ChatFCM: Comprehension-Synthesis Decoupling for LLM-Based Functional Coverage Model Generation

Yuan Pu[1], Yudong He[2], Zhenghao Chen[3], Mingjun Wang[1], Bingkun Yao[1], Zhaotan Lin[1], Qin Chen[4], Bing Li[4], Song Chen[2], Zhuolun He[1*], Wenjian Yu[5], Yanfeng Li[4*], Bei Yu[1*]
[1]Chinese University of Hong Kong [2]University of Science and Technology of China
[3]Beijing University of Posts and Telecommunications [4]ChangXin Memory Technologies
[5]Tsinghua University

Abstract

Deriving functional coverage models from protocol specifications remains a largely manual bottleneck in IC verification. While large language models show promise in automating this task, end-to-end synthesis conflates comprehension with code generation, making it prone to error and omission. We propose ChatFCM, an LLM-based framework that decouples coverage model generation into a comprehension phase and a synthesis phase, bridged by a structured intermediate representation (IR). The comprehension phase extracts specification knowledge into the IR as atomic entities (protocol variables), derived entities (intermediate logic), and relations (verification constraints). Dependencies among these entities form a derivation DAG; a DAG-based validation procedure detects and repairs extraction errors, ensuring that all intermediate dependencies are correctly resolved before code generation begins. The synthesis phase traverses the validated DAG in topological order to generate sampling logic, and applies subtype-specific templates to produce coverage constructs. On benchmarks of 35 and 52 constraints from the DDR5 and LPDDR5 JEDEC specifications, ChatFCM achieves 88.6% and 82.7% functional correctness, compared with 42.9% and 53.8% for the strongest end-to-end baseline.

1 Introduction

Verification accounts for roughly 70% of the modern IC design lifecycle [1], with functional verification representing a dominant share of this effort. Coverage-driven verification (CDV) has become the industry-standard methodology for managing it: engineers derive functional coverage models from protocol specifications to define measurable goals that guide both stimulus generation and sign-off decisions. A complete model comprises coverage constructs that encode the verification targets and sampling logic that faithfully extracts the relevant signals. Constructing such models, however, remains largely manual and error-prone, creating a significant bottleneck in the verification cycle.

To illustrate, consider generating a coverage model for WCK2CK leveling entry verification from a DRAM protocol specification. As shown in Figure 1, the information required for this single model

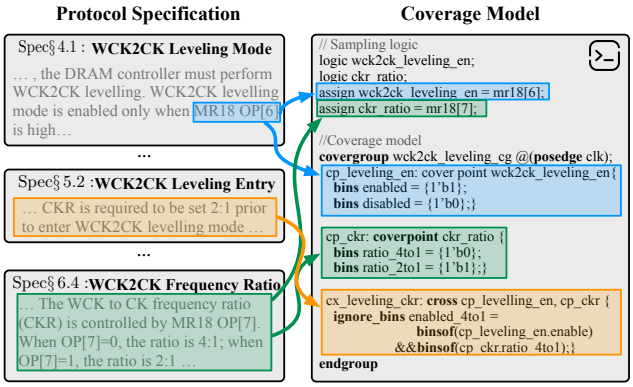


Figure 1: An example of generating a functional coverage model with sampling logic from scattered sections of a DRAM specification.

is scattered across three sections: Spec§ 4.1 identifies the enable signal for leveling mode, Spec§ 6.4 defines the encoding of the clock frequency ratio (CKR), and Spec§ 5.2 states that CKR must be set to a specific value before leveling entry. A correct coverage model must synthesize all three fragments—deriving sampling logic from Spec§ 4.1 and 6.3, and expressing the entry constraint from Spec§ 5.2 as a cross coverage construct. This pattern is representative: even a single coverage model typically depends on information distributed across multiple, often distant, specification sections.

Recent advances in large language models have driven their adoption across numerous EDA tasks [2–5], from hardware design and debugging [6–15] to EDA tool scripting [16–20], documentation question answering [21–26], manufacturing optimization [27, 28], and analog and PCB design [29–34]. Within verification specifically, LLMs have been applied to testbench generation [35–40], test pattern generation [41–43], and assertion generation [44–46], yet functional coverage model generation remains largely unexplored. A natural strategy is to apply an LLM as an end-to-end pipeline that ingests specification text and directly emits SystemVerilog coverage code. This formulation, however, suffers from two fundamental limitations. First, information scatter leads to incompleteness. The specification fragments required for a single coverage model are often distributed across distant sections. Both retrieval-augmented and long-context approaches frequently fail to aggregate all relevant fragments, yielding models with absent sampling logic or hallucinated signal definitions. Second, collapsing specification comprehension and code synthesis into a single inference step overloads the model: it must simultaneously locate relevant definitions, resolve cross-section dependencies, interpret constraint semantics, and produce syntactically correct SystemVerilog. This makes end-to-end generation inherently

*Corresponding authors.
This work is supported by The Research Grants Council of Hong Kong SAR (No. CUHK14211824, CUHK14201624, and T46-415/25-R).

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICCAD '26, San Jose, CA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2873-0/2026/11
https://doi.org/10.1145/3831252.3834070

error-prone, and when errors do occur, they propagate silently with no opportunity for intermediate verification or targeted correction.

We observe that both limitations share a common root: the absence of a structured boundary between understanding the specification and translating that understanding into code. Without such a boundary, comprehension errors propagate silently into the generated code, destroying traceability, while cross-section dependencies must be resolved implicitly in a single pass, often yielding incomplete logic.

This insight motivates ChatFCM, a framework that decouples coverage model generation into a comprehension phase and a synthesis phase, connected through a structured intermediate representation (IR). The IR organizes specification knowledge into three layers: atomic entities capture protocol variables directly grounded in specification clauses, and relations encode the verification constraints that ultimately map to coverage items. Between these two layers lies a significant gap: the operands a constraint requires are often not raw protocol variables but derived quantities—state flags, event counters, or timing measurements—each of which may depend on other derived quantities or atomic entities through non-trivial computation. Derived entities fill this gap by explicitly representing these intermediate computations. Within each relation, its atomic and derived entities together form a derivation DAG, turning the implicit multi-step reasoning that end-to-end approaches must guess at into a structured, inspectable graph. This DAG serves two purposes. In the comprehension phase, it enables a validation-and-repair loop that catches extraction errors before any code is generated, giving engineers an auditable checkpoint with full traceability to the source specification. In the synthesis phase, topological traversal of the DAG generates sampling logic for each entity, ensuring every dependency is resolved before the logic that consumes it is produced. With sampling logic in place, subtype-specific templates then guide the LLM to assemble the final coverage group from each relation.

The main contributions of this paper are summarized as follows:

- We propose ChatFCM, the first LLM-based framework that generates functional coverage models with sampling logic from protocol specifications. ChatFCM decouples the task into a comprehension phase that extracts a structured IR, and a synthesis phase that leverages the verified IR to guide the generation of SystemVerilog coverage models.
- We design a three-layer IR comprising atomic entities, derived entities, and relations. The entities involved in each relation form a derivation DAG that decomposes complex constraints into explicit inter-entity dependencies, enabling automated validation and dependency-aware code synthesis.
- We construct benchmarks for functional coverage model generation from the DDR5 and LPDDR5 JEDEC specifications, and propose progressive evaluation metrics of checkpoint coverage, compilability, and functional correctness.
- We conduct extensive experiments showing that ChatFCM achieves 88.6% and 82.7% functionality on DDR5 and LPDDR5 respectively, significantly outperforming the strongest end-to-end baselines.

