Executable Semantic Alignment for Specification-Driven Program Generation

Anonymous authors
Paper under double-blind review

Abstract

Large language models (LLMs) have substantially improved natural-language-to-program generation, enabling specification-driven program synthesis. However, specification-driven program generation still faces the challenge of preserving execution semantics defined by specifications.In this work, we study the semantic-fidelity problem in specification-to-program transformation and propose Executable Semantic Alignment (ESA). ESA introduces explicit execution-semantic representations, recovers atomic execution behaviors through parameterized semantic interfaces, and decomposes compositional semantic recovery into parsing processes for different semantic blocks to construct Logical IR.We construct a cross-domain dataset containing 50 standard documents and 3,055 specification clauses. We find that although natural-language specifications exhibit high linguistic diversity, their corresponding execution behaviors show substantial semantic convergence. On this dataset, we evaluate semantic fidelity from three perspectives: end-to-end program generation, atomic execution-semantic recovery, and compositional semantic reconstruction. Experimental results show that ESA effectively improves semantic fidelity in specification-driven program generation and validates the effectiveness of explicit semantic representations and structured Logical IR construction.

1 Introduction

Large language models (LLMs) have substantially improved natural-language-to-program generation and program synthesis, advancing the generation of executable programs from natural-language descriptions (Chen et al., 2021; Austin et al., 2021; Li et al., 2022; Nijkamp et al., 2022; Roziere et al., 2023; Li et al., 2023; Guo et al., 2024).

Recent studies have explored specification-driven program generation, where normative documents such as standards and regulations are transformed into executable programs (Li et al., 2026). However, since LLMs generate programs through probabilistic language modeling, their outputs may appear consistent with specifications at the linguistic level while failing to preserve the underlying executable semantics (Brown et al., 2020). Ensuring semantic fidelity between generated programs and original specifications therefore remains a key challenge (Li et al., 2026).

We study semantic fidelity in specification-to-program transformation, where a specification S is transformed into a program P.The goal is for the generated program to implement the executable semantics defined by the specification.

Existing semantic parsing studies demonstrate that natural language can be transformed into logical forms and structured representations, establishing mappings between linguistic expressions and computational structures (Berant et al., 2013; Yih et al., 2015; Liu et al., 2024).However, specification-driven program generation requires not only semantic interpretation but also preservation of executable behaviors during program implementation.

Despite these advances, an important question remains underexplored in specification-driven program generation:

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Does the surface diversity of natural-language specifications correspond to an equally complex underlying executable semantic space?

If the openness of the specification language space does not imply equivalent complexity in the executable semantic space, specification-to-program transformation may benefit from explicitly recovering shared executable semantic structures. To investigate this question, we construct a dataset containing 50 standard documents across different domains and characterize their executable semantic structures. We find that despite substantial expressive diversity in specifications, many differently expressed conditions can be mapped to shared executable patterns. Therefore, we represent atomic executable semantics as parameterized executable semantic interfaces, defined as $e_i = g_{k_i}(\theta_i)$, where g_{k_i} denotes an executable semantic interface and θ_i represents instance-specific bindings.

However, atomic executable semantics alone cannot determine complete specification behavior. Multiple semantic units must be further organized through scope, composition relations, and conditional dependencies to form complete compositional semantic structures (Partee, 1975; Liang, 2013).

Compositional semantic recovery requires models to explicitly represent structural relationships among different semantic units. Existing studies on structured representations and intermediate representations demonstrate that explicit structures can facilitate complex program generation and transformation, while reducing the search space for structural recovery (Agarwal et al., 2024; Jiang et al., 2025).

Motivated by this idea, we introduce Logical IR to represent compositional semantic structures and decompose compositional semantic recovery into separate parsing processes for different semantic blocks.

Based on parameterized executable semantic interfaces and Logical IR, we propose Executable Semantic Alignment (ESA),

which formulates specification-to-program transformation as $S \to E_S \to L_S \to P$.where E_S denotes the set of atomic executable semantics extracted from specifications, and L_S denotes the Logical IR constructed upon atomic semantics.

ESA decomposes specification-to-program transformation into two levels: Atomic Semantic Alignment, which recovers executable meanings of atomic conditions, and Compositional Semantic Alignment, which reconstructs semantic structures among units through Logical IR.

Our contributions are summarized as follows:

1. Executable Semantic Benchmark and Structure Characterization. We construct a cross-domain benchmark for specification-to-program semantic fidelity evaluation, containing 50 standard documents and 3,055 standard clauses. The benchmark provides executable data carriers, atomic execution semantics, and compositional semantic structures for evaluating semantic preservation during specification-to-program transformation.

   Analysis on this dataset shows that although specifications exhibit substantial linguistic diversity, different specification expressions correspond to convergent execution behaviors and can be represented through parameterized executable semantic interfaces.

2. Executable Semantic Alignment Framework. We propose ESA, which achieves semantic alignment in specification-to-program transformation through two stages. First, ESA recovers atomic executable semantics from specifications through parameterized executable semantic interfaces. Then, ESA reconstructs structural relations among atomic semantic units through compositional semantic parsing and constructs Logical IR.

   Through atomic executable semantic recovery and compositional semantic parsing, ESA introduces explicit semantic representations between specifications and program implementations, supporting the alignment between specification behaviors and program behaviors.

3. Multi-level Semantic Fidelity Evaluation. We systematically evaluate specification-to-program transformation from three complementary perspectives: end-to-end program generation, atomic executable semantic recovery, and compositional semantic reconstruction. Through comparisons with different generation strategies, models, and module-level ablations, we analyze the effectiveness of ESA and demonstrate the impact of explicit semantic representations on preserving specification behaviors.

2 Related Work

Large language models have substantially advanced natural-language-to-program generation. Code-focused models, such as Code Llama and DeepSeek-Coder, demonstrate strong capabilities in generating executable programs from natural-language descriptions (Roziere et al., 2023; Guo et al., 2024). Recent studies further investigate code reasoning, structured generation, and execution-based feedback to improve the reliability of program generation (Chen et al., 2023; Yao et al., 2023; Le et al., 2022; Ni et al., 2023). These studies provide important foundations for generating executable programs.

