference PPA for all compared methods on each design. Starting from x_0, the agent iteratively selects one command a_t, executes it within the tool session, and observes the resulting reports as the next state x_{t+1} until the optimization target is met or the budget is exhausted.

B. LLMs for EDA

LLM-based EDA increasingly combines knowledge retrieval with agentic tool interaction. Retrieval-Augmented Generation (RAG) [16] grounds generation in retrieved knowledge, and its retrieval models can be fine-tuned on EDA tool documentation to improve domain-specific retrieval [12]. Beyond retrieval, ReAct [13] interleaves reasoning and action so that observations inform later decisions. AutoGen [14] supports coordination among specialized agents through structured conversations in complex workflows. In practice, ChatEDA [9] decomposes user requests before generating and executing RTL-to-GDSII scripts, while ChipNeMo [10] combines domain adaptation and retrieval for tasks including EDA script generation. ChatLS [11] uses retrieved tool knowledge and chain-of-thought prompting to customize synthesis scripts, whereas LLSM [17] uses EDA-guided prompting to integrate circuit information extracted from RTL with structural features. These studies show that LLMs can retrieve domain knowledge, reason about design tasks, coordinate specialized agents, and invoke EDA tools, laying the foundation for SynAct's closed-loop PPA optimization.

III. SynAct Framework

A. Overview

The left side of Fig. 2 shows the overall flow. Given an RTL design and designer inputs, SynAct first performs bootstrap synthesis to establish the initial circuit state. It then iterates through LLM-guided command generation, optimization execution, and termination checking until the request is satisfied or the iteration budget n_{iter} is exhausted. The resulting netlist is then written out.

The right side lists the designer inputs, including a request, the command manual, and environment variables. The request specifies the optimization preferences and quantitative PPA targets, while the manual and variables provide tool-specific information for command generation. SynAct can prioritize any selected PPA metric. In this work, we use **WNS** as the primary objective and treat the remaining metrics as secondary objectives.

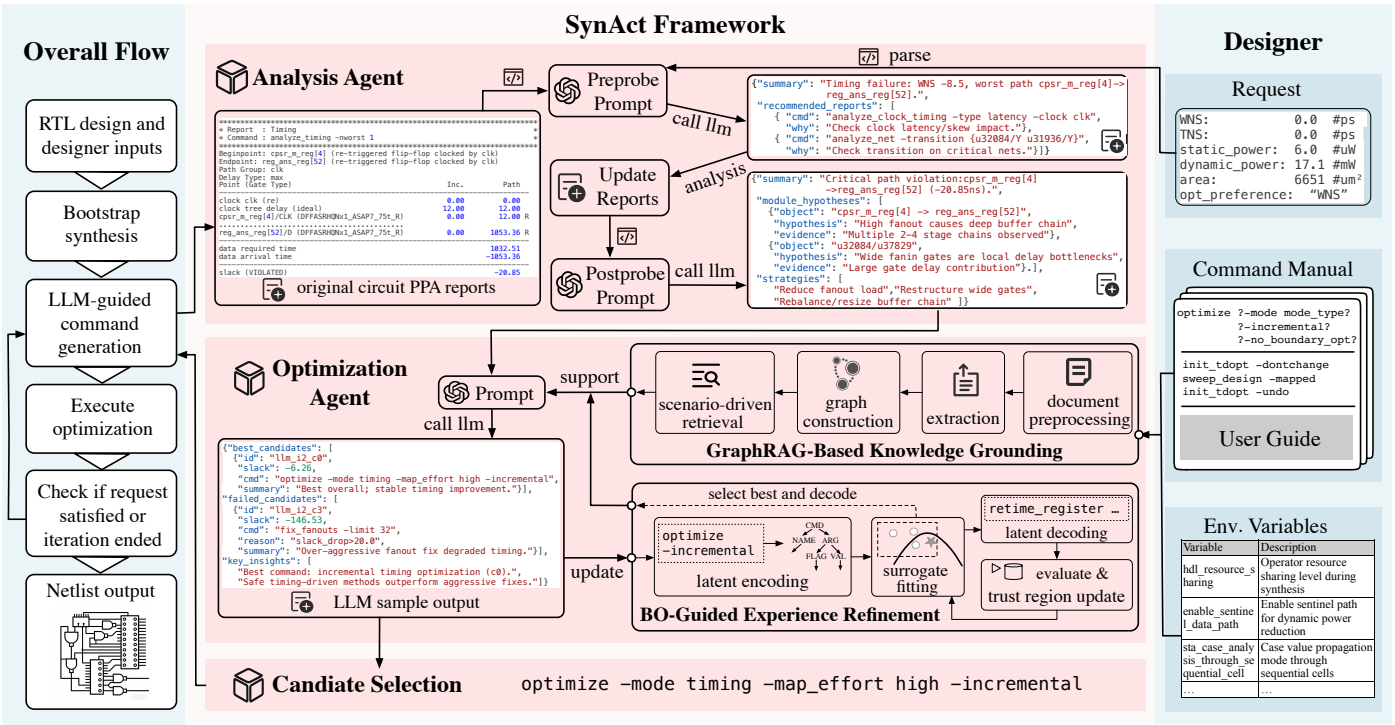

Fig. 2 The overall framework of SynAct.

The center of Fig. 2 details three main components of SynAct. Inspired by ReAct [13], the Analysis Agent examines the current reports and produces a diagnosis. The Optimization Agent combines this diagnosis with retrieved tool knowledge and historical experience to generate command candidates. Candidate Selection chooses the best safe command from the evaluated candidates for execution. Executing the command updates the circuit state, and the resulting reports inform the next iteration, forming a closed reasoning–acting loop. The following subsections detail these three components.

B. Analysis Agent

Existing search-based methods guide optimization primarily using scalar PPA summaries [5], [6], [8], but these summaries provide limited information, making it difficult to uncover the circuit-level causes of PPA violations. SynAct therefore introduces an Analysis Agent that inspects detailed timing, area, and power reports to diagnose potential performance bottlenecks through two stages: Preprobe and Postprobe.

In Preprobe, the agent parses raw report logs of the current design to identify performance bottlenecks. When the available data lacks sufficient detail, it proactively constructs follow-up probes using `analyze_*` commands described in the tool documentation to obtain the necessary information. For precise and context-aware diagnosis, the agent is guided to generate targeted commands for specific design objects such as clock domains, timing constraints, circuit instances, or buffer chains through the following prompt:

> **System:** role = analysis agent (PREPROBE); diagnose root causes and propose `analyze_*` probes; prefer object-level Tcl-braced probes
> **User:** user request, current/previous metrics, raw report log, and preprobe policy
> **Expected JSON:** { summary, recommended_probes }

A typical output may include probes such as:

```
analyze_clock_timing -type latency
analyze_inst -pin {...}
analyze_buffer_chain -from {...}
```

In Postprobe, the outputs collected by executing the probes from the previous stage are incorporated into the user prompt to help the agent refine its diagnosis. The refined result is then returned as a structured diagnosis that forms part of the Optimization Agent's input. For example, it may conclude that one path group dominates the timing bottleneck and that the next actions should focus on buffer optimization, cell sizing, or local restructuring under area and power guardrails. The corresponding Postprobe prompt is:

> **System:** role = analysis agent (POSTPROBE); refine diagnosis using collected probe evidence
> **User:** user request, raw report text, extra analysis reports
> **Expected JSON:** { summary, goals, violations, module_hypotheses, suggested_strategies, reflection }

C. Optimization Agent

The Optimization Agent takes the structured diagnosis and user request as input and generates executable synthesis command candidates. This constitutes our second key design:

rather than optimizing over a small fixed action set, SynAct performs context-aware command generation within a large, structured command space. This generation process is supported by two complementary guidance modules, which are detailed in Section IV:

- **RAG-grounded candidates** $\mathcal{C}_{\text{RAG}}$, which are derived from scenario-relevant manual sections retrieved by GraphRAG;
- **BO-seeded candidates** $\mathcal{C}_{\text{BO}}$, which adapt historically effective command patterns with minimal syntax changes.