2 Preliminaries

2.1 Functional Coverage Model in SystemVerilog

A functional coverage model encodes the verification intent for a given protocol specification. In SystemVerilog, this model is realized through the *covergroup* construct, which contains *coverpoints* and *crosses*. Each *coverpoint* binds a sampled variable to a set of bins that partition its legal value space; bins can be explicit, automatic, wildcard, illegal, or ignored. A cross defines the Cartesian product of multiple coverpoints to capture variable interactions. Covergroups are evaluated at designated sampling events, either a clock edge or an explicit `.sample()` call.

The sampled variables in a covergroup rarely correspond directly to raw design signals. They are typically obtained through sampling logic, a signal-level abstraction layer that transforms low-level pins and registers into protocol-meaningful observables through operations such as decoding, field extraction, and conditional composition.

2.2 Problem Definition

Problem 1 (Functional Coverage Model Generation)**.** Given a hardware protocol specification document $\mathcal{D}$ and a base sampling infrastructure $\mathcal{S}$ that provides directly observable protocol signals obtained through bus decoding, the goal is to automatically generate a SystemVerilog coverage model $\mathcal{M}$ consisting of sampling logic and covergroup constructs. Let $\mathcal{C} = \{c_1, \ldots, c_k\}$ denote the set of coverage checkpoints implicitly defined by $\mathcal{D}$, where each c_j represents an individual verifiable scenario. The generated model $\mathcal{M}$ should satisfy two objectives: (1) *completeness*, requiring that every checkpoint in $\mathcal{C}$ is addressed by some construct in $\mathcal{M}$, i.e., $\forall\, c_j \in \mathcal{C},\ \exists\, m_i \in \mathcal{M}$ such that m_i covers c_j; and (2) *correctness*, requiring that each construct $m_i \in \mathcal{M}$ is syntactically valid SystemVerilog and semantically faithful to the specification $\mathcal{D}$.

3 Three-Layer Intermediate Representation (IR)

Central to ChatFCM is an Intermediate Representation (IR) that decomposes the spec-to-coverage-model pipeline into two distinct stages, document comprehension and code generation, and provides the formal contract between them.

The design of the IR is motivated by three considerations. First, the information required to construct a single coverage model is typically scattered across multiple sections of a specification document. The IR consolidates these dispersed fragments into a unified, self-contained representation, ensuring that all relevant semantics are captured in one place before any code is produced. Second, by factoring the overall problem into two well-scoped subtasks, the code generation stage receives inputs that are already formalized, comprehensive, and free of redundancy, making it substantially easier to design LLM-based algorithms that perform structured reasoning and produce correct code. Third, the IR serves as a structured validation checkpoint: extraction errors can be detected and repaired at this stage rather than propagating silently into downstream code generation, where they would be far more costly to diagnose and fix.

We organize the IR into three hierarchical layers: 1) the Entity IR, 2) the Derived Entity IR, and 3) the Relation IR. We describe each layer in turn.

3.1 Entity IR

The Entity IR captures the semantics of individual protocol-level entities. In a typical protocol specification, information about a single entity is often spread across multiple locations: its bit-width may appear in a signal summary table, while its encoding in a dedicated subsection. The Entity IR resolves these cross-references and distills all per-entity information into a structured, self-contained record.

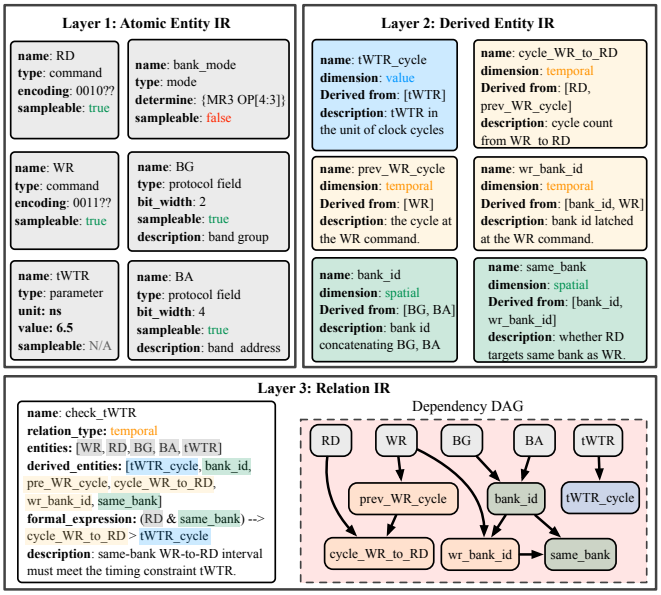


Figure 2: Illustration of the three-layer IR.

Based on our analysis of hardware protocol specifications, we categorize entities into six types: *command*, *mode*, *protocol field*, *signal*, *configuration register*, and *parameter*, each reflecting a distinct role in the protocol semantics. All entity records share common fields including name and description. Beyond these, different types carry type-specific fields. As illustrated in Figure 2, a *command entity* such as `WR` specifies the bit-level encoding that identifies it on the bus, while a *protocol field* entity such as `BG` records its bit width.

A particularly important common field is *sampleable*, which characterizes how the entity's runtime value is obtained. Commands, protocol fields, and configuration registers are directly sampleable (`true`), as their values can be directly translated from the interface bus decoding. Entities such as modes and states are not directly observable (*sampleable* = `False`) and must be inferred from sampleable entities ; for instance, the mode entity `CKR` in Figure 1 cannot be read from the bus directly but is determined by the value of configuration register `MR18` bit `OP[8]`. Finally, certain parameters represent fixed specification constants that have no associated runtime signal (`N/A`); `tWTR` in Figure 2 is one such example, whose value is a constant defined by the specification. This distinction is critical for code generation, as it determines whether a simple signal translation suffices, whether additional inference logic must be synthesized, or whether a compile-time constant can be used directly.

3.2 Derived Entity IR

The constraints in a specification seldom operate on atomic entities directly but instead depend on intermediate quantities that must be derived from them, such as unit-converted timing thresholds, latched signal states, or composite identifiers assembled from multiple protocol fields. The Derived Entity IR bridges this gap by explicitly constructing these intermediate quantities, providing the Relation IR with well-defined operands. We observe that the derivations required by protocol constraints fall into three orthogonal dimensions. The *value* dimension applies stateless domain transformations, such as converting units or remapping encodings. The *spatial* dimension composes or compares atomic entities to establish relationships among addressable targets. The *temporal* dimension tracks state across clock cycles, such as latching a value at a past event or counting inter-event intervals. Because these dimensions are orthogonal, they can be freely stacked: a spatially composed identifier can be temporally latched, and the result can be spatially compared again.

Each derived entity record specifies a *derived_from* list, a *dimension* tag, and a natural-language description. As shown in Figure 2, `tWTR_cycle` is a value-dimension entity that converts nanoseconds to clock cycles; `bank_id` is a spatial-dimension entity that concatenates `BG` and `BA` into a composite identifier; and `cycle_WR_to_RD` is a temporal-dimension entity that counts elapsed cycles between two commands.

Since *derived_from* may reference other derived entities, derivations naturally compose into a directed acyclic graph (DAG). The dependency DAG in Figure 2 illustrates this: `bank_id` is temporally latched to produce `wr_bank_id`, which is then spatially compared with the current `bank_id` to yield `same_bank`.