Structured representations and intermediate process modeling have also been explored to improve complex reasoning and generation tasks. Existing approaches show that explicit intermediate structures can provide additional constraints and facilitate the organization of complex reasoning processes (Yao et al., 2023; Chen et al., 2023). Meanwhile, semantic parsing studies investigate mappings between natural language and structured semantic representations, providing foundations for transforming language expressions into computational structures (Berant et al., 2013; Yih et al., 2015; Liang, 2016).

Different from these studies, ESA focuses on semantic preservation in specification-driven program generation. ESA represents atomic execution behaviors through parameterized executable semantic interfaces and models compositional semantic structures through Logical IR, enabling explicit alignment between specification semantics and program implementations.

3 Method

3.1 Overview

Figure 1 illustrates the overall framework of ESA. Given a standard document S and its structured data carrier D, ESA aims to generate an executable program P satisfying the requirements specified by the standard.

The specification-driven program generation task is formulated as $(S, D) \rightarrow P$.

ESA is motivated by two observations. First, although natural-language specifications exhibit diverse expressions, many specification conditions correspond to structured executable patterns. ESA therefore maps specification conditions into a parameterized executable semantic interface space.

Second, complete specification behaviors depend on compositional structures among multiple semantic units. ESA separates semantic composition from program synthesis and recovers these structures through explicit semantic representations.

Overall, ESA formulates specification-to-program transformation as $S \rightarrow E_S \rightarrow L_S \rightarrow P$, where E_S denotes recovered atomic executable semantics and L_S denotes the Logical IR constructed over atomic semantics.

The final program is generated as $P = \text{Generate}(E_S, L_S)$.

3.2 Atomic Semantic Alignment

Individual constraints in standards usually correspond to local executable behaviors, such as comparisons, state checks, and constraint conditions.

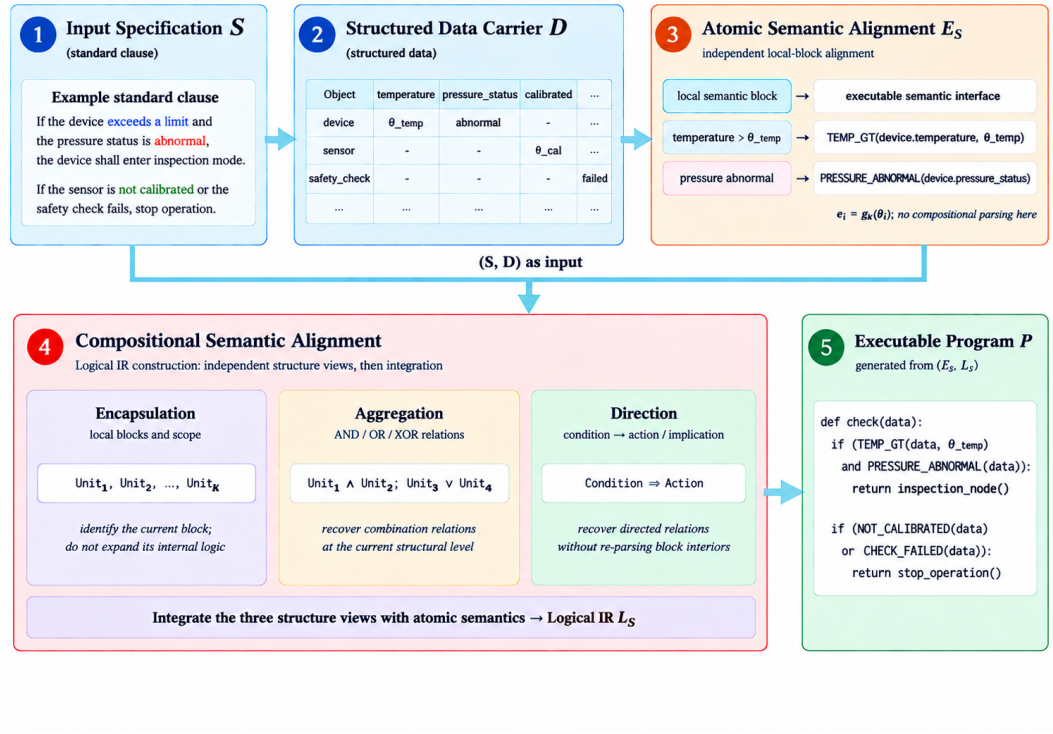


Figure 1: Overview of the Executable Semantic Alignment (ESA) framework. Given a standard clause and its structured data carrier, ESA first recovers atomic executable semantics, then reconstructs compositional semantic structures through Logical IR, and finally generates an executable compliance program.

ESA represents atomic execution behaviors using parameterized executable semantic interfaces, defined as $e_i = g_{k_i}(\theta_i)$, where g_{k_i} denotes the k_i-th executable semantic interface and θ_i contains the corresponding object, parameter, and constraint bindings.

For each atomic condition, ESA first filters candidate interfaces according to specification objects and condition descriptions, reducing the semantic search space. For ambiguous cases, ESA generates boundary test cases for candidate interfaces and obtains execution feedback to distinguish their behavioral differences. The LLM then aligns the final executable semantic interface by jointly considering the specification condition, candidate interfaces, parameter constraints, and execution feedback.

The recovered atomic semantic set is $E_S = \{e_1, e_2, \ldots, e_n\}$, which provides the basis for subsequent compositional semantic reconstruction and program generation.

3.3 Compositional Semantic Alignment

Atomic executable semantics describe local execution meanings, while complete specification behaviors depend on compositional structures among multiple semantic units.

ESA separates compositional semantic reconstruction from program generation and parses different types of semantic blocks. Specifically, ESA analyzes encapsulation, aggregation, and direction semantic blocks, where each block focuses on a specific type of semantic structure.

After semantic block parsing, ESA constructs Logical IR by composing atomic semantics with the recovered semantic blocks: $L_S = \text{Compose}(E_S, B)$, where B denotes the structured representation composed of different semantic blocks.