By combining these sources, the agent generates commands that reflect the current diagnosis while leveraging retrieved knowledge and prior experience. Each candidate is accompanied by a concise rationale. The corresponding structured prompt is:

System: role = optimization agent; generate command candidates
User: user request, Postprobe diagnosis, RAG-retrieved manual sections, BO seed recommendation
Expected JSON: { recommended_candidates, rationale }

D. Candidate Selection

Given the candidates produced by the Optimization Agent, SynAct first performs safety filtering, removing those whose WNS falls beyond a preset threshold (e.g., -20 ps) or whose reward drops below $0.2 \times r_{\text{boot}}$, where r_{boot} is the initial reward from bootstrap synthesis. Among the remaining safe candidates, it then selects the one with the highest score (defined below). If no candidate passes filtering, the previous checkpoint is retained.

Each candidate command C is evaluated by the score

$$\text{score}(C) = r(C) + \kappa\sigma(z) - \rho(C), \tag{1}$$

where $r(C)$ is the reward measuring objective satisfaction, κ controls exploration strength, and $\rho(C)$ penalizes recent repetitions (waived for large WNS improvements). Here $\sigma(z)$ represents a local uncertainty estimate from the BO surrogate at latent point z, encouraging exploration of under-explored regions. All candidates, whether from $\mathcal{C}_{\text{BO}}$ or $\mathcal{C}_{\text{RAG}}$, are encoded into this shared latent space, enabling the surrogate to generalize learned experience and provide consistent exploration guidance across both sources. The latent space and surrogate model are detailed in Section IV-B. This score balances high reward, moderate exploration, and redundancy avoidance.

The reward $r \in (0, 1]$ aggregates violations across all objectives:

$$r = \exp\left(-\sum_i w_i \log(1 + \text{err}_i)\right), \tag{2}$$

$$\text{err}_i = \begin{cases} \max(0,\ \text{target}_i - \text{metric}_i), & i \in \{\text{WNS}, \text{TNS}\}, \\ \dfrac{\max(0,\ \text{metric}_i - \text{target}_i)}{|\text{target}_i|}, & i \in \{\text{area}, P_{\text{dyn}}, P_{\text{stat}}\}, \end{cases} \tag{3}$$

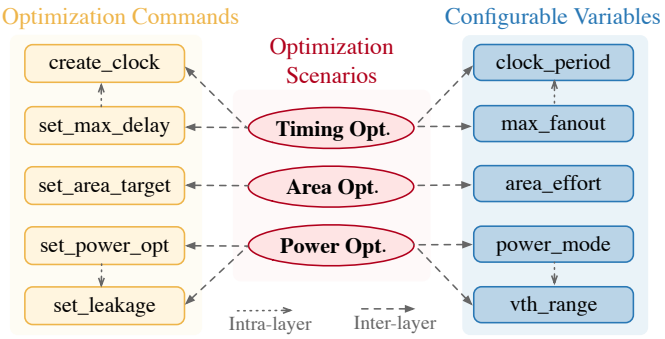

Fig. 3 Three-layer structure of the proposed GraphRAG framework. The optimization scenario layer (center) connects related commands (left) and configurable variables (right), forming inter- and intra-layer relationships.

where target_i is the user-specified target and metric_i is the current measured value. Timing metrics (WNS, TNS) are maximized, while area and power are minimized and normalized by target magnitude. The $\log(1+\text{err}_i)$ term compresses large violations to reduce their influence in the aggregate reward. Each weight w_i is assigned a base value, upweighted for the user-selected primary objective, and then normalized to ensure $\sum_i w_i = 1$. Thus $r = 1$ when all targets are met and decreases faster for larger or higher-priority violations.

IV. Knowledge- and Experience-Guided Optimization

The Optimization Agent is supported by two complementary modules shown in Fig. 2: GraphRAG grounds candidate generation in scenario-relevant tool knowledge, while BO reuses historical synthesis experience in a GrammarVAE latent space. This section details these two modules.

A. GraphRAG-Based Knowledge Grounding

The Optimization Agent requires precise, context-aware domain knowledge to generate effective synthesis commands. A natural approach is to retrieve relevant documentation via RAG; however, forming effective synthesis strategies requires connecting applicable scenarios, specific commands, and configurable variables scattered across disparate documentation sections. This structured, multi-hop reasoning process goes beyond the ability of traditional vector-similarity retrieval. To address this, we develop a multi-layer **GraphRAG module** that unifies graph-based reasoning and retrieval augmentation [18], [19], detailed in the following four parts.

Document Preprocessing. Building the GraphRAG module begins with extracting relevant content from the synthesis tool documentation. We manually curate document sections related to optimization commands and categorize them into three functional roles:

1) Applicable scenarios: describe design goals (e.g., timing, area, power) with typical associated commands and variables;

2) Executable commands: define specific tool operations, including syntax, structure, and invocation patterns;
3) Configurable variables: specify tunable parameters, valid ranges, and default configurations.

This taxonomy follows practical EDA usage: optimizations are scenario-driven, executed through commands, and tuned by variables. Based on this categorization, we organize the extracted content into a three-layer graph of scenarios, commands, and variables.

Entity and Relationship Extraction. An LLM is prompted to convert the unstructured text into structured entities belonging to the three layers defined above. We introduce three disjoint sets of entities corresponding to the three layers, where $\mathcal{E}_S = \{s_1, s_2, \ldots, s_{n_S}\}$, $\mathcal{E}_C = \{c_1, c_2, \ldots, c_{n_C}\}$, and $\mathcal{E}_V = \{v_1, v_2, \ldots, v_{n_V}\}$ represent scenarios, commands, and variables, respectively.

The semantic structure of the knowledge graph is built through two complementary processes: intra-layer and inter-layer relationship identification. For intra-layer relationships, we use an LLM-based identification function to detect semantic connections within a single layer, defined as follows:

$$\mathcal{R}^{(intra)} = \mathcal{R}_{CC} \cup \mathcal{R}_{VV}, \tag{4}$$

$$\mathcal{R}_{CC} = \{(c_i, c_j, r_{ij}) \mid c_i, c_j \in \mathcal{E}_C, \text{LLM}_{rel}(c_i, c_j) = \text{True}\}, \tag{5}$$

$$\mathcal{R}_{VV} = \{(v_i, v_j, r'_{ij}) \mid v_i, v_j \in \mathcal{E}_V, \text{LLM}_{rel}(v_i, v_j) = \text{True}\}. \tag{6}$$

Here, $\text{LLM}_{rel}(\cdot, \cdot)$ determines whether two entities share a semantic dependency based on their descriptions and usage patterns.

For inter-layer relationships, scenarios are linked to related commands and variables that co-occur in documentation examples, formalized as follows:

$$\mathcal{R}^{(inter)} = \mathcal{R}_{SC} \cup \mathcal{R}_{SV}, \tag{7}$$

$$\mathcal{R}_{SC} = \{(s, c) \mid s \in \mathcal{E}_S, c \in \mathcal{E}_C, c \in \mathcal{C}_s^{(cmds)}\}, \tag{8}$$

$$\mathcal{R}_{SV} = \{(s, v) \mid s \in \mathcal{E}_S, v \in \mathcal{E}_V, v \in \mathcal{C}_s^{(vars)}\}. \tag{9}$$

A link (s, c) or (s, v) is established when the LLM identifies co-occurrence between scenario s and the corresponding command or variable in application examples.