3.3 Relation IR

The Relation IR assembles entities and derived entities into complete, LLM-interpretable constraints. Each relation record contains a *name*, a *relation_type*, lists of referenced *entities* and *derived_entities*, a *formal_expression*, and a natural-language *description*. The *formal_expression* is a logical expression composed entirely of operands already defined in the preceding two IR layers, ensuring that every symbol is traceable to its origin. We categorize relations into four types: *temporal* relations capture ordering or timing dependencies between events; *implication* relations express conditional dependencies where one condition mandates another; *enumeration* relations enumerate all valid values of an entity or valid combinations among entities.; and *exclusion* relations specify illegal or mutually exclusive combinations among entities. These four types partition the constraint space along two axes: a constraint either *permits* behavior (temporal, implication, enumeration) or *prohibits* it (exclusive), and either reasons *across* time (temporal) or *within* a single state (implication, enumeration, exclusive). Every constraint encountered in our evaluation maps to exactly one type without residual.

As shown in Figure 2, the relation `check_tWTR` is a temporal relation referencing five base entities and six derived entities. Its formal expression, `(cmd=RD & same_bank)` $\rightarrow$ `cycle_WR_to_RD >= tWTR_cycle`, reads as: when a `RD` command targets the same bank as a preceding `WR`, the elapsed interval must meet the converted timing threshold. The guard combines an event filter (`cmd=RD`) with a spatial condition (`same_bank`), while the consequent compares a temporal quantity (`cycle_WR_to_RD`) against a value-domain quantity (`tWTR_cycle`). All three derived-entity dimensions thus converge in a single, compact expression.

4 ChatFCM

4.1 Overall Flow

Figure 3 presents the overall flow of ChatFCM, a two-phase pipeline that ingests specification chunks and produces synthesizable SystemVerilog sampling logic together with functional covergroup construct. In Phase 1 (Document Comprehension), *Atomic Entity IR Extraction* (Section 4.2.1) scans each spec chunk with an LLM to identify coverage-relevant entities, and cross-references them against a

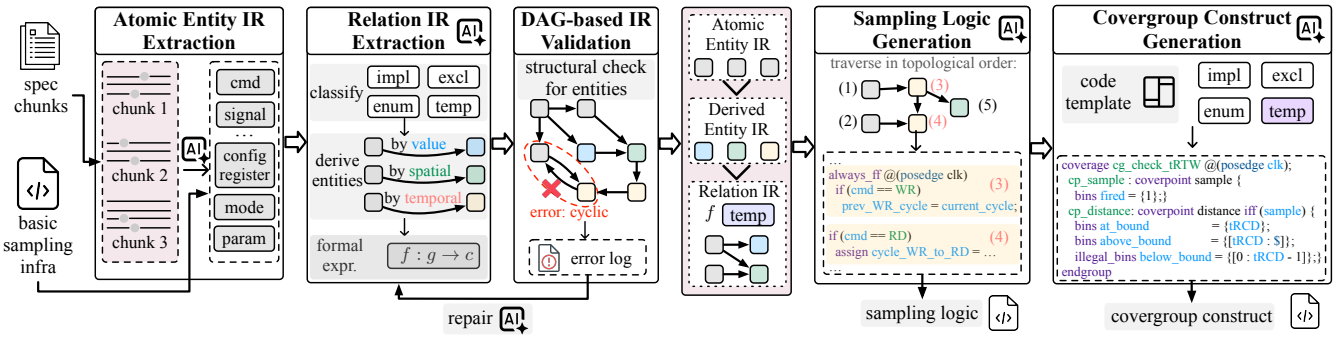


Figure 3: Overall Flow of ChatFCM.

basic sampling infrastructure to determine their sampleability. *Relation IR Extraction* (Section 4.2.2) then pairs each chunk with its associated entities to extract inter-entity relations classified into four subtypes (*implication*, *exclusion*, *enumeration*, and *temporal*); when atomic entities alone are insufficient to characterize a relation, the framework derives intermediate entities along the value, spatial, or temporal dimensions, and constructs a formal expression over both atomic and derived entities. *DAG-based IR Validation* (Section 4.2.3) performs structural checks on the derived-entity graph, feeding detected errors back to the LLM for repair. In Phase2 (Code Synthesis), *Sampling Logic Generation* (Section 4.3.1) traverses the validated derivation DAG in topological order, synthesizing SystemVerilog logic for each non-sampleable entity by resolving its dependencies against infrastructure-provided signals and previously generated signals. *Covergroup Construct Generation* (Section 4.3.2) then instantiates type-specific code templates with the formal expressions from the Relation IR to produce the target functional coverage models.

4.2 Document Comprehension: IR Construction

This sub-section elaborates the document comprehension phase of ChatFCM. Rather than generating coverage code directly from specification text, ChatFCM first distills the specification into a structured IR that captures exactly the entities, derived entities, and inter-entity relations needed for coverage modeling. Algorithm 1 gives the complete algorithmic procedure.

4.2.1 Atomic Entity IR Extraction. Section 3.1 defines a structured atomic entity record with fields such as *name*, *type*, *values*, and *sampleable*. The goal of entity extraction (Lines 2–6 of Algorithm 1) is to populate these records automatically from specification text. The central difficulty is that an entity's attributes are seldom confined to one paragraph but instead scattered across tables, descriptive paragraphs, and footnotes throughout the specification. Extraction therefore proceeds in two stages: per-paragraph extraction followed by cross-paragraph merging.

For each paragraph p_i, the `ExtractEntities` call (Line 3) invokes an LLM to produce zero or more partial entity records in structured JSON aligned with the schema of the corresponding entity type. The *type* field is constrained to the six categories defined in the IR. The *sampleable* field is determined by cross-referencing the entity against the basic sampling infrastructure $\mathcal{S}$: an entity is marked sampleable if $\mathcal{S}$ already defines a signal from which its value can be directly observed, and non-sampleable otherwise. Each record

Algorithm 1 Document Comprehension: IR Construction

Input: Specification paragraphs $\mathcal{P} = \{p_1, \ldots, p_n\}$, basic sampling infrastructure $\mathcal{S}$.
Output: IR: atomic entities $\mathcal{E}$, derived entities $\mathcal{E}^d$, relations $\mathcal{R}$

1: $\mathcal{E} \leftarrow \emptyset$; $\mathcal{E}^d \leftarrow \emptyset$; $\mathcal{R} \leftarrow \emptyset$
// Pass 1: Entity Extraction (Section 4.2.1)
2: **for** each paragraph $p_i \in \mathcal{P}$ **do**
3: $\quad \mathcal{E}_i \leftarrow \texttt{ExtractEntities}^\star(p_i, \mathcal{S})$
4: $\quad \mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{E}_i$
5: **end for**
6: $\mathcal{E} \leftarrow \texttt{MergeEntities}(\mathcal{E})$ ▷ cross-paragraph merging
// Pass 2: Relation Extraction & Entity Derivation (Section 4.2.2)
7: **for** each paragraph $p_i \in \mathcal{P}$ **do**
8: $\quad \mathcal{E}_i \leftarrow \{e \in \mathcal{E} \mid p_i \in e.src\}$
9: $\quad \mathcal{R}_i \leftarrow \texttt{ClassifyRelations}^\star(p_i, \mathcal{E}_i)$
10: $\quad$ **for** each relation $r \in \mathcal{R}_i$ **do**
11: $\quad\quad \mathcal{E}^d_r \leftarrow \emptyset$
12: $\quad\quad$ **if** atomic entities insufficient for r **then**
13: $\quad\quad\quad \mathcal{E}^d_r \leftarrow \texttt{DeriveEntities}^\star(r, \mathcal{E}_i)$ ▷ value / spatial / temporal
14: $\quad\quad\quad \mathcal{E}^d \leftarrow \mathcal{E}^d \cup \mathcal{E}^d_r$
15: $\quad\quad$ **end if**
16: $\quad\quad r.expr \leftarrow \texttt{BuildFormalExpr}^\star(r, \mathcal{E}_i, \mathcal{E}^d_r)$
17: $\quad$ **end for**
18: $\quad \mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{R}_i$
19: **end for**
// DAG-based IR Validation (Section 4.2.3)
20: $G \leftarrow \texttt{BuildDerivationDAG}(\mathcal{E}, \mathcal{E}^d)$
21: **repeat**
22: $\quad \mathcal{F} \leftarrow \texttt{StructuralCheck}(G)$
23: $\quad (G, \mathcal{E}^d) \leftarrow \texttt{RePrompt}^\star(G, \mathcal{E}^d, \mathcal{F})$
24: **until** $\mathcal{F} = \emptyset$
25: **return** $(\mathcal{E}, \mathcal{E}^d, \mathcal{R})$