Finally, ESA generates the executable program based on atomic semantics and Logical IR: $P = \text{Generate}(E_S, L_S)$, where E_S preserves atomic execution meanings and L_S preserves compositional semantic structures.

4 Experimental

4.1 Dataset and Task Decomposition

For each clause, we provide a structured data carrier and executable test cases covering the boundary conditions of all atomic decision points. Detailed construction procedures are provided in the Appendix.

After manual filtering, the dataset contains 3,055 valid specification clauses and 9,634 atomic decision points. Clauses requiring additional non-textual information, such as images, external tables, or other references, are excluded to ensure that the evaluation focuses on executable semantic recovery from textual specifications.

To analyze the structural complexity of specifications, we further characterize each clause according to its logical depth and compositional width. The detailed distribution statistics are provided in the Appendix.

Furthermore, we construct logical ASTs describing compositional relations among atomic semantics within specification clauses. These structures provide the ground truth for evaluating compositional semantic reconstruction.

Based on this dataset, we evaluate specification-to-program transformation from three perspectives: end-to-end program generation, atomic semantic recovery with repair, and compositional semantic reconstruction with fixed atomic semantics. For atomic semantic evaluation, we sample 15 instances for each of the 68 executable semantic interfaces, resulting in 1,020 instances evaluated over three independent runs.

4.2 Models and General Settings.

Unless otherwise specified, all experiments use temperature 0.2, and each method is independently executed three times.

For end-to-end specification-to-program generation, we evaluate: DeepSeek-V4.1-Flash, Qwen3.8b, and Qwen3-Coder-30B-A3B-Instruct.

For atomic semantic alignment and test-driven repair, we use DeepSeek-V4.1-Flash.

For compositional semantic reconstruction, we use DeepSeek-V4.1-Flash.

4.3 End-to-end Specification-to-Program Generation

This experiment evaluates the ability of different approaches to generate executable compliance programs from standard specifications and structured data carriers. The goal is to analyze how different generation strategies affect semantic fidelity in specification-to-program transformation.

Compared Strategies. We compare four general generation strategies (Direct, CoT, Few-shot, and Planning) and three semantic modeling strategies (Semantic Representation, Semantic Representation with Atomic Repair, and ESA).

Metrics For each occupied depth–width cell c, let $p_{c,r}$ denote the test pass rate of the r-th independent generation run, and let w_c denote the number of instances in the corresponding cell.

We first report a distribution-weighted average pass rate, defined as $\overline{p}_{\text{weighted}} = \frac{\sum_c w_c \left(\frac{1}{3}\sum_{r=1}^{3} p_{c,r}\right)}{\sum_c w_c}$.

To evaluate generation stability across independent runs, we report three consistency metrics. The common correctness across all three runs is defined as $P_{\text{common}} = \frac{|V_1 \cap V_2 \cap V_3|}{|V|}$. The pairwise consistency bounds are defined as $P_{2-\text{pass min}} = \min_{i \neq j} \frac{|V_i \cap V_j|}{|V|}$ and $P_{2-\text{pass max}} = \max_{i \neq j} \frac{|V_i \cap V_j|}{|V|}$.

Here, P_{common} measures the proportion of cases that remain correct across all independent runs, while $P_{2-\text{pass min}}$ and $P_{2-\text{pass max}}$ characterize the lower and upper bounds of pairwise consistency among runs, respectively.

For visualization, the heatmaps report the within-cell mean pass rate, computed as $\overline{p}_c = \frac{p_{c,1}+p_{c,2}+p_{c,3}}{3}$.

Unlike the weighted average metric, the heatmap does not apply additional weighting across cells and is used to analyze the impact of specification structural complexity on model performance.

End-to-end Semantic Fidelity The end-to-end experiment evaluates the impact of different semantic modeling strategies on semantic fidelity in specification-to-program generation. Table 1 reports the weighted mean pass rate and stability metrics under different models and strategies.

Across all three models, ESA achieves higher end-to-end semantic fidelity compared with Direct Generation. ESA improves the weighted mean pass rate by 16.22, 16.70, and 16.80 percentage points on Qwen3-Coder-30B-A3B-Instruct, Qwen3.8B, and DeepSeek-V4.1-Flash, respectively. These results indicate that explicit executable semantic alignment provides more reliable intermediate representations for specification-to-program transformation.

General generation strategies, including Chain-of-Thought, Few-shot Prompting, and Planning, do not consistently improve performance across models. In contrast, semantic representation-based approaches achieve more stable improvements, suggesting that explicitly modeling executable semantics provides additional constraints for preserving specification behaviors. Semantic Representation with Atomic Repair brings limited additional gains, indicating that execution feedback alone is insufficient to resolve semantic mismatches without correct semantic abstractions.

Table 1: End-to-end specification-to-program generation results. The first column reports the weighted mean pass rate, followed by three stability metrics.

Model + Strategy	Weighted Mean Pass Rate	Common Pass Rate	Two-run Min Pass Rate	Two-run Max Pass Rate
Qwen3-Coder-30B-A3B-Instruct + Direct Generation	61.57%	47.13%	48.47%	56.59%
Qwen3-Coder-30B-A3B-Instruct + Chain-of-Thought (CoT)	60.32%	45.63%	47.55%	54.99%
Qwen3-Coder-30B-A3B-Instruct + Few-shot Prompting	43.32%	33.07%	34.16%	37.09%
Qwen3-Coder-30B-A3B-Instruct + Planning	59.66%	45.27%	49.03%	55.55%
Qwen3-Coder-30B-A3B-Instruct + Semantic Representation	73.11%	54.16%	62.78%	67.13%
Qwen3-Coder-30B-A3B-Instruct + Semantic Representation with Atomic Repair	73.20%	52.36%	59.74%	64.20%
Qwen3-Coder-30B-A3B-Instruct + ESA	77.79%	57.70%	64.91%	69.99%
Qwen3.8B + Direct Generation	63.32%	51.17%	53.35%	59.04%
Qwen3.8B + Chain-of-Thought (CoT)	61.55%	52.78%	52.40%	60.21%
Qwen3.8B + Few-shot Prompting	54.32%	42.21%	44.69%	49.87%
Qwen3.8B + Planning	60.82%	51.04%	52.52%	56.92%
Qwen3.8B + Semantic Representation	74.64%	55.90%	62.79%	66.83%
Qwen3.8B + Semantic Representation with Atomic Repair	75.20%	58.03%	65.21%	69.99%
Qwen3.8B + ESA	80.02%	59.38%	66.61%	70.38%
DeepSeek-V4.1-Flash + Direct Generation	63.79%	44.98%	47.15%	55.12%
DeepSeek-V4.1-Flash + Chain-of-Thought (CoT)	60.60%	43.91%	47.54%	55.08%
DeepSeek-V4.1-Flash + Few-shot Prompting	48.15%	39.29%	40.34%	43.33%
DeepSeek-V4.1-Flash + Planning	65.54%	48.45%	51.28%	57.85%
DeepSeek-V4.1-Flash + Semantic Representation	76.68%	52.35%	60.24%	65.31%
DeepSeek-V4.1-Flash + Semantic Representation with Atomic Repair	77.57%	52.16%	59.89%	64.89%
DeepSeek-V4.1-Flash + ESA	80.59%	54.48%	63.57%	69.07%