Hierarchical Knowledge Graph Construction. Based on the extracted entities and relations, we construct a hierarchical knowledge graph defined as:

$$\mathcal{G} = (\mathcal{E}, \mathcal{R}), \tag{10}$$

where $\mathcal{E} = \mathcal{E}_S \cup \mathcal{E}_C \cup \mathcal{E}_V$ denotes the set of all entities, which is partitioned into disjoint subsets of scenarios, commands, and variables, and $\mathcal{R} = \mathcal{R}^{(\text{intra})} \cup \mathcal{R}^{(\text{inter})}$ represents intra-layer and inter-layer relations. The scenario layer acts as the central hub connecting the command and variable layers, providing the foundation for subsequent reasoning and retrieval, as illustrated in Fig. 3.

Scenario-Driven Retrieval. Before retrieval, the **Analysis Agent** diagnoses optimization bottlenecks and generates an analysis report $\mathcal{A}$, which serves as the input query. Given $\mathcal{A}$, GraphRAG retrieves relevant synthesis knowledge via four compact steps:

1) Scenario description generation: An LLM summarizes an optimization scenario description d_{scene} from $\mathcal{A}$.
2) Semantic embedding: scenario description d_{scene} and scenario entity s are encoded into vectors using a domain-customized text embedding model [12], fine-tuned on EDA tool documentation via supervised contrastive learning. This model better captures EDA terminology and semantics than general-purpose embeddings, ensuring all entities/descriptions in GraphRAG share a consistent EDA-aware vector space for precise retrieval.
3) Scenario selection: Top-k relevant scenarios are retrieved by semantic similarity:

$$\mathcal{S}^* = \underset{\mathcal{S} \subseteq \mathcal{E}_S, |\mathcal{S}|=k}{\arg\max} \sum_{s \in \mathcal{S}} \text{sim}(\text{Embed}(d_{\text{scene}}), \text{Embed}(s)). \tag{11}$$

4) Knowledge expansion: Retrieve commands and variables directly connected to $\mathcal{S}^*$ as the initial associated set $\mathcal{C}_0/\mathcal{V}_0$, and further retrieve their top-k_{intra} intra-layer neighbors via k-nearest neighbor (KNN) to form the final expanded set. Specifically:

$$\mathcal{C}_0 = \{c \mid (s, c) \in \mathcal{R}_{SC}, s \in \mathcal{S}^*\}, \ \mathcal{C}^* = \mathcal{C}_0 \cup \text{KNN}(\mathcal{C}_0, k_{\text{intra}}), \tag{12}$$

$$\mathcal{V}_0 = \{v \mid (s, v) \in \mathcal{R}_{SV}, s \in \mathcal{S}^*\}, \ \mathcal{V}^* = \mathcal{V}_0 \cup \text{KNN}(\mathcal{V}_0, k_{\text{intra}}). \tag{13}$$

Here, k_{intra} denotes the number of intra-layer neighbors retrieved for each seed entity.

The final retrieved knowledge set is $\mathcal{K} = \mathcal{S}^* \cup \mathcal{C}^* \cup \mathcal{V}^*$, providing the Optimization Agent with concise, scenario-specific, non-redundant EDA synthesis knowledge.

The hierarchical GraphRAG mechanism tightly couples knowledge grounding with design diagnosis, which allows the Optimization Agent to leverage structured, reliable synthesis knowledge for improved reasoning accuracy and decision reliability.

B. BO-Guided Experience Refinement

Commands validated in earlier iterations provide design-specific evidence for promising optimization directions [5], [6], [8]. Inspired by this, the Optimization Agent reuses executed commands and their feedback as experience. In practice, reapplying a previously effective command can yield further gains and sometimes outperform one newly generated from retrieved documentation. This motivates an experience-guided search based on Bayesian optimization (BO), which uses accumulated evaluations to efficiently identify promising commands.

Applying BO directly to discrete command strings remains inefficient because the search space is large and similar commands can exhibit different optimization behavior. Inspired by

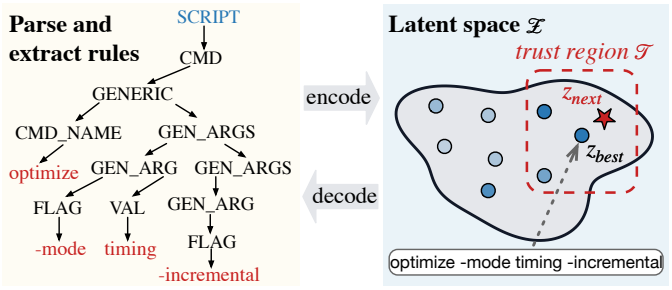

Fig. 4 GrammarVAE encoding and latent-space BO. A command is parsed into a grammar tree and encoded into latent space $\mathcal{Z}$; BO then identifies z_{best} and proposes z_{next} within the trust region $\mathcal{T}$, which is decoded back into a command recommendation for the Optimization Agent.

Algorithm 1 BO-Guided Experience Refinement

Input: Experience log $\mathcal{D}$, trust region $\mathcal{T}$, last commit reward r_{prev}, diagnosis Φ, RAG docs $\mathcal{R}$
Output: Updated log $\mathcal{D}'$, updated trust region $\mathcal{T}'$, BO seed $\hat{C}_{\text{BO}}$

1: /* **Surrogate Fitting** */
2: $w_i \leftarrow \min(w_{\max}, 1+\lambda \max(0, r_i))$ **for each** $(z_i, r_i) \in \mathcal{D}$
3: Refit RBF surrogate on $\mathcal{D}$ to obtain $\mu(z)$, $\sigma(z)$
4: /* **UCB Acquisition and Latent Decoding** */
5: Sample pool $\mathcal{P} \subset \mathcal{T}$
6: $z_{\text{next}} \leftarrow \arg\max_{z \in \mathcal{P}} [\mu(z) + \kappa_{\text{acq}}\, \sigma(z)]$
7: $\hat{C}_{\text{BO}} \leftarrow \text{Dec}(z_{\text{next}})$ ▷ GrammarVAE decode
8: $\mathcal{C} \leftarrow \text{OptAgent}(\hat{C}_{\text{BO}}, \Phi, \mathcal{R})$
9: /* **Evaluation and Trust Region Update** */
10: $\mathcal{D}' \leftarrow \mathcal{D}$
11: **for** $C \in \mathcal{C}$ **do**
12: Execute C; record $r(C)$
13: $z \leftarrow \text{Enc}(\text{Norm}(C))$ ▷ GrammarVAE encode
14: $\mathcal{D}' \leftarrow \mathcal{D}' \oplus (z, r(C))$ ▷ $\oplus$: insert or replace
15: **end for**
16: $\mathcal{C}_{\text{safe}} \leftarrow \{C \in \mathcal{C} \mid C \text{ passes safety filter}\}$ ▷ Section III-D
17: $C^\star \leftarrow \arg\max_{C \in \mathcal{C}_{\text{safe}}} \text{score}(C)$ ▷ see Equation (1)
18: $z_{\text{best}} \leftarrow \text{Enc}(\text{Norm}(C^\star))$
19: Recenter $\mathcal{T}'$ on z_{best}; expand if $r(C^\star) > r_{\text{prev}}$, else shrink
20: **return** $(\mathcal{D}', \mathcal{T}', \hat{C}_{\text{BO}})$

grammar-based VAEs that map discrete grammatical structures into continuous spaces [20], [21], we adapt GrammarVAE to encode synthesis commands for BO. This representation enables smooth optimization while preserving syntactic validity [22]. BO then iteratively fits a surrogate model, proposes and evaluates candidates, and updates its data to focus on empirically effective regions.