* LLM-driven step.

also includes an *alias* set capturing alternative names or abbreviations that co-occur in the same paragraph (e.g., *clock ratio mode* and *CKR*). Because a paragraph typically covers only a subset of attributes, each record is intentionally partial; the consolidation step (`MergeEntities`, Line 6) later fuses co-referent records into complete entities. Let $\mathrm{Id}(r) = r.\texttt{name} \cup r.\texttt{alias}$. Two records are placed in the same cluster when their identifier sets overlap:

$$r_a \sim r_b \iff \mathrm{Id}(r_a) \cap \mathrm{Id}(r_b) \neq \emptyset. \tag{1}$$

Taking the transitive closure of $\sim$ yields a partition over all partial records. Within each equivalence class, disjoint fields are filled by

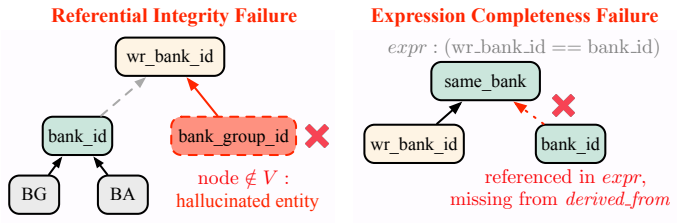


Figure 4: Two examples of relation IR failures detected by DAG-based IR validation.

complement; when multiple records supply the same field, an LLM call reconciles the conflicting values.

4.2.2 Entity Derivation and Relation IR Extraction. The atomic entities extracted in the previous step represent the elementary concepts that the specification defines individually. However, an executable coverage constraint typically operates on quantities that no single atomic entity can capture. Consider the `check_tWTR` relation IR example in Figure 2: the specification states a minimum write-to-read delay in nanoseconds, yet a cycle-accurate coverage monitor must (i) convert the parameter to clock cycles, (ii) compose a bank identifier from separate address fields, and (iii) track the cycle distance between a past write and the current read command. None of these operands are atomic entities; they must be **derived**. Section 3.2 organizes entity derivation into three orthogonal dimensions: *value*, *spatial*, and *temporal*. A single constraint may require derivations along all three axes, as the above example illustrates.

Algorithm 1 (lines 7-19) details the procedure. For each paragraph p_i, the LLM, via `ClassifyRelations` (line 9), identifies relations $\mathcal{R}_i$ among the atomic entities $\mathcal{E}_i$ and determines, for each relation r, whether the atomic entities alone suffice to express r as a formal constraint. When a gap is detected, `DeriveEntities` (line 13) is invoked: the LLM receives r, $\mathcal{E}_i$, and the derivation taxonomy (introduced in Section 3.2) as a structured prompt. Guided by the three derivation dimensions, it reasons about *what* quantity is missing (e.g., a composite bank identifier to scope the constraint), *which* atomic entities serve as sources (e.g., the `BG` and `BA` address fields), and *how* the derivation is computed (e.g., concatenation {`BG`, `BA`}). Each derived entity is annotated with its type, sources, and computation rule. Providing the taxonomy as an explicit schema is critical: it constrains the LLM's output space to well-defined categories, preventing ad-hoc or redundant intermediate quantities.

With all entities in place, `BuildFormalExpr` (line 16) constructs a structured predicate for each relation (e.g., `if same_bank then cycle_WR_to_RD >= tWTR_cycle`), which maps to a coverage model expression. Each derived entity records an explicit dependency on its sources, and derived entities may themselves be sources for further derivations.

4.2.3 DAG-Based IR Validation. The LLM-driven extraction steps in Algorithm 1 (marked with ⋆) may introduce semantic errors such as fabricated entities, omitted dependencies, or incorrectly scoped constraints. Systematic post-hoc validation is therefore essential. We observe that two structural invariants hold across all well-formed IRs: (i) every derived entity $e^d \in \mathcal{E}^d$ is defined strictly over previously established entities, and (ii) every unsampleable atomic entity $e^u \in \mathcal{E}^u$ is functionally determined by a subset of sampleable entities $\mathcal{E}^s$. Together, these invariants guarantee that the dependency structure is acyclic, enabling us to organise all entities into a *derivation DAG* $G = (V, E)$ that serves as a checkable structural scaffold:

$$V = \mathcal{E}^s \cup \mathcal{E}^u \cup \mathcal{E}^d, \tag{2}$$

where $\mathcal{E}^s$ and $\mathcal{E}^u$ denote the sampleable and unsampleable entities, respectively. The edge set combines two kinds of dependency:

$$E_{\text{deriv}} = \{(u, v) \mid v \in \mathcal{E}^d,\ u \in v.\textit{derived_from}\}, \tag{3}$$

$$E_{\text{det}} = \{(u, v) \mid v \in \mathcal{E}^u,\ u \in v.\textit{determined_by}\}, \tag{4}$$

$$E = E_{\text{deriv}} \cup E_{\text{det}}. \tag{5}$$

This induces a topological stratification of V: sampleable entities $\mathcal{E}^s$ form the source nodes, unsampleable entities $\mathcal{E}^u$ depend on them via E_{det}, and derived entities $\mathcal{E}^d$ are progressively composed at increasing depth via E_{deriv}. Each relation $r \in \mathcal{R}$ consumes a subset of V as operands. One example of a concrete DAG for the `check_tWTR` relation IR can be found in Figure 2.

Algorithm 1 (lines 20–24) describes the validation loop. The DAG G returned by `BuildDerivationDAG` (line 20) is verified by a single, fully deterministic pass `StructuralCheck` (line 25) that requires no LLM invocation. The check first confirms that G is acyclic and that every edge endpoint exists in V (referential integrity), ruling out fabricated or misspelled entity names. It then verifies type consistency, ensuring that derivation edges respect the category hierarchy ($\mathcal{E}^s \to \mathcal{E}^d$ or $\mathcal{E}^d \to \mathcal{E}^d$), and that every symbol appearing in a relation constraint r.`expr` corresponds to a node in V. For unsampleable entities, the check enforces sampling-chain completeness: every $e^u \in \mathcal{E}^u$ must possess at least one E_{det} edge from a sampleable source. Finally, an expression-level completeness rule requires that, for every derived entity e^d, the set $e^d.\textit{derived_from}$ covers all entity symbols referenced in its definition expression. This last rule targets a subtle class of error that is structurally legal yet semantically incorrect: a derivation whose dependency list is a strict subset of its true operands.