4.4 Atomic Semantic Alignment and Test-driven Repair

4.5 Metrics

We evaluate atomic semantic recovery and test-driven repair from two granularity levels.

Core Logic Correctness. This metric measures whether individual atomic semantic predictions are consistent with the Gold execution semantics. Each executable semantic interface

contains 15 atomic clause instances, and the accuracy is averaged over three independent runs.

For the k-th executable semantic interface, the r-th run, and the j-th atomic clause, we define $a_{k,j,r} = 1$ if the recovered semantic matches the Gold semantic, and $a_{k,j,r} = 0$ otherwise.

The Core Logic Correctness is defined as $\text{CLC} = \frac{1}{68\times3\times15}\sum_{k=1}^{68}\sum_{r=1}^{3}\sum_{j=1}^{15} a_{k,j,r}$, which measures the average correctness of atomic semantic recovery across interfaces, runs, and clauses.

Complete Interface Recovery Rate. This metric evaluates whether a complete executable semantic interface can be correctly recovered in a single generation process. Different from Core Logic Correctness, which averages individual atomic judgments, this metric requires all 15 atomic clauses within one interface to be correct in the same run.

For interface k and run r, we define $E_{k,r} = \prod_{j=1}^{15} a_{k,j,r}$, where $E_{k,r} = 1$ indicates that all atomic clauses of interface k are consistent with Gold semantics in run r, otherwise $E_{k,r} = 0$.

The Complete Interface Recovery Rate is defined as $\text{CIR} = \frac{1}{68\times3}\sum_{k=1}^{68}\sum_{r=1}^{3} E_{k,r}$.

This metric evaluates complete interface recovery in a single generation process and measures the effect of test-driven repair on recovery performance.

4.6 Atomic Recovery

This experiment evaluates the ability of models to recover atomic executable semantics from individual specification conditions.

We analyze different executable semantic interfaces according to their Core Logic Correctness.

Table 2: Core Logic Correctness by semantic difficulty.

Difficulty Group	Mean Core Logic Correctness
Easy ($\geq 90\%$)	95.37%
Moderate ($60\%- < 90\%$)	75.90%
Difficult ($33\%- < 60\%$)	48.11%

The results show that executable semantic interfaces exhibit different recovery difficulties. Simple semantic patterns usually correspond to clearer execution structures, while interfaces involving complex constraint relations are more likely to introduce recovery errors.

Detailed interface-level recovery results are provided in the Appendix.

4.7 Test-driven Repair

To analyze the effect of execution feedback on atomic semantic recovery, we compare operator-level Pass@1 before and after test-driven repair.

Table 3: Operator-level Pass@1 before and after test-driven repair.

Metric	Value
Before repair (Operator-level Pass@1)	41.18%
After repair (Operator-level Pass@1)	40.69%
Overall change	-0.49 pp

The results show that test-driven repair does not improve operator-level semantic recovery under this setting.

For simple semantics, models often recover the correct operator logic directly, leaving limited correction space. For complex semantics, generated boundary tests may reveal behavioral inconsistencies, but the subsequent repair process may fail to identify the underlying semantic mismatch and produce an incorrect correction.

For example, REL_DIFF_EQ_BASE involves a continuous numerical constraint defined as $(C_1(O_1) - C_2(O_1))/C_2(O_1) = P_1$.

Finite test cases can expose behavioral differences, but the model may still generate an incorrect repair without correctly understanding the underlying semantic error.

Overall, test feedback provides additional behavioral evidence, while atomic executable semantic recovery still requires consistency with the original specification semantics.

4.8 Compositional Semantic Reconstruction

4.8.1 Metrics

A prediction is considered correct if its Logical IR is logically equivalent to the Gold Logical IR. We report the average Logical Equivalence Pass@1 over three independent runs, denoted as $\text{MeanPass} = \frac{1}{3}\sum_{r=1}^{3}\text{Pass}_r$. We also report the average token consumption per clause to measure the cost of different parsing strategies.

4.8.2 Parsing Strategy Comparison

We compare three strategies for compositional semantic reconstruction.

Direct Parsing directly takes the specification clause and its corresponding atomic execution semantics as input and generates the complete Logical IR in one step.

Iterative Parsing performs hierarchical parsing. It first identifies the outermost logical structure and then recursively refines unresolved regions until the complete Logical IR is generated.

ESA adopts the proposed compositional semantic parsing approach. It parses different types of semantic blocks and constructs Logical IR based on the parsed structures.

Table 4: Comparison of different parsing strategies for compositional semantic reconstruction.

Method	Mean Pass@1 (%)	Token Cost
Iterative Parsing	63.1	4,792
Direct Parsing	64.3	2,821
ESA	68.5	16,028

Direct Parsing and Iterative Parsing achieve mean logical equivalence accuracies of 64.3% and 63.1%, respectively. ESA achieves 68.5%, improving over Direct Parsing and Iterative Parsing by 4.2 and 5.4 percentage points, respectively. This result indicates that parsing semantic blocks and constructing Logical IR based on these structures can improve compositional semantic reconstruction.