Latent Encoding. Each synthesis command C is normalized by $\text{Norm}(\cdot)$ and encoded into a latent vector $z = \text{Enc}(\text{Norm}(C)) \in \mathbb{R}^d$ via the GrammarVAE encoder [20], where z is a point in latent space $\mathcal{Z} \subseteq \mathbb{R}^d$. As shown in Fig. 4, a command is first parsed into a context-free grammar tree, then compressed into z, such that nearby points in $\mathcal{Z}$ correspond to grammatically similar commands. Because synthesis-command syntax encodes the optimization mode, transformation scope, effort level, and parameter configuration, these grammar-preserving neighborhoods provide a structured local prior that BO calibrates with measured synthesis rewards. GrammarVAE is pre-trained offline on a corpus of synthesis commands using the standard VAE objective (reconstruction and KL regularization); during BO, each command is mapped to the encoder's posterior mean for a consistent latent representation.

Definition 1 (Trust Region $\mathcal{T}$)**.** *BO limits its search to an axis-aligned trust region $\mathcal{T} \subseteq \mathcal{Z}$, centered at the latent point of the current best command z_{best}, to focus optimization on the most promising area. The radius of $\mathcal{T}$ adapts across iterations: it expands when performance improves and shrinks otherwise.*

Definition 2 (Experience Log $\mathcal{D}$)**.** *The experience log $\mathcal{D} = \{(z_i, r_i)\}_{i=1}^N$ stores the latent encoding and reward of each executed command C_i, with $z_i = \text{Enc}(\text{Norm}(C_i))$ and r_i defined in Equation (2). During bootstrap synthesis, $\mathcal{D}$ is initialized from the default* `optimize` *command, and $\mathcal{T}$ is centered at its latent point.*

Algorithm 1 illustrates how the refinement procedure is executed within one iteration: it takes $\mathcal{D}$, $\mathcal{T}$, last commit reward r_{prev}, the Analysis Agent's diagnosis Φ, and RAG-retrieved documents $\mathcal{R}$ as input, and returns updated $\mathcal{D}'$, $\mathcal{T}'$, and BO seed $\hat{C}_{\text{BO}}$.

Surrogate Fitting. At each iteration, a reward-weighted RBF surrogate [23] is refit from scratch on the full $\mathcal{D}$ (lines 2–3). Each observation (z_i, r_i) is assigned a weight $w_i = \min(w_{\max}, 1 + \lambda \max(0, r_i))$, where $\lambda \geq 0$ scales how aggressively positive rewards upweight a point in the fit and $w_{\max}$ caps each w_i, so that high-reward regions receive greater influence while no single sample dominates. The surrogate estimates the expected reward and local uncertainty through kernel-weighted averages:

$$\mu(z) = \frac{\sum_i w_i K(z, z_i) r_i}{\sum_i w_i K(z, z_i)}, \quad \sigma^2(z) = \frac{\sum_i w_i K(z, z_i)[r_i - \mu(z)]^2}{\sum_i w_i K(z, z_i)}, \tag{14}$$

where $K(z, z_i) = \exp(-\|z - z_i\|^2 / 2\ell^2)$ is the RBF kernel. The resulting $\sigma(z)$ is smoothed and clipped for numerical stability and serves as the local uncertainty estimate used by the commitment score described in Section III-D.

UCB Acquisition and Latent Decoding. BO samples a pool $\mathcal{P} \subset \mathcal{T}$ and selects the next latent point by maximizing the UCB acquisition function [24] (lines 5–6):

$$\alpha(z) = \mu(z) + \kappa_{\text{acq}}\, \sigma(z), \tag{15}$$

where $\mu(z)$ exploits known high-reward regions and $\kappa_{\text{acq}}\, \sigma(z)$ promotes exploration of regions with high local uncertainty. The selected z_{next} is decoded by GrammarVAE into a command string $\hat{C}_{\text{BO}}$ (line 7) and passed to the Optimization Agent as the BO seed recommendation. While GrammarVAE

decoding enforces grammatical validity, the decoded command may still contain tool-level errors (e.g., unsupported flags or invalid parameter ranges). The Optimization Agent retains $\hat{C}_{\mathrm{BO}}$ as an explicit conditioning input and refines it with retrieved tool documentation to produce $\mathcal{C}_{\mathrm{BO}}$, while independently generating $\mathcal{C}_{\mathrm{RAG}}$ from the retrieved knowledge; together they form the full candidate set $\mathcal{C} = \mathcal{C}_{\mathrm{BO}} \cup \mathcal{C}_{\mathrm{RAG}}$ (line 8).

Evaluation and Trust Region Update. All candidates in $\mathcal{C}$ are executed from the same parent checkpoint and their rewards recorded (lines 11–12). Each candidate is then re-encoded and logged into $\mathcal{D}'$ (lines 13–14), including those that fail or yield poor results, as broader latent-space coverage helps BO distinguish effective from ineffective regions and improve future proposals. The committed command $C^\star$ is selected by first filtering $\mathcal{C}$ for safe candidates $\mathcal{C}_{\mathrm{safe}}$ (line 16), and then maximizing the score function $\mathrm{score}(C)$ defined in Equation (1) (line 17). The trust region $\mathcal{T}'$ is recentered on $z_{\mathrm{best}} = \mathrm{Enc}(\mathrm{Norm}(C^\star))$ and expanded if $r(C^\star) > r_{\mathrm{prev}}$, otherwise shrunk (lines 18–19).

V. Running Example

To illustrate the practical use of SynAct, Fig. 5 presents a five-iteration run on *arm9*. Starting from the initial synthesis report, SynAct follows a reasoning, acting, and observation loop. At each iteration, the LLM interprets the tool feedback, explains the optimization intent, and produces an executable Tcl command. After execution, the updated circuit state is reflected in new synthesis reports, which serve as observations for the next iteration.

In the first two iterations, the LLM follows BO guidance and selects incremental timing optimization to limit changes to the current implementation. It then uses updated timing observations and retrieved tool knowledge to select `tdopt`, register retiming, and phase assignment. The figure records the reasoning, committed command, and PPA observation at each iteration. Over the five iterations, WNS changes from -20.85 ps to -3.70 ps.

(Example) SynAct Optimization Trajectory of case *arm9* ($k = 5$)

Bootstrap synthesis

Reasoning: Bootstrap phase: Initial design setup and baseline transformation via a default `optimize`.

Act 0: `set db_path ...; set rtl_path ...; read_db ...; read_design ...; create_clock -period 1050 ...; verify_design; optimize`

Obs 0: wns=-20.85, tns=-419.62, area=2031.81, P_{dyn}=1.821 mW, P_{static}=2.078 μW

Iteration 1

Reasoning: BO-guided incremental timing optimization with high mapping effort to address the critical timing violation while minimizing disruption to the current implementation.

Act 1: `optimize -mode timing -map_effort high -incremental`

Obs 1: wns=-8.50, tns=-118.71, area=2031.61, P_{dyn}=1.816 mW, P_{static}=2.072 μW

Iteration 2

Reasoning: BO-recommended incremental timing optimization with high mapping effort to improve timing without major structural changes. This conservative approach maintains current structure while focusing on timing-critical paths.

Act 2: `optimize -mode timing -map_effort high -incremental`

Obs 2: wns=-6.26, tns=-65.84, area=2029.78, P_{dyn}=1.815 mW, P_{static}=2.065 μW

Iteration 3

Reasoning: RAG-guided: Focused critical path optimization using `tdopt` with high effort and area constraint. The analysis shows a specific critical path with multiple high-incidence cells that need aggressive timing optimization.

Act 3: `tdopt -flat -resyn2 -area 1.000000`

Obs 3: wns=-4.03, tns=-41.65, area=2035.0, P_{dyn}=1.818 mW, P_{static}=2.078 μW

Iteration 4

Reasoning: RAG-guided sequential retiming optimization. Attempts register retiming to improve timing by moving registers across combinational logic, which could help reduce the long critical path identified in the analysis.