Figure 4 illustrates two failure examples on the `check_tWTR` example from Figure 2. Fault F_1 (referential integrity): the extracted `wr_bank_id` lists a dependency on `bank_group_id`, yet no such node exists in V because it is a hallucinated entity name; the correct dependency should be `bank_id`. Fault F_2 (expression completeness): `same_bank` lists only `wr_bank_id` in its `derived_from` set, while its definition expression `(wr_bank_id == bank_id)` also references `bank_id`, leaving a true operand unrecorded. Both faults can be caught by the DAG-based IR validation procedure.

When the fault set $\mathcal{F} \neq \emptyset$, `RePrompt` (line 23) feeds each fault description back to the LLM for targeted repair. The corrected entities update G, after which the validation loop repeats. The loop terminates when $\mathcal{F} = \emptyset$, i.e. no faults remain. Because every check is deterministic, the validation outcome is fully reproducible and independent of LLM stochasticity.

4.3 IR-to-Code Synthesis

After obtaining the relation IR from Phase 1, Phase 2 translates it into an executable SystemVerilog coverage model comprising two components: *sampling logic* and *covergroup constructs*. The separation is necessary because a covergroup can only reference signals that already exist in the testbench, yet the verification intent in the specification frequently involves temporal measurements (e.g. counting cycles between two commands), state retention (e.g. latching a bank

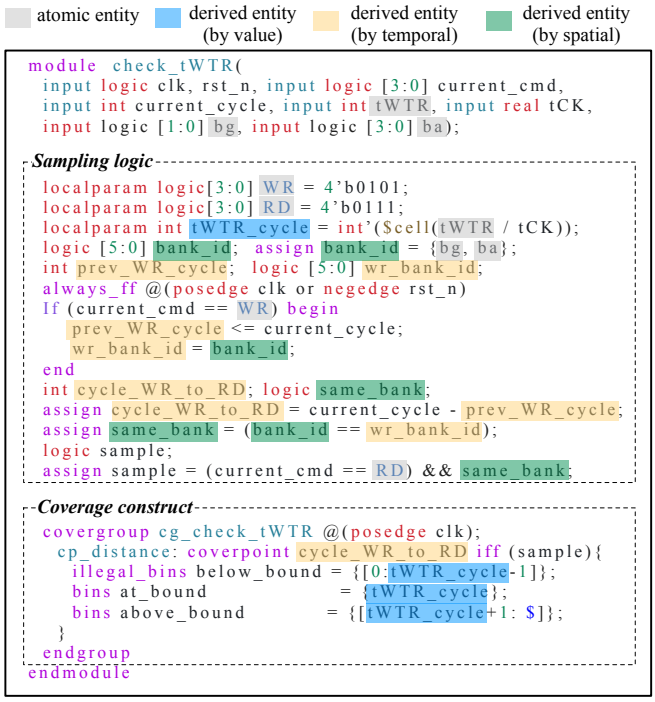


Figure 5: Sampling logic and covergroup construct generation example.

address at write time), or cross-event comparisons with no direct counterpart on the DUT interface. Sampling logic bridges this gap by materializing such abstract concepts into concrete, cycle-accurate signals. Accordingly, Phase 2 proceeds in two stages: the first performs **sampling logic generation**, invoking the LLM once per entity in topological order of G to build a signal-monitoring module; the second performs **covergroup construct generation**, invoking the LLM once per relation to synthesise a covergroup encoding the corresponding verification intent. In both stages the IR serves as structured context that constrains the LLM, ensuring dependency correctness, entity completeness, and semantic fidelity to the source specification.

4.3.1 Sampling Logic Generation. Given a relation IR with derivation DAG $G = (V, E)$, a topological sort yields an entity sequence

$$\sigma = (e_1,\ e_2,\ \ldots,\ e_n) \tag{6}$$

satisfying $i < j$ for every edge $(e_i, e_j) \in E$. The sampling module is built by processing σ left to right, issuing one LLM call per entity. For entity e_i (either atomic or derived), the prompt context is

$$\mathcal{C}_i \;=\; \left\langle\, \mathcal{S}_\text{sys},\ \ \mathcal{P},\ \ d(e_i),\ \ \left\{\, d(e_j) \mid (e_j, e_i) \in E \,\right\} \,\right\rangle, \tag{7}$$

where $\mathcal{S}_\text{sys}$ is a system prompt encoding SystemVerilog conventions instructions, $\mathcal{P}$ is the module interface skeleton (port declarations for clock, reset, command bus, and other design-level signals), $d(e_i)$ is the IR definition record of the target entity (name, category, and relevant attributes), and the last component supplies the definition records of all direct predecessors of e_i in G. For each entity e_i, the LLM generates a single SystemVerilog construct that produces a sampleable signal. The construct type is determined by the derivation dimension: `assign` for value, spatial or some temporal derivations that can be expressed combinationally, and `always_ff` for temporal derivations that require sequential state. Fixed protocol constants from atomic entities are emitted as `localparam` declarations.

Because the DAG enforces strict dependency ordering, each entity is defined exclusively over its direct predecessors. Supplying only their IR definitions therefore provides sufficient context; no accumulated code from earlier steps needs to appear in the prompt. The topological ordering further guarantees that every signal referenced by the construct for e_i has already been declared by some earlier construct for e_j ($j < i$). This per-entity decomposition narrows each LLM call to a single, well-defined translation task, empirically reducing hallucination.

Figure 5 illustrates the sampling logic generated for `check_tWTR`. Atomic entities such as `bg` and `ba` map directly to module input ports. Following the topological order of the DAG in Figure 2, constant-valued entities (`WR`, `RD`, `tWTR_cycle`) become `localparam` declarations. Spatial derivations such as `bank_id` and `same_bank` are expressed with combinational `assign` statements. Temporal derivations such as `prev_WR_cycle` and `wr_bank_id` require sequential state and are placed inside `always_ff` blocks that latch values upon detecting a `WR` command.

4.3.2 Covergroup Construct Generation. Once the sampling module is assembled, Phase 2 issues a single LLM call per relation to synthesise the covergroup construct. The prompt combines the relation metadata $r \in \mathcal{R}$ (constraint type, constraint formal expression, and participating entity names) with the signal declarations produced by the sampling stage: the former specifies *what* must be covered, while the latter specifies *what* can be observed.

To guide LLM generation, each constraint type is associated with a covergroup template that is included in the prompt as a structural scaffold. For *temporal* constraints, the template partitions an observed timing interval signal into three bins relative to a threshold τ: `illegal_bins` for values below τ, a boundary bin at τ, and a compliance bin above it. For *implication* constraints, the cause and effect are instantiated as separate coverpoints and combined via a `cross` construct. For *exclusion* constraints, the prohibited cases are enumerated as `illegal_bins`. For *enumeration* constraints, all valid cases are listed exhaustively as named bins.

In the coverage model example of `check_tWTR` (Figure 5), the constraint is temporal: the generated covergroup defines a single coverpoint on `cycle_WR_to_RD`, partitioned into an `illegal_bins` region below `tWTR_cycle`, a boundary bin at `tWTR_cycle`, and a compliance bin above it, gated by an `iff(sample)` clause that excludes non-triggered cycles.

