SECO: Towards Code Compaction for Simulink via Verified Equivalence Extraction

Zehong Yu*, Zhuo Su✉†, Yixiao Yang‡, Haowei Qiu*, Rui Wang‡, and Yu Jiang✉*
*KLISS, BNRist, School of Software, Tsinghua University, Beijing 100084, China
†School of Software, Beihang University, Beijing 100191, China
‡Information Engineering College, Capital Normal University, Beijing 100089, China

***Abstract*—Simulink is widely used in embedded systems and provides automatic code generation. However, code size remains a critical concern in resource-constrained embedded systems. Existing code compaction techniques in code generators and compilers remain limited: they mainly rely on structural or syntactic similarity, and therefore are brittle to deviations from predefined patterns and ineffective at identifying semantically equivalent code segments. While Large Language Models (LLMs) exhibit semantic understanding capabilities, directly applying them in Simulink code generation remains impractical due to missing model semantics and the absence of correctness guarantees.**

In this paper, we propose SECO, a code compaction framework for Simulink models via verified equivalence extraction. *Our key insight is to leverage Simulink semantics as the partitioning context to guide LLMs in identifying candidate extractable segments, while providing equivalence checking to ensure soundness.* To enable LLMs to perceive Simulink semantics, SECO generates code for the target Simulink model, and augments it through subsystem embedding and parameterization. Then, SECO implements an equivalence extraction agent to identify candidate code segments and introduces a constraint checking agent to validate specifications and syntactic correctness. Once validated, SECO generates harness functions and verifies the semantic equivalence between candidate segments. Finally, SECO precompiles both original code and rewritten code and retains those rewrites that contribute to code size reduction. We implement SECO and evaluate it on benchmark Simulink models across three representative embedded architectures: ARM, Tricore, and RISC-V. Across these evaluations, SECO consistently outperforms Simulink Embedded Coder, Frodo, and SalSSA under `-Oz`, achieving an average of 38.2% reduction in assembly code size while introducing only 1.0% average runtime overhead.

***Index Terms*—Simulink, Embedded System, Code Generation, Semantic Equivalence**

I. Introduction

Safety-critical embedded systems in automotive, aerospace, and healthcare increasingly rely on model-based development to manage growing functional complexity [1], [2]. Simulink [3] is a dominant platform in this workflow: it abstracts system behaviors into high-level models and automatically generates production code. However, this generated code is ultimately deployed on low-cost and resource-constrained devices that are required to execute complex tasks [4]. As a result, code size becomes a critical concern, directly affecting memory footprint, system area, and cost [5]. In addition, the trend toward unified libraries and cross-architecture frameworks further amplifies the need for compact code across platforms [6], [7]. Consequently, generating space-efficient code becomes essential.

Both code generators and compilers have employed optimization techniques to reduce binary code size. Simulink Embedded Coder [8], for example, uses a matching-based method to eliminate code duplication, which identifies identical subsystems, *i.e.*, subsystems with the same blocks, parameters, and configurations, and merges them into shared function calls. Modern compilers, such as GCC [9] and Clang [10], further shrink code size through various optimizations, such as function inlining and dead code elimination. In addition, prior works leverage compiler intermediate representations (IRs) to implement targeted redundancy-reduction passes [4], [11]. For example, SalSSA [4] merges similar functions at the LLVM IR level using a sequence alignment algorithm. Overall, these techniques are largely driven by structural or syntactic similarity, with limited capability for semantic understanding.

Despite their success, existing works still face fundamental problems when applied to embedded systems: (1) *Intricate patterns.* Existing techniques rely on rule-based analyses to identify code similarity, but embedded code exhibits irregular patterns involving complex control structures, diverse expressions, and hardware features, making deep semantic equivalence difficult to capture. (2) *Incomplete identification.* Many compaction techniques stop at partial optimization stages. For example, Simulink Embedded Coder merges only identical subsystems into shared functions.Meanwhile, discovering structurally isomorphic code segments requires complex and brittle matching rules. The rules often fail when code segments deviate slightly from predefined patterns [4]. (3) *Platform-specific.* Compiler-based approaches are typically tied to specific intermediate representations. However, embedded platforms often rely on proprietary toolchains, *e.g.*, GCC for TriCore [12], making optimizations implemented on LLVM IR or other SSA-based representations inapplicable. Similarly, optimizations tuned for one architecture may not generalize to others. This hinders consistent and portable code-size reduction across heterogeneous embedded environments.