In terms of token consumption, ESA requires more generation cost than the other strategies.

In terms of token consumption, ESA requires more generation cost than the other strategies. Although explicit semantic parsing introduces additional computational overhead, this overhead is necessary for constructing the structured semantic blocks and Logical IR, rather than being irrelevant output redundancy. The resulting structured semantic blocks provide an inspectable intermediate representation, allowing specification decisions to be explicitly traced and analyzed and supporting verification between specifications and program implementations (Gotel & Finkelstein, 1994; Cleland-Huang et al., 2012).

4.8.3 Ablation Study

To analyze the contribution of different semantic block parsing components for Logical IR construction, we conduct module-level ablation studies.

During compositional semantic reconstruction, ESA parses three types of semantic blocks: Encapsulation, Aggregation, and Direction.

We remove one semantic block parsing component at a time while keeping the other components unchanged:

w/o Encapsulation, w/o Aggregation, and w/o Direction.

Table 5: Ablation results of different semantic block parsing modules.

Method	Pass rate (%)
ESA (Full)	68.5
w/o Encapsulation	66.4
w/o Aggregation	66.1
w/o Direction	65.3

The ablation results show that removing any semantic block parsing module decreases Logical IR reconstruction performance. The full ESA model achieves 68.5% Pass rate, while removing Encapsulation, Aggregation, and Direction reduces the performance to 66.4%, 66.1%, and 65.3%, respectively. This indicates that different semantic block types provide complementary contributions to compositional semantic reconstruction.

Furthermore, each module provides a distinct intermediate semantic representation. Removing any module not only affects the corresponding semantic structure recovery but also removes the associated intermediate output, reducing the inspectability of specification decision processes.

5 Limitations and Future Work

The current dataset is mainly constructed from Chinese standard specifications. The generalization of ESA to multilingual and diverse normative texts remains to be explored.

The current evaluation relies on generated boundary tests and Gold execution semantics for behavioral validation. More direct metrics connecting semantic representations and program implementations remain an important direction for future work.

Although ESA introduces executable semantic interfaces and Logical IR, recovering complex compositional structures with implicit dependencies remains challenging. Future work will investigate more expressive semantic representations and fine-grained semantic alignment methods.

AI Assistance Disclosure

During the preparation of this paper, we used large language models to assist with literature retrieval, writing refinement, and manuscript preparation.

Specifically, AI tools were used for improving language clarity, organizing manuscript structure, exploring related work, and supporting the preparation of experimental descriptions.

All generated content was carefully reviewed and verified by the authors. The authors take full responsibility for the final manuscript and all reported results.

References

Mayank Agarwal et al. Structured code representations enable data-efficient adaptation of code language models. arXiv preprint arXiv:2401.10716, 2024.

Jacob Austin et al. Program synthesis with large language models. arXiv preprint arXiv:2108.07732, 2021.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In Proceedings of EMNLP, pp. 1533–1544, 2013.

Tom B. Brown et al. Language models are few-shot learners. Advances in Neural Information Processing Systems, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

Wenhu Chen et al. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. Transactions on Machine Learning Research, 2023.

Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman. Software and Systems Traceability. Springer, 2012.

Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In Proceedings of the IEEE International Conference on Requirements Engineering, 1994.

Daya Guo et al. DeepSeek-Coder: When the large language model meets programming—the rise of code intelligence. arXiv preprint arXiv:2401.14196, 2024.

Hailong Jiang et al. Can large language models understand intermediate representations in compilers? In International Conference on Machine Learning, 2025.

Hung Le et al. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In NeurIPS, 2022.

Raymond Li et al. Starcoder: May the source be with you! Transactions on Machine Learning Research, 2023.

Siyuan Li, Jian Chen, Rui Yao, Xuming Hu, Peilin Zhou, Weihua Qiu, Simin Zhang, Chucheng Dong, Zhiyao Li, Qipeng Xie, and Zixuan Yuan. Compliance-to-code: Enhancing financial compliance checking via code generation. In Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2026.

Yujia Li et al. Competition-level code generation with alphacode. Science, 2022.

Percy Liang. Lambda dependency-based compositional semantics. arXiv preprint arXiv:1309.4408, 2013.

Percy Liang. Learning executable semantic parsers for natural language understanding. Communications of the ACM, 59(9):68–76, 2016.

Jinxin Liu et al. How proficient are large language models in formal languages? an in-depth insight for knowledge base question answering. In Findings of ACL, 2024.

Ansong Ni et al. Lever: Learning to verify language-to-code generation with execution. In ICML, 2023.

Erik Nijkamp et al. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474, 2022.

Barbara Hall Partee. Montague grammar and generative semantics. In Formal Philosophy. 1975.

Baptiste Roziere et al. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950, 2023.

Shunyu Yao et al. React: Synergizing reasoning and acting in language models. In International Conference on Learning Representations, 2023.

Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. Semantic parsing via staged query graph generation: Question answering with knowledge base. In Proceedings of ACL, pp. 1321–1331, 2015.

A Dataset Statistics and Construction Details

A.1 Dataset Overview

We construct a cross-domain specification dataset from national standards. Each instance contains a natural-language specification clause, its corresponding structured data carrier, atomic decision points, compositional logical structure, and executable evaluation cases.

The dataset contains 50 national standard documents, covering 3,055 specification clauses and 9,634 atomic decision points. On average, each clause contains 3.15 atomic decision points.

To characterize the structural complexity of specification clauses, we further analyze their logical depth and logical width. Logical depth denotes the maximum nesting level of logical composition within a clause, while logical width denotes the maximum number of atomic decision points jointly affected by a single logical operator.

The logical structures in the dataset range from depth 1 to 5 and width 1 to 8.

Table 6: Dataset statistics.

Statistic	Quantity
Standard documents	50
Specification clauses	3,055
Atomic decision points	9,634
Average atomic points per clause	3.15
Logical depth range	1–5
Logical width range	1–8

A.2 Domain Coverage

The dataset contains 50 national standards from multiple application domains, including mechanical engineering, vehicles, energy, electrical engineering, instrumentation, engineering construction, and other related fields.