5 Experiments

5.1 Benchmark Construction

We construct our benchmark from the JEDEC specifications of DDR5 and LPDDR5, two widely adopted DRAM protocols with complex verification requirements. We draw specification chunks from chapters covering initialization and training flow, mode register configuration, command protocol, functionality, and timing constraints, as these chapters contain the constraints that verification engineers routinely target when writing coverage models in practice.

https://www.jedec.org/standards-documents/docs/jesd79-5d
https://www.jedec.org/standards-documents/docs/jesd209-5c

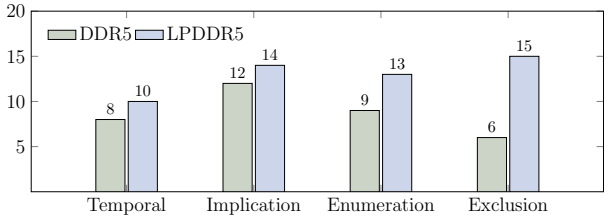


Figure 6: Distribution of relation subtypes in the DDR5 (35 constraints) and LPDDR5 (52 constraints) benchmarks.

Each benchmark instance is a tuple $(c, E, \{(d_i, t_i, m_i, p_i)\})$, where c is a specification chunk, E is the list of associated entity IRs, and each checkpoint comprises a natural-language description d_i, a constraint type t_i, a golden coverage model m_i, and a corresponding set of test patterns p_i used to validate functional correctness. A single chunk may yield multiple checkpoints. Checkpoints are classified into four constraint types: *temporal*, capturing ordering or timing-interval requirements between signals; *implication*, specifying that a behavior must hold when a condition is satisfied; *enumeration*, asserting that a signal's value must belong to a defined legal set; and *exclusion*, prohibiting certain signal combinations or value assignments. Figure 6 summarizes the benchmark composition.

All instances were constructed by professional verification engineers with expertise in DRAM protocol verification. Each instance, including entity IRs, checkpoint descriptions, golden coverage models, and test patterns, was independently reviewed by a second engineer through cross-validation. The golden coverage models along with the test patterns were further validated by compiling and executing them in an industrial EDA environment.

5.2 Experimental Setup

Implementation Details. We use GPT-4o as the backbone LLM for both the document comprehension and code generation phases, with the maximum token length set to 2048 for all tasks. The specification document is first converted into Markdown at the paragraph level, where each table is treated as an individual paragraph and figures are excluded. Entity IR extraction is performed over the entire document to ensure comprehensive coverage of all relevant entities. Relation IR extraction is then performed on the benchmark chunks, each containing one or more targeted verification constraints. For each chunk and its associated entity IRs, the model extracts one or more relation IRs along with any derived entity IRs. In the code generation stage, each IR is translated into a coverage model encapsulated in an independent module. All coverage models are compiled and simulated using Synopsys VCS 2016 on Ubuntu 18.04.

Baselines. We introduce an `End-to-End` baseline to evaluate the necessity of the proposed intermediate representation in coverage model generation. Since the full specification spans hundreds of pages and far exceeds practical context limits, feeding the entire document to an LLM is infeasible. Instead, the baseline provides the LLM with the specification paragraphs that are precisely relevant to the sampling logic of each benchmark checkpoint, and directly asks the model to generate the corresponding coverage model without any intermediate representation. The model is instructed to produce the sampling logic and coverage group for the target checkpoints and wrap them in an independent module to enable isolated testing. We evaluate this baseline with four backend LLMs: GPT-4o, Qwen3-max, DeepSeek-V3.2, and GLM-5. All hyperparameters, including the maximum token length, are kept identical to those used in ChatFCM, which employs GPT-4o throughout its pipeline.

Evaluation Metrics. We evaluate ChatFCM along two dimensions: IR construction quality and code generation quality.

On the IR construction side, we measure entity IR extraction by the recall of key fields defined in each entity type's schema (e.g., `encoding` for commands, `value` for configuration registers). Given the set of atomic entities $\mathcal{E}$:

$$\text{Recall}_{\text{entity}} = \frac{1}{|\mathcal{E}|} \sum_{e \in \mathcal{E}} \frac{|F_e^{\text{ext}} \cap F_e^{\text{gold}}|}{|F_e^{\text{gold}}|}, \tag{8}$$

where F_e^{gold} and F_e^{ext} denote the key fields in the golden and extracted entity IR. Derived entities $\mathcal{E}^d$ are excluded from IR-level evaluation, as the same relation IR may admit multiple valid derivation chains; their correctness is instead captured in code generation. For relation IR, we measure *checkpoint coverage*: whether the generated relation IRs semantically cover all target checkpoints associated with each chunk. For the two end-to-end baselines, checkpoint coverage is measured analogously over the generated code. Formally:

$$\text{Cov} = \frac{1}{|S|} \sum_{s \in S} \frac{|C_s^{\text{covered}}|}{|C_s|}, \tag{9}$$

where S is the set of specification chunks and $C_s^{\text{covered}} \subseteq C_s$ is the subset of checkpoints judged as covered.

On the code generation side, we measure *compilability* (*Cmp* for short) as the fraction of generated modules that pass compilation under an industrial EDA toolchain. For modules that compile, we further measure *functional correctness* (*Fun* for short): the EDA tool executes each module against test patterns and exports a coverage report, and a module is deemed correct if its triggering behavior is equivalent to that of the golden model. Checkpoint coverage and functional correctness both require semantic judgment. For these two metrics, an LLM first produces a binary judgment for each instance, and a human reviewer subsequently verifies every judgment.

5.3 Main Results

Entity IR Extraction. We first validate the Entity IR extraction module by measuring the recall of entities referenced by the benchmark constraints. On DDR5 and LPDDR5, the recall reaches 97.8% and 98.3%, respectively. The few unrecovered entities are predominantly command encoding values, which do not participate in the subsequent Relation IR construction or coverage model generation, and thus have no impact on the final output.

Overall Comparison. Tables 1 and 2 summarize the results on the benchmarks of DDR5 and LPDDR5, respectively, in terms of checkpoint coverage (Cov), compilability (Cmp), and functional correctness (Fun). ChatFCM achieves an overall functional correctness of 88.6% and 82.7% on the two benchmarks, substantially outperforming all End-to-End baselines. Notably, ChatFCM and the strongest baseline GPT-4o (End-to-End) share the same underlying LLM, yet ChatFCM outperforms it by 45.7% on DDR5 and 40.4% on LPDDR5. This controlled comparison confirms that the performance gain stems from decoupling comprehension and synthesis via the structured IR, rather than from a more powerful model.

Table 1: Results on the DDR5 benchmark. Cov, Cmp, and Fun denote checkpoint coverage (%), compilability (%), and functional correctness (%), respectively.

Method	Model	Type I: temporal			Type II: implication			Type III: enumeration			Type IV: exclusion			Overall		
		Cov	Cmp	Fun	Cov	Cmp	Fun	Cov	Cmp	Fun	Cov	Cmp	Fun	Cov	Cmp	Fun
End-to-End	GPT-4o	62.5	50.0	25.0	75.0	75.0	58.3	33.3	33.3	33.3	**100.0**	**100.0**	50.0	65.7	62.9	42.9
	Qwen3-max	50.0	50.0	0	50.0	41.7	16.7	33.3	22.2	11.1	83.3	83.3	50.0	51.4	45.7	17.1
	DeepSeek-V3.2	**100.0**	87.5	0	66.7	41.7	33.3	88.9	33.3	22.2	**100.0**	83.3	16.7	85.7	57.1	20.0
	GLM-5	62.5	50.0	12.5	75.0	66.7	41.7	66.7	22.2	22.2	**100.0**	**100.0**	16.7	74.3	57.1	25.7
ChatFCM (ours)	GPT-4o	**100.0**	**100.0**	**75.0**	**100.0**	**100.0**	**91.7**	**88.9**	**88.9**	**88.9**	**100.0**	**100.0**	**100.0**	**97.1**	**97.1**	**88.6**

Table 2: Results on the LPDDR5 benchmark.