This motivates us to explore a new paradigm of code compaction for Simulink code generation, which leverages the capabilities of Large Language Models (LLMs) to shrink the generated code and rigorously validates the code correctness, enabling deployment in safety-critical embedded systems.

Recent advancements in LLMs have demonstrated semantic understanding capabilities in tasks such as code generation

✉Zhuo Su and Yu Jiang are the corresponding authors.

and program repair [13], [14], [15]. Given these capabilities, at first glance, using LLMs to extract semantically equivalent code segments is a reasonable and platform-agnostic way. **However, effectively applying LLMs in Simulink code generation while ensuring reliability in embedded systems remains challenging.** First, even with multimodal inputs, general-purpose LLMs are not well grounded in the domain-specific and highly structured semantics of Simulink models, making model-level reasoning and equivalence analysis unreliable [16]. Moreover, the generated code already omits high-level semantics of Simulink models, *e.g.,* hierarchical structure, which further makes it difficult for LLMs to capture equivalence at the model level. Second, LLMs occasionally shrink the code in a plausible yet incorrect way, for example, generating invalid definitions, and recent work also provides a mathematical proof suggesting that these hallucinations are inevitable. [17]. Furthermore, even when the resulting code is syntactically correct, LLMs provide no built-in correctness assurance, making them unsuitable for safety-critical embedded systems such as those in aerospace and automotive domains.

To tackle these challenges, we propose SECO, a code compaction framework for Simulink models via verified equivalence extraction. *Our key insight is to leverage Simulink semantics as the partitioning context to guide LLMs in identifying candidate code segments, while providing formal equivalence checking to rigorously verify their semantic equivalence.* First, SECO augments the embedded code with subsystem embedding and parameterization. These augmentations allow LLMs to reason about candidate code segments at the model level and avoid misleading cases caused by constant variations. Second, for candidate code segments, SECO incorporates a constraint checking agent that validates formatting rules, invokes the compiler to detect syntax issues, and iteratively repairs errors using failure feedback. Once validated, SECO generates harness functions for candidate segments, checks I/O equivalence between them using an SMT solver, and conservatively discards cases which fail the verification. Finally, SECO conducts pre-compile analysis on valid rewrites and retains the size-efficient ones.

We implemented and evaluated the effectiveness of SECO on benchmark Simulink models [18], across three representative embedded architectures, *i.e.*, ARM, Tricore, and RISC-V. The results demonstrate that SECO gains pronounced assembly code size reduction. Compared with the state-of-the-art code generators (Simulink Embedded Coder and Frodo) and the compiler-based tool (SalSSA), SECO achieves a 38.2% average reduction in assembly code size, when compiled with `-Oz` compilation flag (optimize for code size). Furthermore, we measured the execution time of the generated code, and the statistics show that SECO introduces only a 1.0% average runtime overhead. We further study the sensitivity of SECO to different LLM backends (`GPT-4o`, `GPT-5`, `Claude-Opus-4.6`, and `DeepSeek-V4-Pro`), and the results confirm that SECO remains effective across different code-capable LLMs.

In summary, we make the following contributions:

- We formulate a new paradigm of code compaction for Simulink, to extract semantically equivalent code segments, thereby ensuring resource-efficient development.
- We propose and implement SECO [1], which leverages Simulink semantics as partitioning context for LLMs in reasoning extractable segments and provides formal equivalence checking to ensure soundness.
- We evaluate SECO on benchmark models across three representative embedded architectures. The results illustrate that SECO effectively reduces the assembly code size, while incurring minimal runtime overhead.

II. Preliminaries of SECO

A. Model Driven Development and Simulink

Model Driven Development. As a software engineering approach, model driven development emphasizes the creation, refinement, and manipulation of abstract models as the primary artifacts. By shifting the focus from manual coding to model construction, it enables automatic code generation, thereby improving development efficiency and avoiding coding errors. These models also act as high-level blueprints of the target system, supporting simulation and formal verification to ensure functional correctness and compliance with specifications. Due to these benefits, model driven development has been widely adopted in the embedded system domain, particularly in safety-critical scenarios [19], [20], [21].

Simulink. As part of the MATLAB software suite [22], Simulink is one of the most widely used tools for model driven development, particularly in the context of embedded and real-time systems. It supports a wide range of application domains, including aerospace engineering, automotive design, and healthcare system [23], [24], [25]. Simulink provides a graphical block diagram environment to design and analyze the target systems, along with integrated toolsets for simulation, verification, and code generation.