The complete list of standard identifiers and corresponding metadata is provided in the supplementary dataset package.

A.3 Logical Structure Distribution

To analyze the structural complexity of specification clauses, we characterize each clause using two structural properties: logical depth and logical width.

Logical depth represents the maximum nesting level of logical composition within a clause.

Logical width represents the maximum number of atomic decision points associated with a single logical operator.

Table 7 presents the distribution of clauses under different depth-width configurations.

Table 7: Distribution of specification clauses under different logical depths and widths.

Depth \ Width	W1	W2	W3	W4	W5	W6	W7	W8
D1	837	0	0	0	0	0	0	0
D2	0	735	202	89	54	30	29	14
D3	0	334	175	90	64	27	14	9
D4	0	91	65	49	32	15	13	6
D5	0	17	15	18	13	9	5	4

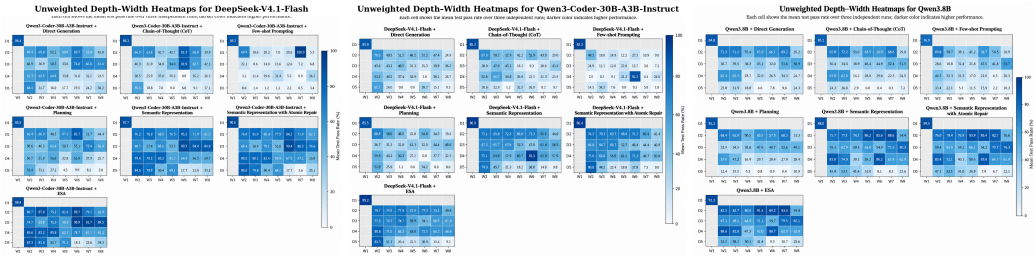


Figure 2: Depth–width performance heatmaps under different models. Each cell represents the mean pass rate across three independent runs. The heatmaps illustrate the impact of specification structural complexity on semantic fidelity.

A.4 Gold Logical Structure Construction

The gold logical structures are constructed through a multi-stage verification process.

Multiple large language models are first used to generate candidate logical structures from specification clauses. The generated structures are then iteratively compared and refined.

When inconsistent logical structures are produced, the conflicting cases are manually reviewed according to the original specification text.

This process reduces annotation ambiguity and improves the reliability of the gold logical structures used for compositional semantic evaluation.

A.5 Structured Data Carrier Construction

For each specification clause, we construct a structured data carrier containing the objects, attributes, and values required for executable evaluation.

The construction process combines automatic extraction and manual verification. Multiple model-based checks are applied to identify missing information, inconsistent descriptions, and ambiguous references.

Human verification is subsequently performed to ensure that the constructed data carriers provide sufficient information for semantic execution and evaluation.

A.6 Atomic Decision Point and Test Case Construction

Each atomic decision point is associated with executable evaluation cases derived from its corresponding gold semantic definition.

The test construction process first identifies the semantic boundaries of each atomic decision point and then generates executable cases covering different semantic states.

The final test cases are verified against the gold semantic definitions to ensure reliable behavioral evaluation of generated programs.

After verification, the dataset contains 154,144 executable evaluation cases, with each atomic decision point associated with evaluation cases for satisfied, violated, and boundary conditions when applicable.

A.7 Structural Complexity Analysis

To further analyze the impact of specification structural complexity, we characterize each clause according to its depth and width. Figure 2 presents the depth–width performance heatmaps for different models. Each cell reports the mean pass rate across three independent runs, providing a detailed view of how structural complexity affects semantic fidelity.

Across different models and generation strategies, increasing depth generally introduces a stronger performance degradation than increasing width. Specifications with larger width contain more parallel semantic units, which can often be processed through relatively independent local decisions. In contrast, increased depth introduces nested dependencies among semantic units, requiring models to preserve intermediate semantic relations across multiple levels. Therefore, deeper structures appear to introduce greater challenges for semantic preservation.

The performance degradation is not uniformly distributed across the depth–width space. For shallow structures, model performance remains relatively stable even when width increases, suggesting that a larger number of independent constraints does not necessarily lead to severe degradation. However, when depth increases, especially in regions with simultaneously large depth and width, performance drops more substantially. This pattern indicates that the accumulation of hierarchical dependencies may have a stronger impact on generation difficulty than the number of semantic units alone.

Different models exhibit different sensitivity to structural depth. The code model maintains relatively stable performance under increasing depth, showing stronger robustness in deeper regions. In comparison, Seek and Qwen3.8B exhibit more evident performance degradation as structural depth increases, suggesting that complex hierarchical semantic composition remains challenging when models rely primarily on implicit semantic reasoning.

ESA improves performance across most depth–width configurations and reduces the degradation associated with increasing structural complexity. The improvements are particularly visible in regions where baseline methods begin to decline, indicating that explicit executable semantic representations provide additional structural information for specification-to-program transformation. Nevertheless, performance decreases in extremely deep structures, suggesting that recovering complex hierarchical dependencies among multiple semantic units remains challenging.

Overall, the heatmaps indicate that specification complexity is influenced by both the number of semantic units and their organizational structures. The stronger sensitivity to depth compared with width suggests that hierarchical compositional dependencies may represent a more challenging aspect of semantic preservation. This observation is consistent with ESA's approach of explicitly modeling atomic semantics and compositional structures during specification-to-program transformation.

B Executable Semantic Interface Space Analysis

B.1 Interface Space Convergence

To investigate whether diverse specification expressions correspond to a compact executable semantic space, we analyze the convergence of executable semantic interfaces as the number of atomic decision points increases.

The executable semantic interface space contains 68 interfaces. Each interface represents a parameterized execution semantic unit:

$$I = (f, O, C, B)$$

where f denotes the execution function type, O denotes object binding, C denotes constraint conditions, and B denotes concrete parameters.

B.2 Convergence under Different Sampling Strategies

We analyze interface space convergence under two sampling strategies.

Sequential Coverage gradually accumulates atomic decision points according to their original order.

Random Sampling Coverage estimates the average discovered interfaces over 1,000 independent random sampling trials.