Method	Model	Type I: temporal			Type II: implication			Type III: enumeration			Type IV: exclusion			Overall		
		Cov	Cmp	Fun	Cov	Cmp	Fun	Cov	Cmp	Fun	Cov	Cmp	Fun	Cov	Cmp	Fun
End-to-End	GPT-4o	90.0	70.0	60.0	78.6	64.3	57.1	69.2	61.5	30.8	**100.0**	60.0	26.7	84.6	63.5	42.3
	Qwen3-max	90.0	80.0	30.0	92.9	85.7	0	84.6	46.2	15.4	**100.0**	**100.0**	46.7	92.3	78.8	23.1
	DeepSeek-V3.2	90.0	50.0	40.0	92.9	85.7	28.6	92.3	69.2	38.5	86.7	53.3	46.7	90.4	65.4	38.5
	GLM-5	**100.0**	**90.0**	20.0	**100.0**	92.8	71.4	84.6	61.5	38.5	80.0	80.0	73.3	90.4	80.8	53.8
ChatFCM (ours)	GPT-4o	90.0	**90.0**	**80.0**	**100.0**	**100.0**	**85.7**	**100.0**	**84.6**	**69.2**	**100.0**	**100.0**	**93.3**	**98.1**	**94.2**	**82.7**

Beyond Compilability. A consistent pattern emerges across baselines: even when compilability is high, functional correctness remains low. For instance, on LPDDR5, Qwen3-max attains 78.8% compilability yet only 23.1% functional correctness; GLM-5 achieves 80.8% compilability yet only 53.8% functional correctness. This indicates that end-to-end approaches can produce syntactically valid code, but fail to translate constraint semantics into functionally correct assertions. The root cause is that end-to-end generation conflates comprehension and synthesis into a single pass, so an error in resolving entity definitions or inter-entity dependencies propagates silently into the final output. ChatFCM addresses this by decomposing each constraint into a dependency DAG of derived entities and a formal relational expression during comprehension; the synthesis phase then traverses this DAG to generate code in which every dependency is resolved, significantly reducing the chance of semantic errors.

5.4 Analysis by Constraint Type

Baseline difficulty scales along two dimensions: information scatter and derivation depth. Type IV (*exclusion*) constraints are typically self-contained and shallow, so baselines perform best here. Type III (*enumeration*) is scatter-dominant, as valid values often involve combinations of multiple entities whose definitions span distant sections, causing the steepest baseline functionality drop (best: 33.3% on DDR5, 38.5% on LPDDR5); ChatFCM's IR captures these values systematically, lifting functionality to 88.9% and 69.2%. Type I (*temporal*) demands deep multi-level signal derivation, where baselines can locate relevant sections but fail to produce correct sampling logic, e.g., DeepSeek-V3.2 reaches 100% coverage on DDR5 yet 0% functionality; ChatFCM resolves this via dependency resolution, achieving 75.0% and 80.0%. Type II (*implication*) is moderate on both dimensions; baselines perform comparatively well, yet ChatFCM improves functionality by over 27% and 28% on DDR5 and LPDDR5 respectively.

5.5 Ablation Study

To isolate the contribution of the relation IR, we ablate it on DDR5 and denote the variant as *ChatFCM w/o Relation IR* (Figure 7). Removing

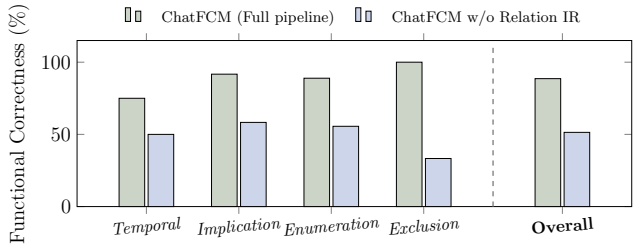


Figure 7: Ablation study on DDR5 benchmark.

the relation IR reduces overall functional correctness from 88.6% to 51.4%. The impact varies across constraint types: Type IV (*exclusion*) suffers the most severe drop (66.7%), as exclusion logic is defined entirely by inter-entity relations and cannot be recovered without explicit structural encoding; Type I (*temporal*) drops by 25%, losing the inter-signal dependency chains required for correct sampling logic; Type II (*implication*) and Type III (*enumeration*) each degrade by approximately 33%, reflecting balanced reliance on the relation layer for both structural constraints and value resolution.

Given the critical role of the relation IR, ensuring its correctness is equally important. The DAG-based IR validation and repair mechanism serves this purpose: on LPDDR5 benchmark, it automatically corrected one *implication* and one *temporal* instance whose initial IR contained dependency errors.

6 Conclusion

This paper presented ChatFCM, the first LLM-based framework for automatically generating functional coverage models with sampling logic from protocol specifications. ChatFCM separates comprehension from synthesis through a structured IR, avoiding the error-prone coupling of specification understanding and code generation in end-to-end approaches. Entities within each relation form a derivation DAG, enabling validation and repair before code generation and topologically ordered synthesis afterward. On 35 DDR5 and 52 LPDDR5 constraints drawn from JEDEC specifications, ChatFCM achieves 88.6% and 82.7% functional correctness, significantly outperforming the strongest end-to-end baseline.

References

[1] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[2] Z. He, Y. Pu, H. Wu, T. Qiu, and B. Yu, "Large language models for eda: Future or mirage?" *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 6, pp. 1–53, 2025.

[3] J. Pan, G. Zhou, C.-C. Chang, I. Jacobson, J. Hu, and Y. Chen, "A survey of research in large language models for electronic design automation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 3, pp. 1–21, 2025.

[4] Z. He, Y. Pu, H. Wu, Y. Qiu, T. Qiu, and B. Yu, "Large language models for eda:: From assistants to agents," *Foundations and Trends® in Electronic Design Automation*, vol. 14, no. 4, pp. 295–314, 2025.

[5] Z. Zang, Y. Song, A. Wang, B. W.-K. Ling, Q. Sun, Z. Lei, F. Yang, C. Zhuo, and J. Luo, "The dawn of agentic eda: A survey of autonomous digital chip design," *arXiv preprint arXiv:2512.23189*, 2025.

[6] 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," *arXiv preprint*, 2023.

[7] 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)*, 2024, pp. 722–727.

[8] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2023.

[9] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang, and B. Yu, "BetterV: Controlled Verilog Generation with Discriminative Guidance," *arXiv preprint*, 2024.

[10] X. Yao, Y. Wang, X. Li, Y. Lian, R. Chen, L. Chen, M. Yuan, H. Xu, and B. Yu, "Rtl-rewriter: Methodologies for large models aided rtl code optimization," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–7.

[11] X. Yao, H. Li, T. H. Chan, W. Xiao, M. Yuan, Y. Huang, L. Chen, and B. Yu, "Hdldebugger: Streamlining hdl debugging with large language models," *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 6, pp. 1–26, 2025.

[12] M. Akyash, K. Azar, and H. Kamali, "Decortl: A run-time decoding framework for rtl code generation with llms," *arXiv preprint arXiv:2507.02226*, 2025.

[13] S. Liu, W. Fang, Y. Lu, J. Wang, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 4, pp. 1448–1461, 2024.