Act 4: `retime_register -period 0 -effort high`

Obs 4: wns=-3.85, tns=-39.64, area=2038.9, P_{dyn}=1.820 mW, P_{static}=2.100 μW

Iteration 5

Reasoning: RAG-guided candidate: Phase assignment optimization for timing recovery, particularly effective for buffer-heavy designs. This can help optimize signal phases in the complex buffer chains identified in the analysis.

Act 5: `phase_assign -flat -mode timing -effort medium`

Obs 5: wns=-3.70, tns=-37.42, area=2038.79, P_{dyn}=1.819 mW, P_{static}=2.100 μW

Fig. 5 An actual SynAct run on *arm9*, showing the reasoning, committed action, and tool observation at each iteration.

VI. Experiment

A. Experimental Setup

All experiments are conducted on a workstation equipped with an Intel® Xeon® Gold 6226R CPU (2.90 GHz) and an NVIDIA GeForce RTX 3090 GPU. SynAct is implemented in Python with PyTorch for neural modules and connects to large language models via an OpenAI-compatible API supporting both commercial and internal endpoints. It integrates with AltiSyn®, the commercial logic synthesis tool from ZeniSyn Design Systems [25], for automated command execution and report collection. The synthesis flow employs the standard-cell library from the 7 nm ASAP7 edge-cut PDK [26].

SynAct is implemented as a multi-agent system comprising an Analysis Agent for report-driven diagnosis and an Optimization Agent for command generation. The Optimization Agent performs Bayesian optimization in a GrammarVAE latent space to efficiently explore discrete synthesis commands and utilizes a Neo4j-based multi-layer GraphRAG module for structured knowledge retrieval. Semantic retrieval within this module uses the transformer model fine-tuned for EDA documentation in [12].

For evaluation, we use 14 open-source RTL designs collected from OpenCores [27]. TABLE I summarizes the benchmark statistics obtained after importing each design and running a bootstrap synthesis pass, including the number of nets, leaf, sequential, buffer, and inverter cell instances, and the configured clock period (CPD). Appropriate clock periods are assigned according to design size and fixed across all methods to ensure sufficient optimization margin. Across all experiments, we set WNS as the primary optimization objective, with quantitative targets specified at the input stage (a configuration example is shown in Fig. 2). Additionally, we track TNS, area, dynamic power, and static power as secondary metrics to assess overall PPA quality. To account for stochastic variation, each experiment is repeated five times, and the average results are reported unless otherwise specified.

TABLE I Benchmark statistics.

Benchmarks	#Nets	#Leaf	#Seq.	#Buf.	#Inv.	CPD.[1]
uart16550	3057	3006	605	143	336	310
picorv32	9193	9091	1556	268	688	390
yacc	10504	10363	1102	487	1285	560
aes_core	11816	11557	530	521	1091	365
wb2axip	12539	11381	1900	257	1221	335
wb_conmax	18170	17040	770	492	1354	365
arm9	19267	19195	1222	823	2634	1050
sha3	24927	24161	2949	674	2903	365
spimaster	38365	38313	8705	862	3909	320
ethernet	44765	44637	10543	1097	3390	385
ecg	53613	53071	7676	1116	7807	340
linkruncca	69138	69134	16606	1720	4879	480
xge_mac	81564	81358	21667	29	5896	135
fft256	198981	198897	42432	3129	21095	490

[1] CPD: Clock period (ps).

TABLE II Original PPA metrics after bootstrap synthesis.

Benchmarks	WNS(ps)	TNS(ps)	Area (μm^2)	DP (mW)[1]	SP (μW)[2]
uart16550	-7.45	-10.10	400.94	1.572	0.392
picorv32	-15.70	-3132.97	1128.68	3.529	1.248
yacc	-26.86	-1592.03	1153.77	4.095	1.300
aes_core	-8.03	-482.04	1192.06	3.695	1.316
wb2axip	-20.54	-12344.79	1325.09	5.158	1.169
wb_conmax	-11.35	-3840.85	1698.47	4.399	1.481
arm9	-20.85	-419.62	2031.81	1.821	2.078
sha3	-14.41	-3446.74	2892.50	17.930	2.874
spimaster	-16.96	-151.50	5459.98	11.290	6.182
ethernet	-9.83	-213.09	6435.68	23.480	7.433
ecg	-5.04	-205.58	5873.96	22.760	5.765
linkruncca	-41.40	-146498.44	10380.14	28.330	12.290
xge_mac	-5.86	-43.86	12747.57	7.556	14.090
fft256	-17.14	-550.53	26638.81	72.630	31.600
Ratio Avg.	100.00%	100.00%	100.00%	100.00%	100.00%

[1] Dynamic Power. [2] Static Power.

B. Overall PPA Improvement

Comparison setup and method reproduction. TABLE II reports the PPA metrics after the mandatory bootstrap synthesis, which configures the technology library and timing constraints, reads and elaborates each RTL design, and runs the default `optimize` command without additional recipe-level tuning. These metrics serve as the reference for subsequent comparisons. The "Ratio Avg." is the mean of the per-design ratios between each method and bootstrap synthesis. Each ratio is computed by dividing the method result by its bootstrap value. For WNS and TNS, the reported value is first converted to its violation magnitude, $\max(0, -\text{slack})$, so timing closure contributes zero. A lower "Ratio Avg." indicates greater improvement.

TABLE III compares two baselines representing distinct paradigms of synthesis optimization. We set SynAct's iteration limit to $n_{\text{iter}} = 5$. To control optimization depth, all methods commit five actions, each consisting of a command or command sequence. This equalizes the committed-action budget but not the oracle-query budget, because the methods require different numbers of candidate evaluations, as detailed below. We therefore report the end-to-end runtime in TABLE IV alongside the PPA results.

ChatLS [11] is an LLM-based end-to-end script customization method that combines multimodal RAG and chain-of-thought reasoning. As its one-shot script cannot use feedback from intermediate synthesis results, it provides a natural counterpart to SynAct's iterative optimization. Since its source code is unavailable, we reproduce ChatLS following the published methodology. Under its original seven-benchmark 45 nm FreePDK setup, our reproduction matches every reported result within 5%. We then port it without further tuning to our 14-design setup using ASAP7 and AltiSyn®, retaining the same prompts, retrieval configuration, and five-step script. The resulting comparison represents our controlled reproduction rather than the official implementation.

CBTune [8] is a LinUCB-based contextual bandit method originally developed for synthesis sequence generation in ABC [3]. We reproduce it using its public implementation.[1] Unlike ChatLS and SynAct, LinUCB requires predefined, enumerable arms for reward estimation and action selection, so adapting CBTune to AltiSyn®requires bounding the tool's much larger command space. We manually construct the seven entries in TABLE V from documented AltiSyn®commands and the optimization categories used by CBTune on ABC. They cover timing optimization, mapping, sizing, phase assignment, retiming, and resynthesis, and remain fixed across all 14 benchmarks without design-specific tuning. At each of five steps, CBTune evaluates the arms, updates LinUCB using short- and long-term rewards, and applies the selected arm.

SynAct performs five sequential iterations without a fixed action list, enabling it to reason over a broader command space than CBTune. In each iteration, the Analysis Agent diagnoses the latest synthesis reports, and the Optimization Agent combines this diagnosis with GraphRAG-retrieved documentation and BO-based experience to generate ten candidates. After synthesis evaluation, SynAct commits the best safe command to optimize the circuit and feeds the updated reports into the next iteration. Both agents use DeepSeek V3.1.

Optimization results with WNS as the primary objective. As shown in TABLE III, with **WNS** as the primary optimization objective, SynAct achieves the best overall timing improvement among all methods. In terms of "Ratio Avg.", SynAct reduces the remaining WNS violation to 27.03% of that from bootstrap synthesis, substantially outperforming ChatLS [11] (71.73%) and CBTune [8] (66.67%). It achieves nonnegative WNS on two designs, whereas neither baseline does. Notably, for *wb2axip*, SynAct achieves WNS of -6.78 ps versus -9.23 ps (ChatLS) and -16.22 ps (CBTune); for *arm9*, it reaches -3.70 ps versus -8.25 ps and -8.44 ps, respectively. For *picorv32* and *wb_conmax*, SynAct further approaches timing closure with WNS values of 0.06 ps and 0.01 ps.