Figure 3 presents the convergence trends of the executable semantic interface space under both sampling strategies.

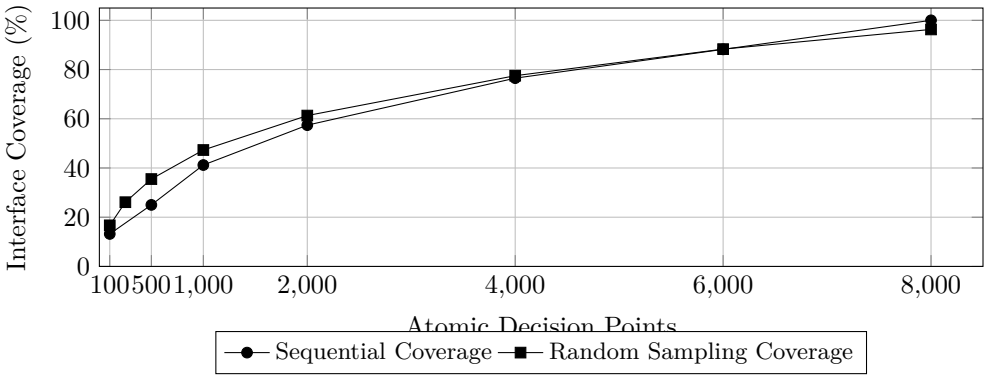


Figure 3: Interface space convergence under different sampling strategies.

The results show that both sequential accumulation and random sampling exhibit similar convergence patterns.

Although specification expressions exhibit substantial linguistic diversity, their corresponding executable behaviors gradually converge into a finite interface space.

With sufficient atomic decision points, the discovered interfaces approach the complete set of 68 executable semantic interfaces.

B.3 Interface Structural Characterization

After analyzing the convergence of the executable semantic interface space, we further characterize the structural properties of the 68 executable semantic interfaces from three perspectives: object binding complexity, semantic parameter complexity, and interface function family distribution.

B.3.1 Object Binding Complexity

Object binding complexity measures the number of entity objects required by an executable semantic interface.

The statistics are shown in Table 8.

Table 8: Object binding complexity of executable semantic interfaces.

Binding Complexity	Interface Number	Ratio
Single-object interface	45	66.18%
Two-object interface	21	30.88%
Three-object interface	2	2.94%
Total	68	100%

The results show that most executable semantic interfaces require only single-object semantic reasoning.

Specifically, 45 out of 68 interfaces (66.18%) involve only one object, while 21 interfaces (30.88%) require relations between two objects.

Only two interfaces require three-object relations.

Although natural-language specifications may describe complex relationships among multiple entities, executable semantic abstraction maps most behaviors into relatively low-order object binding structures.

B.3.2 Semantic Parameter Complexity

We further analyze the number of execution-related parameters required by each interface.

Shared fields such as objects and targets are excluded, and only parameters required for recovering execution behavior are counted.

The statistics are shown in Table 9.

Table 9: Semantic parameter complexity of executable semantic interfaces.

Parameter Complexity	Interface Number	Ratio
Two-parameter interfaces	26	38.24%
Three-parameter interfaces	30	44.12%
Variable-length interfaces	12	17.65%
Total	68	100%

The results show that most executable semantic interfaces can be represented using a limited number of parameters.

Specifically, fixed-parameter interfaces account for:

$$26 + 30 = 56$$

interfaces, corresponding to:

$$\frac{56}{68} = 82.35\%$$

Only 12 interfaces (17.65%) require variable-length parameters.

These interfaces mainly involve aggregation operations, such as SUM, PRODUCT, ALL_EQUAL, ALL_DISTINCT, MAX/MIN, and multi-object statistics.

Such interfaces require simultaneously recovering the input collection, aggregation scope, and aggregation relation, resulting in higher semantic complexity.

B.3.3 Interface Function Family Distribution

Finally, we classify the 68 executable semantic interfaces according to their functional semantic families.

Table 10: Functional family distribution of executable semantic interfaces.

Interface Family	Number	Ratio
Basic comparison	12	17.65%
Range and threshold	8	11.76%
Set relation	8	11.76%
Difference and distance	12	17.65%
Ratio and relative relation	8	11.76%
Arithmetic aggregation	10	14.71%
Counting statistics	4	5.88%
Special relation and predicate	4	5.88%
Geometric relation	2	2.94%
Total	68	100%

C Operator-Level Stability Analysis

C.1 Three-run Operator Stability

The main paper evaluates atomic semantic recovery using Core Logic Correctness and Complete Interface Recovery Rate.

In this appendix, we further analyze the stability of individual executable operators across three independent generations.

Different from interface-level recovery metrics, this analysis focuses only on whether the core operator semantics remain consistent across repeated generations.

For each operator, we define the three-run operator stability as:

$$\text{Operator Stability} = \frac{\#\text{Correct Generations}}{\#\text{Total Generations}}$$

where a generation is considered correct when the recovered operator logic matches the Gold operator semantics.

C.2 Operator Stability Results

Table 11 reports the stability of all 68 executable operators.

Table 11: Three-run operator stability. The metric measures the consistency of operator-level semantic recovery across three independent generations.