[14] C.-T. Ho, H. Ren, and B. Khailany, "Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 1, 2025, pp. 300–307.

[15] K. Xu, J. Sun, Y. Hu, X. Fang, W. Shan, X. Wang, and Z. Jiang, "Meic: Re-thinking rtl debug automation using llms," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[16] 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*, vol. 43, no. 10, pp. 3184–3197, 2024.

[17] B.-Y. Wu, U. Sharma, A. Rovinski, and V. A. Chhabria, "Openroad agent: An intelligent self-correcting script generator for openroad," in *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*. IEEE, 2025, pp. 16–22.

[18] Z. Xiao, X. He, H. Wu, B. Yu, and Y. Guo, "Eda-copilot: A rag-powered intelligent assistant for eda tools," *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 6, pp. 1–24, 2025.

[19] N. Xu, Z. Zhang, S. Shu, L. Qi, J. Lv, W. Wang, T. Zhao, C. Zhang, Z. Yang, X. Li *et al.*, "iscript: A domain-adapted large language model and benchmark for physical design tcl script generation," *arXiv preprint arXiv:2603.04476*, 2026.

[20] 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)*, 2025, pp. 1–7.

[21] Y. Pu, Z. He, T. Qiu, H. Wu, and B. Yu, "Customized retrieval augmented generation and benchmarking for eda tool documentation qa," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[22] B.-Y. Wu, U. Sharma, S. R. D. Kankipati, A. Yadav, B. K. George, S. R. Guntupalli, A. Rovinski, and V. A. Chhabria, "Eda corpus: A large language model dataset for enhanced interaction with openroad," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–5.

[23] U. Sharma, B.-Y. Wu, S. R. D. Kankipati, V. A. Chhabria, and A. Rovinski, "Openroad-assistant: An open-source large language model for physical design tasks," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, 2024, pp. 1–7.

[24] A. Kaintura, S. S. Luar, I. I. Almeida *et al.*, "Orassistant: A custom rag-based conversational assistant for openroad," *arXiv preprint arXiv:2410.03845*, 2024.

[25] Y. Pu, Z. He, S. Lin, J. Qin, X. Zhang, H. Han, H. Zheng, Y. Jiang, C. Zhuo, Q. Sun *et al.*, "Mm-grade: A multi-modal eda tool documentation qa framework leveraging retrieval augmented generation," in *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2025, pp. 1–9.

[26] Y. Qin, Y. Pu, T. Qiu, Z. He, and B. Yu, "Infusing eda knowledge into llm systems: An information-source perspective," in *Proceedings of the 2026 International Symposium on Physical Design*, 2026, pp. 112–120.

[27] Y. Jiang, X. Lu, Q. Jin, Q. Sun, H. Wu, and C. Zhuo, "FabGPT: An efficient large multimodal model for complex wafer defect knowledge queries," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2024.

[28] Z. Wang, H. He, T. Wu, X. He, Q. Sun, C. Zhuo, B. Yu, J. Yu, and H. Geng, "Lmlitho: A large vision model-driven lithography simulation framework," in *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2025, pp. 1–9.

[29] Y. Lai, S. Lee, G. Chen, S. Poddar, M. Hu, D. Z. Pan, and P. Luo, "Analogcoder: Analog circuit design via training-free code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 1, 2025, pp. 379–387.

[30] Y. Yin, Y. Wang, B. Xu, and P. Li, "Ado-llm: Analog design bayesian optimization with in-context learning of large language models," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[31] G. Chen, K. Zhu, S. Kim, H. Zhu, Y. Lai, B. Yu, and D. Z. Pan, "Llm-enhanced bayesian optimization for efficient analog layout constraint generation," *arXiv preprint arXiv:2406.05250*, 2024.

[32] Y. Lai, S. Poddar, S. Lee, G. Chen, M. Hu, B. Yu, P. Luo, and D. Z. Pan, "Analogcoder-pro: Unifying analog circuit generation and optimization via multi-modal llms," *arXiv preprint arXiv:2508.02518*, 2025.

[33] L. Chen, R. Chen, S. Hu, X. Yao, Z. Tang, S. Kai, S. Xu, M. Yuan, J. Hao, B. Yu *et al.*, "Pcbagent: An agent-based framework for high-density printed circuit board placement," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, 2025, pp. 781–787.

[34] H. Zou, P. Han, E. Nazerian, and A. Q. Huang, "Pcbschemagen: Constraint-guided schematic design via llm for printed circuit boards (pcb)," *arXiv preprint arXiv:2602.00510*, 2026.

[35] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "Autobench: Automatic testbench generation and evaluation using llms for hdl design," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, 2024, pp. 1–10.

[36] G.-W. Wan, S. Su, J. Zhang, S. Z. Wong, M. Xing, L. Ji, Z. Jiang, X. Wang, and J. Yang, "Chattest: Coverage-enhanced testbench generation for agile hardware verification with llms," in *IEEE/ACM Proceedings Design, Automation and Test in Europe (DATE)*, 2026, pp. 1–7.

[37] N. S. Murthy, E. Nelson, S. S. Sapatnekar, and J. Sartori, "Verifllmbench: An open-source benchmark for testbenches generated with large language models," in *Proceedings of the Design & Verification Conference and Exhibition (DVCon US)*, 2025.

[38] J. Ye, Y. Hu, K. Xu, D. Pan, Q. Chen, J. Zhou, S. Zhao, X. Fang, X. Wang, N. Guan *et al.*, "From concept to practice: an automated llm-aided uvm machine for rtl verification," *arXiv preprint arXiv:2504.19959*, 2025.

[39] J. Ye, T. Liu, Q. Tian, S. Su, Z. Jiang, and X. Wang, "Chatmodel: Automating reference model design and verification with llms," *arXiv preprint arXiv:2506.15066*, 2025.

[40] Y. Bai, G. B. Hamad, S. Suhaib, and H. Ren, "Assertionforge: Enhancing formal verification assertion generation with structured representation of specifications and rtl," in *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*. IEEE, 2025, pp. 85–92.

[41] Z. Zhang, B. Szekely, P. Gimenes, G. Chadwick, H. McNally, J. Cheng, R. Mullins, and Y. Zhao, "Llm4dv: Using large language models for hardware test stimuli generation," in *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2025, pp. 133–137.

[42] R. Ma, Y. Yang, Z. Liu, J. Zhang, M. Li, J. Huang, and G. Luo, "Verilogreader: Llm-aided hardware test generation," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–5.

[43] Y. Lu, C. Bai, Y. Zhao, Z. Zheng, Y. Lyu, M. Liu, and B. Yu, "Deepverifier: Learning to update test sequences for coverage-guided verification," *ACM Transactions on Design Automation of Electronic Systems*, vol. 31, no. 4, pp. 1–23, 2026.

[44] Y. Wang, J. Zhou, H. Lyu, Z. Chao, T. Wang, and H. Li, "Deepassert: An llm-aided verification framework with fine-grained assertion generation for modules with extracted module specifications," *arXiv preprint arXiv:2509.14668*, 2025.

[45] Z. Yan, W. Fang, M. Li, M. Li, S. Liu, Z. Xie, and H. Zhang, "Assertllm: Generating hardware verification assertions from design specifications via multi-llms," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, 2025, pp. 614–621.

[46] E. Tian, Y. Ci, Q. Yang, Y. Li, and Z. Lyu, "Assertcoder: Llm-based assertion generation via multimodal specification extraction," *arXiv preprint arXiv:2507.10338*, 2025.