For secondary metrics, TNS is assigned lower priority than WNS and tends to improve as WNS becomes tighter. SynAct reduces TNS to 17.86% of the bootstrap synthesis result, compared with 96.43% for ChatLS and 77.14% for CBTune. In terms of area and power trade-offs, SynAct keeps metrics close to the bootstrap synthesis levels, with slight reductions in area and power (99.28% area, 98.92% dynamic

[1] https://github.com/sallyliu921/CB-EVO

TABLE III Five-run average PPA comparison with **WNS** as the primary objective and TNS/area/power as secondary metrics.

Benchmarks	**WNS** (ps)			TNS (ps)			Area (μm^2)			Dynamic Power (mW)			Static Power (μW)		
	[11]	[8]	SynAct	[11]	[8]	SynAct	[11]	[8]	SynAct	[11]	[8]	SynAct	[11]	[8]	SynAct
uart16550	-6.62	-2.04	**-1.18**	-6.69	-2.04	**-1.30**	403.57	**393.19**	397.73	1.575	**1.570**	1.573	0.394	**0.384**	0.391
picorv32	-11.92	-2.94	**0.06**	-329.10	-268.32	**0.00**	**1123.33**	1146.73	1123.56	**3.489**	3.524	3.510	**1.190**	1.287	1.276
yacc	-12.42	-18.61	**-4.38**	-595.46	-996.20	**-68.27**	**1155.38**	1155.90	1159.37	**4.115**	4.162	4.150	1.306	1.333	**1.304**
aes_core	-9.83	-3.68	**-3.41**	-869.29	**-108.98**	-118.37	1249.87	1210.61	**1194.67**	4.020	3.765	**3.703**	1.390	1.369	**1.316**
wb2axip	-9.23	-16.22	**-6.78**	-4170.73	-8771.95	**-3488.09**	1393.82	1523.99	**1328.59**	5.498	5.573	**5.162**	1.222	1.687	**1.173**
wb_conmax	-12.20	-11.12	**0.01**	-3562.51	-4704.21	**0.00**	1700.23	1686.60	**1684.22**	4.141	4.392	**4.040**	1.368	1.469	**1.314**
arm9	-8.25	-8.44	**-3.70**	-120.85	-149.88	**-37.42**	2022.95	**1971.41**	2038.79	1.812	**1.785**	1.819	2.059	**1.983**	2.100
sha3	-14.33	-6.32	**-3.82**	-8183.41	-1849.45	**-705.50**	4079.15	2901.76	**2837.75**	30.540	**15.380**	17.210	3.979	3.261	**2.761**
spimaster	-13.51	-4.47	**-3.04**	-142.14	**-35.01**	-53.20	5448.90	5428.24	**5415.65**	11.280	11.270	**11.240**	6.165	6.166	**6.128**
ethernet	-3.56	-6.53	**-2.85**	-304.41	-67.81	**-25.05**	6438.15	6449.11	**6406.14**	23.620	**23.280**	23.320	**7.080**	7.465	7.378
ecg	-3.35	-4.56	**-2.23**	**-22.11**	-453.69	-33.09	**5838.97**	6304.30	5842.29	**22.650**	23.960	22.680	**5.714**	7.089	**5.714**
linkruncca	-6.40	**-3.14**	-6.19	-12595.18	**-801.79**	-5865.33	**9832.23**	9950.35	9876.56	**27.390**	27.770	27.660	**11.520**	11.870	11.730
xge_mac	-6.30	-7.04	**-4.36**	-34.54	-35.47	**-25.77**	**12613.77**	12622.96	12668.15	**7.478**	7.540	7.541	13.970	**13.960**	14.010
fft256	-12.68	-34.27	**-7.90**	-1805.46	-1794.78	**-137.37**	26628.94	**26475.57**	26617.73	**72.660**	72.700	**72.660**	31.590	**31.010**	31.570
Ratio Avg.[1]	71.73%	66.67%	**27.03%**	96.43%	77.14%	**17.86%**	103.14%	101.02%	**99.28%**	105.33%	99.79%	**98.92%**	101.67%	105.51%	**98.64%**

[1] "Ratio Avg." averages per-design metric ratios relative to bootstrap synthesis; lower values indicate greater improvement.

TABLE IV Runtime and token usage averaged over five runs.

Benchmark	Total execution time (s)				Token usage	
	Boot.[1]	[11]	[8]	SynAct	Anal.[2]	Opt.[3]
uart16550	19.02	187.93	428.59	425.20	43381	95823
picorv32	83.86	225.47	1617.28	783.49	43747	96444
yacc	101.57	225.55	2232.63	898.45	42262	96144
aes_core	93.88	351.18	2478.49	1078.54	43955	95322
wb2axip	84.40	265.93	2323.97	822.28	45103	95840
wb_conmax	121.26	258.71	3427.69	1383.27	44795	95416
arm9	617.47	980.17	10229.67	6941.71	45002	96493
sha3	248.03	515.75	5218.05	3242.30	43580	95474
spimaster	325.07	594.11	7005.90	2084.52	36585	95186
ethernet	327.66	848.57	6408.45	2706.38	51182	94415
ecg	253.50	567.90	5083.02	1850.21	37951	96071
linkruncca	1219.77	3379.13	24823.51	10079.57	42836	95464
xge_mac	381.86	866.17	7669.93	1573.92	40991	96447
fft256	1551.27	2169.11	29751.55	10167.44	41060	96651
Ratio Avg.	100.00%	289.83%	2174.18%	988.62%	-	-

[1] Bootstrap synthesis. [2] Analysis Agent. [3] Optimization Agent.

TABLE V Bounded action space for CBTune reproduction.

ID	Command
1	`optimize -mode timing -map_effort high`
2	`tdopt -flat -perturb -stop {1} -area 1.0; phase_assign -mode area_recovery`
3	`optimize -mode timing -map_effort high -incremental`
4	`tdopt -flat -resyn2 -area 1.000000`
5	`retime_register -period 0 -effort medium; optimize -incremental`
6	`impl_select -mode down -scope s+ -tns; resize -mode normal -cost c>s>a -flat -area 1.0`
7	`pre_td; phase_assign -flat -mode timing -effort medium; tdopt -flat -stop {10 1.0} -area 1.000000`

power, 98.64% static power). These results show that SynAct achieves substantial improvements in both WNS and TNS while maintaining overall PPA balance and efficiency.

Runtime and LLM API token cost. TABLE IV summarizes the end-to-end execution time of all methods. All methods begin with the same bootstrap synthesis stage, whose runtime is reported in the "Boot." column and included in each method's total runtime. The percentages represent end-to-end runtime normalized to the bootstrap time. ChatLS has the lowest normalized runtime, averaging 289.83% of the bootstrap synthesis time, because it generates the script in a single pass. CBTune has the highest runtime at 2174.18% due to its

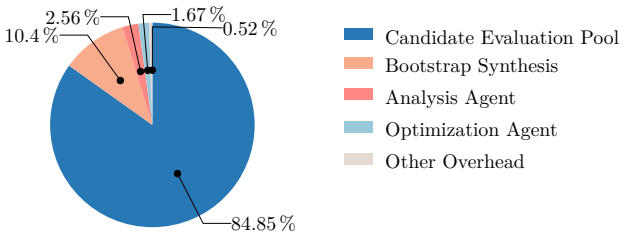

Fig. 6 Runtime breakdown of SynAct.

bandit-based exploration. SynAct strikes a favorable balance at 988.62%, less than half of CBTune's runtime, while achieving the best WNS ratio of 27.03%, compared with 71.73% for ChatLS and 66.67% for CBTune. The additional runtime relative to single-pass ChatLS comes from feedback-driven candidate evaluation, which enables iterative refinement.