Operator	Stability	Operator	Stability
I001_EQ_CONST	100.0%	I035_ABS_GT_BASE	100.0%
I002_EQ_CROSS	100.0%	I036_ABS_DIFF_EQ_SAME	93.3%
I003_NE_CONST	0.0%	I037_ABS_DIFF_EQ_CROSS	0.0%
I004_NE_CROSS	100.0%	I038_ABS_DIFF_LT_SAME	66.7%
I005_GT_CONST	0.0%	I039_ABS_DIFF_LT_CROSS	0.0%
I006_GT_CROSS	100.0%	I040_ABS_DIFF_LE_SAME	0.0%
I007_GE_CONST	100.0%	I041_ABS_DIFF_LE_CROSS	0.0%
I008_GE_CROSS	100.0%	I042_ABS_DIFF_GT_SAME	0.0%
I009_LT_CONST	0.0%	I043_ABS_DIFF_GT_CROSS	0.0%
I010_LT_CROSS	100.0%	I044_RATIO_EQ_SAME	0.0%
I011_LE_CONST	96.7%	I045_RATIO_EQ_CROSS	100.0%
I012_LE_CROSS	100.0%	I046_RATIO_GE_SAME	100.0%
I013_BETWEEN_CC_BASE	66.7%	I047_RATIO_GE_CROSS	0.0%
I014_BETWEEN_OC_BASE	66.7%	I048_RATIO_LE_SAME	0.0%
I015_IN_BASE	33.3%	I049_RATIO_LE_CROSS	0.0%
I016_SUBSET_OF_SAME	66.7%	I050_RATIO_LT_BASE	100.0%
I017_SUBSET_OF_CROSS	66.7%	I051_REL_DIFF_EQ_BASE	60.0%
I018_INTERSECTS_SAME	100.0%	I052_REL_DIFF_LE_BASE	92.3%
I019_INTERSECTS_CROSS	66.7%	I053_PRODUCT_EQ_BASE	35.7%
I020_DISJOINT_SAME	100.0%	I054_PRODUCT_GE_BASE	46.7%
I021_DISJOINT_CROSS	100.0%	I055_PRODUCT_LE_BASE	92.3%
I022_ALL_EQUAL_SAME	100.0%	I056_PRODUCT_LT_BASE	90.0%
I023_ALL_EQUAL_MULTI	100.0%	I057_SUM_EQ_BASE	33.3%
I024_ALL_DISTINCT_SAME	0.0%	I058_SUM_GT_BASE	92.3%
I025_ALL_DISTINCT_MULTI	90.0%	I059_MEAN_EQ_BASE	66.7%
I026_DIFF_EQ_SAME	0.0%	I060_MAX_EQ_BASE	44.4%
I027_DIFF_EQ_CROSS	66.7%	I061_MIN_GE_BASE	66.7%
I028_DIFF_GT_SAME	0.0%	I062_RANGE_SPREAD_LE_BASE	93.3%
I029_DIFF_GT_CROSS	66.7%	I063_COUNT_EQ_BASE	60.0%
I030_DIFF_GE_SAME	100.0%	I064_COUNT_GE_BASE	46.7%
I031_DIFF_GE_CROSS	66.7%	I065_WITHIN_TOLERANCE_BASE	53.3%
I032_DIFF_LE_SAME	66.7%	I066_NONE_MATCH_BASE	53.3%
I033_DIFF_LE_CROSS	0.0%	I067_PARALLEL_BASE	100.0%
I034_ABS_LE_BASE	93.3%	I068_SATISFIES_BASE	93.3%

D Experimental Design Details

D.1 Model Selection

We select three models with different scales and training objectives to evaluate the effectiveness of ESA under diverse model capabilities.

DeepSeek-V4.1-Flash is selected as a strong general-purpose model with advanced language understanding and code generation capabilities. It provides a representative setting for evaluating semantic alignment on capable general LLMs.

Qwen3.8B is selected as a lightweight model to examine whether explicit semantic representations can provide additional benefits when model capacity is limited. Improvements on this model indicate that ESA does not rely solely on model scale but benefits from intermediate semantic structures.

Qwen3-Coder-30B-A3B-Instruct is selected as a code-specialized model to investigate whether ESA can further improve semantic fidelity beyond code generation capability. This setting evaluates whether ESA addresses the semantic alignment problem rather than only improving program generation ability.

Together, these models cover general-purpose reasoning models, lightweight models, and code-specialized models, enabling evaluation across different model characteristics.

D.2 Baseline Strategy Selection

We compare four commonly used generation strategies, including Direct Generation, Chain-of-Thought (CoT), Few-shot Prompting, and Planning. These strategies represent different approaches for improving LLM-based program generation.

Direct Generation directly generates executable programs from specifications and structured data carriers. It serves as the basic generation baseline and measures the capability of models without additional guidance.

Chain-of-Thought (CoT) introduces explicit reasoning steps before program generation. It evaluates whether additional reasoning traces can improve the understanding of complex specifications and preserve executable semantics.

Few-shot Prompting provides specification-program examples to guide generation. It evaluates whether additional demonstrations can reduce the difficulty of specification-to-program transformation through example-based learning.

Planning requires the model to construct a generation plan before producing programs. It evaluates whether explicit planning can improve complex generation processes through better task organization.

These strategies mainly improve reasoning, demonstrations, or generation procedures, while ESA focuses on explicit executable semantic representations for semantic alignment.

D.3 Atomic Repair Design

Execution feedback and automated repair mechanisms have been widely used to improve the reliability of generated programs. Therefore, we include Semantic Representation with Atomic Repair as a feedback-based enhancement strategy.

The goal of this setting is not to introduce a new repair algorithm, but to analyze whether execution feedback can further improve semantic recovery after semantic representation is constructed.

By comparing Semantic Representation and Semantic Representation with Atomic Repair, we evaluate the additional contribution of feedback-based correction beyond explicit semantic modeling.

D.4 Atomic Interface Sampling

Each executable semantic interface contains 15 atomic clause instances. This design balances semantic coverage and annotation cost.

Multiple instances are required because a single executable semantic interface may correspond to different object bindings, parameter settings, boundary conditions, and linguistic expressions. Evaluating multiple instances helps determine whether the model recovers the underlying executable behavior rather than memorizing specific expressions.

At the same time, atomic semantic evaluation requires Gold semantic annotation and execution verification. Therefore, 15 instances per interface provide a practical balance between coverage and verification cost, resulting in:

$$68 \times 15 = 1020$$

atomic semantic evaluation instances.

D.5 Boundary Test Construction

Atomic semantic errors often occur near decision boundaries, where small changes in input values can change execution outcomes.

Therefore, we construct boundary test cases to evaluate semantic consistency. For example, comparison-based constraints require cases covering satisfied, violated, and boundary conditions.

For a range constraint $a \leq x \leq b$, the test cases include values around the boundaries, such as $x = a$, $x = b$, values smaller than a, and values larger than b.

Such boundary-focused tests provide more informative execution feedback and help identify mismatches between generated semantics and Gold execution semantics.