Fig. 6 further breaks down the internal runtime composition of SynAct. Candidate evaluation dominates the runtime at 84.85%. Each iteration evaluates ten LLM-generated candidates in parallel and commits the best safe action. This setting balances server capacity and robustness to LLM stochasticity: fewer candidates reduce synthesis cost, whereas more broaden exploration. Despite parallelism, repeated synthesis remains the primary bottleneck. Bootstrap synthesis, the required initialization step, accounts for the second-largest share at 10.4%. The Analysis and Optimization Agents account for 2.56% and 1.67%, respectively, while GrammarVAE encoding, decoding, and result processing account for the remaining 0.52%. Future work may consider early stopping upon WNS saturation or lightweight candidate filtering before execution to reduce unnecessary synthesis runs.

Token usage in TABLE IV is consistent across all 14 benchmarks. On average, SynAct consumes about 139k tokens per design, with 69% from the Optimization Agent and 31% from the Analysis Agent. The Optimization Agent uses more tokens because it processes more documents, even after RAG filtering. Total usage varies slightly from 132k to 146k, showing low cross-design variance and a stable ratio between the two agents.

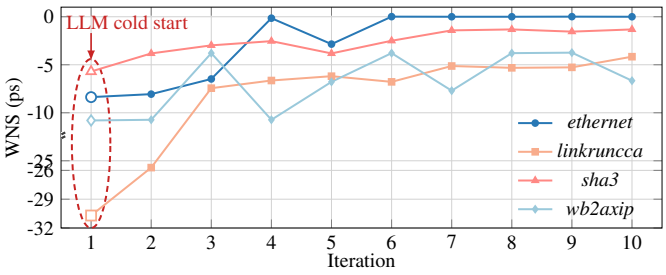

Fig. 7 Optimization trajectory of SynAct.

C. Optimization Trajectory Analysis

Recall that SynAct incrementally optimizes a design over at most n_{iter} LLM-guided iterations, where n_{iter} is the tunable maximum number of iterations. We use $n_{\text{iter}} = 5$ in the main experiments in TABLE III to balance optimization quality and runtime cost. Fig. 7 evaluates the sensitivity to n_{iter} by plotting optimization trajectories from $n_{\text{iter}} = 1$ onward. For readability, we show four representative benchmarks: *ethernet*, *linkruncca*, *sha3*, and *wb2axip*. Their clear WNS improvements and well-separated curves provide a concise view of SynAct's overall optimization trend without the visual clutter of plotting all benchmarks.

The first iteration, labeled "LLM cold start," refers to SynAct's initial attempt to generate optimization commands that rely almost solely on RAG-retrieved documents without historical experience, often leading to suboptimal results. As iterations proceed, SynAct refines its strategy using updated analyses and synthesis feedback, improving WNS over time. Most trajectories trend upward, while minor fluctuations in *linkruncca* and *sha3* reflect the exploratory nature of LLM-guided optimization. Although additional iterations can further improve WNS when runtime is less constrained, $n_{\text{iter}} = 5$ provides a practical balance between optimization quality and runtime cost, supporting our default setting. Overall, these results demonstrate the effectiveness of iterative adaptation driven by cumulative synthesis feedback.

D. Generalizability Across LLMs

To evaluate whether SynAct generalizes across different LLMs, we rerun the complete 14-design experiment using GPT-5.2 for both the Analysis and Optimization Agents under the same setup as TABLE III. As reported in TABLE VI, this configuration reduces the WNS and TNS Ratio Avg. to 20.37% and 13.69%, respectively, while keeping area and power within 1% of the bootstrap results. The default DeepSeek V3.1 configuration already achieves a WNS Ratio Avg. of 27.03% and substantially outperforms the baselines, confirming that SynAct's closed-loop diagnosis, retrieval, and experience refinement remain effective with different LLMs. GPT-5.2 further enhances report interpretation, diagnosis, and command generation within this pipeline, yielding additional timing gains. These results show that SynAct provides the adaptive optimization mechanism, while a stronger LLM can exploit it more effectively and raise the performance ceiling.

TABLE VI Five-run average PPA results of SynAct with GPT-5.2 under the same setup as TABLE III.

Benchmarks	**WNS**(ps)	TNS(ps)	Area (μm^2)	DP (mW)[1]	SP (μW)[2]
uart16550	-1.43	-1.53	401.39	1.584	0.395
picorv32	0.04	0.00	1131.68	3.542	1.303
yacc	-3.28	-59.29	1156.95	4.133	1.300
aes_core	-2.90	-158.80	1207.75	3.747	1.345
wb2axip	-6.46	-3352.07	1329.51	5.163	1.175
wb_conmax	-0.59	-9.60	1685.97	4.363	1.460
arm9	-4.42	-47.12	2034.10	1.817	2.077
sha3	-2.54	-246.83	2839.00	17.220	2.765
spimaster	0.02	0.00	5387.82	11.240	6.103
ethernet	-2.94	-17.15	6377.96	23.530	7.330
ecg	0.01	0.00	5846.38	22.690	5.722
linkruncca	0.00	0.00	9815.93	27.540	11.440
xge_mac	-4.36	-25.63	12683.96	7.540	14.020
fft256	-6.49	-151.60	26612.28	72.650	31.560
Ratio Avg.	20.37%	13.69%	99.36%	99.65%	99.41%

[1] Dynamic Power. [2] Static Power.

TABLE VII Diagnostics of GrammarVAE latent locality and BO-seed utility over 14 benchmarks.

Category	Diagnostic	Result
Latent locality	Mean Δr, near pairs	0.188
	Mean Δr, far pairs	0.238
BO seed utility	BO-seeded mean reward	0.297
	RAG-grounded mean reward	0.289
	Iterations where BO has highest mean	65.3%
	Iterations where best is BO-seeded	47.2%

E. Validation of Latent-Space BO Guidance

To validate BO-guided experience refinement, we test two properties required by its design. First, if GrammarVAE provides a useful local search prior, commands that are close in latent space should produce more similar synthesis rewards than distant commands. Using $r(C)$ from Equation (2), we measure this similarity by $\Delta r(C_i, C_j) = |r(C_i) - r(C_j)|$, where a smaller value indicates more consistent optimization outcomes. As shown in TABLE VII, near command pairs have a mean Δr of 0.188 versus 0.238 for far pairs, a 21.0% reduction. Second, the BO guidance should remain useful after the LLM converts a decoded seed into tool-valid candidates. BO-seeded candidates achieve a mean reward of 0.297 versus 0.289 for RAG-grounded candidates and obtain the highest mean reward in 65.3% of benchmark iterations. These results support the two design properties: locally consistent latent neighborhoods and BO guidance that remains effective after tool-aware LLM refinement.

F. Ablation Study on BO and Retrieval Modules

Fig. 8 presents the ablation results using the full SynAct configuration ("BO+GraphRAG") as the reference. In this configuration, candidate generation combines report diagnosis, documentation retrieved by GraphRAG, and GrammarVAE seeds guided by BO. All variants generate ten candidates per iteration, evaluate them through synthesis, and select the best safe command, thereby isolating the effects of BO and retrieval. The "w/o BO" variant removes BO guidance and the reuse of historical commands, while "vanilla RAG" uses

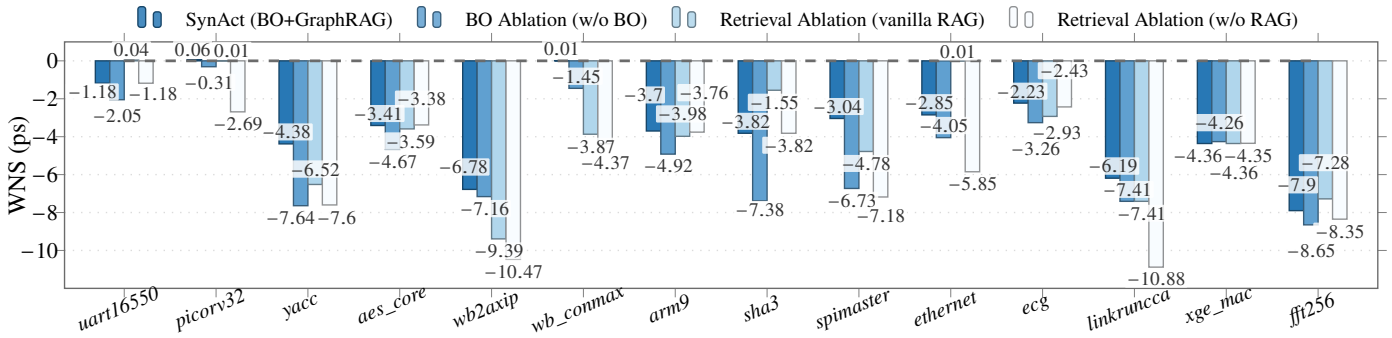

Fig. 8 Ablation of SynAct components on WNS. The full configuration (BO + GraphRAG) achieves the best timing improvement.

flat retrieval and "w/o RAG" provides the full documentation directly.

Without BO, timing performance clearly degrades, with the WNS Ratio Avg. rising from 27.0% to 37.5%, indicating less effective WNS improvement. The full configuration outperforms this variant on 13 of 14 benchmarks, with WNS improving from -7.16 ps to -6.78 ps on *wb2axip* and from -8.65 ps to -7.90 ps on *fft256*. These results show that BO-guided experience reuse steers exploration of the command space toward better optimization outcomes.

The "vanilla RAG" variant raises the WNS Ratio Avg. from 27.0% to 28.6%, while the "w/o RAG" variant further increases it to 38.3%. WNS degradations exceeding 2 ps on *yacc* and *linkruncca* suggest that GraphRAG's structured representation captures richer contextual dependencies than flat retrieval or direct use of the full documentation. Overall, SynAct achieves the most effective timing improvement among all variants, with BO supporting effective exploration and GraphRAG providing structured contextual knowledge for optimization.

VII. Conclusion

In summary, SynAct is an adaptive closed-loop LLM framework for iterative PPA optimization in commercial synthesis, addressing the limitations of fixed-action search and one-shot script generation. By combining synthesis feedback, prior experience, and retrieved tool knowledge, SynAct enables state-aware decisions with explicit rationales while improving timing and maintaining balanced area and power. Although evaluated only on AltiSyn®and open-source designs, SynAct is designed for portability to other logic synthesis tools with scripting interfaces, command documentation, and structured PPA and timing reports by replacing its tool-specific adapters and knowledge base. Future work will validate cross-tool portability and industrial-scale performance and reduce candidate-evaluation overhead.

Acknowledgment

The authors thank ZeniSyn Design Systems for providing access to AltiSyn® and related technical support. Regarding the use of generative AI, DeepSeek V3.1 and GPT-5.2 were used in SynAct to analyze synthesis reports and generate candidate commands, while OpenAI Codex was used solely for language editing and grammar checking. All technical content, ideas, and results are original work by the authors.

References

[1] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[2] A. B. Kahng, "New directions for learning-based IC design tools and methodologies," in *2018 23rd Asia and South pacific design automation conference (ASP-DAC)*. IEEE, 2018, pp. 405–410.

[3] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *International Conference on Computer-Aided Verification (CAV)*. Springer, 2010, pp. 24–40.

[4] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "DRiLLS: Deep reinforcement learning for logic synthesis," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*. IEEE, 2020, pp. 581–586.

[5] A. Grosnit, C. Malherbe, R. Tutunov, X. Wan, J. Wang, and H. B. Ammar, "BOiLS: Bayesian optimisation for logic synthesis," in *IEEE/ACM Design, Automation and Test in Europe (DATE)*. IEEE, 2022, pp. 1193–1196.

[6] Z. Pei, F. Liu, Z. He, G. Chen, H. Zheng, K. Zhu, and B. Yu, "AlphaSyn: Logic synthesis optimization with efficient Monte Carlo Tree Search," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[7] C. Yu, "FlowTune: Practical multi-armed bandits in boolean optimization," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

[8] F. Liu, Z. Pei, Z. Yu, H. Zheng, Z. He, T. Chen, and B. Yu, "CBTune: Contextual bandit tuning for logic synthesis," in *IEEE/ACM Design, Automation and Test in Europe (DATE)*. IEEE, 2024, pp. 1–6.

[9] H. Wu, Z. He, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "ChatEDA: A large language model powered autonomous agent for EDA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 43, no. 10, pp. 3184–3197, 2024.

[10] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu *et al.*, "ChipNemo: Domain-adapted LLMs for chip design," 2023. [Online]. Available: https://arxiv.org/abs/2311.00176

[11] H. Zheng, H. Wu, and Z. He, "ChatLS: Multimodal retrieval-augmented generation and chain-of-thought for logic synthesis script customization," in *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2025, pp. 1–7.

[12] Y. Pu, Z. He, T. Qiu, H. Wu, and B. Yu, "Customized retrieval augmented generation and benchmarking for EDA tool documentation QA," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2024, pp. 1–9.

[13] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations (ICLR)*, 2022.

[14] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu *et al.*, "AutoGen: Enabling next-gen LLM applications via multi-agent conversations," in *First Conference on Language Modeling (CoLM)*, 2024.
[15] Synopsys, *Design Compiler User Guide*, Mountain View, CA, USA, 2022, available via Synopsys SolvNet.
[16] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive NLP tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.
[17] S. Huang, J. Li, Z. Yu, J. Ye, J. Xu, N. Xu, and G. Dai, "LLSM: LLM-enhanced logic synthesis model with EDA-guided CoT prompting, hybrid embedding and AIG-tailored acceleration," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*. IEEE, 2025, pp. 974–980.
[18] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, D. Metropolitansky, R. O. Ness, and J. Larson, "From local to global: A graph RAG approach to query-focused summarization," *arXiv preprint arXiv:2404.16130*, 2024.
[19] J. Wu, J. Zhu, Y. Qi, J. Chen, M. Xu, F. Menolascina, Y. Jin, and V. Grau, "Medical graph RAG: Evidence-based medical large language model via graph retrieval-augmented generation," in *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025, pp. 28 443–28 467.
[20] M. J. Kusner, B. Paige, and J. M. Hernández-Lobato, "Grammar variational autoencoder," in *International Conference on Machine Learning (ICML)*. PMLR, 2017, pp. 1945–1954.
[21] D. Lynch, J. McDermott, and M. O'Neill, "Program synthesis in a continuous space using grammars and variational autoencoders," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2020, pp. 33–47.
[22] H. Dai, Y. Tian, B. Dai, S. Skiena, and L. Song, "Syntax-directed variational autoencoder for molecule generation," in *International Conference on Learning Representations (ICLR)*, 2018.
[23] H.-M. Gutmann, "A radial basis function method for global optimization," *Journal of global optimization*, vol. 19, no. 3, pp. 201–227, 2001.
[24] N. Srinivas, A. Krause, S. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design," in *International Conference on Machine Learning (ICML)*. PMLR, 2010, pp. 1015–1022.
[25] "ZeniSyn Design Systems," https://www.zenisyn.com.
[26] 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 (MEJ)*, vol. 53, pp. 105–115, 2016.
[27] "OpenCores," https://opencores.org.