Key Elements of Simulink. A Simulink model is built upon several components that define both the structural layout and execution behavior of the system. These modeling elements are organized hierarchically into two types: (1) *Atomic Block* is a fundamental unit representing a single operation; and (2) *Subsystem* is a composite block that encapsulates multiple atomic blocks and other subsystems. Developers often encapsulate cohesive functions into subsystems to improve model readability and manage complexity. However, equivalent but structurally different subsystems, or even directly copied subsystems, can lead to unnecessary code size growth. Conversely, subsystem semantics become essential for understanding the model and identifying such duplications in the generated code.

B. Equivalence Checking

When extracting code segments into reusable functions, it is essential to verify whether they exhibit identical behavior. This process, referred to as *equivalence checking*, determines whether two segments can be extracted without changing program behavior. We consider two levels of semantic equivalence. The first is *semantic equivalence*, which requires

[1]The artifact and the experimental results are available at the repository: https://github.com/YzhDDDing/SECO

that two functions produce identical outputs for all possible inputs. The second is *contextual equivalence*, a relaxed variant in which equivalence is guaranteed only under a specific execution context. These notions are formalized as follows:

Definition 1 (Semantic Equivalence)**.** *Let $f_1, f_2 : V_I \to V_O$ be two functions mapping inputs V_I to outputs V_O. We say that f_1 and f_2 are* semantically equivalent*, denoted $f_1 \equiv f_2$, if and only if:*

$$\forall\ \vec{x} \in V_I,\ \ f_1(\vec{x}) = f_2(\vec{x})$$

Definition 2 (Contextual Equivalence)**.** *Let $f_1 : V_I \to V_O$ be a function, and $f_s : V_I \times \mathcal{C} \to V_O$ be a shared function that takes an execution context from the context space $\mathcal{C}$. We say that f_s is* contextually equivalent *to f_1 under the specific context $c \in \mathcal{C}$, denoted $f_s \equiv_c f_1$, if and only if:*

$$\forall \vec{x} \in V_I,\ \ f_s(\vec{x}, c) = f_1(\vec{x}).$$

Definition 2 states that, once the context c is fixed, the shared function f_s behaves identically to the original function f_1 for all valid inputs. In SECO, the context c captures auxiliary values local to a candidate segment, such as local variables or constant values derived from the Simulink model, rather than arbitrary global states. Hence, the scope of c is explicitly bounded to instantiating the shared computation for a specific segment, without altering memory writes or externally observable states. Therefore, using f_s under the corresponding context does not introduce additional side effects.

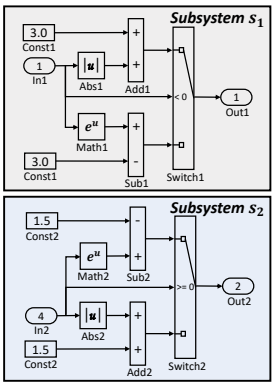
Corollary 1 (Soundness of Extraction)**.** *Let F be a set of functions. If, for each $f_i \in F$, either $f_s \equiv f_i$ or there exists a context c_i such that $f_s \equiv_{c_i} f_i$, then replacing each f_i with the corresponding use of f_s preserves the original behavior.*

Proof. Consider any $f_i \in F$. If $f_s \equiv f_i$, then by Definition 1, for every $\vec{x} \in V_I$, $f_s(\vec{x}) = f_i(\vec{x})$. If there exists a context c_i such that $f_s \equiv_{c_i} f_i$, then by Definition 2, for every $\vec{x} \in V_I$, $f_s(\vec{x}, c_i) = f_i(\vec{x})$. Thus, in either case, replacing f_i with the corresponding use of f_s preserves the output for the same input. Since this holds for every $f_i \in F$, replacing all such functions preserves the original behavior. □

In SECO, equivalence checking provides the formal basis for ensuring the soundness of LLM-assisted code compaction. While LLMs are used to identify potentially extractable code segments, equivalence checking determines whether these candidates can be safely rewritten into shared functions. By verifying that the shared implementation is input-output equivalent to the original code segment, either unconditionally or under an explicit context, SECO guarantees that the rewrite preserves the original behavior. This verification step is essential for filtering out unsound candidates and making the final compaction results reliable for safety-critical embedded systems.

III. Motivation

State-of-the-art code generators and compilers, *e.g.*, Simulink Embedded Coder, GCC, and Clang [8], [9], [10], employ various code-size optimization techniques. However, these methods largely rely on pattern matching to identify redundant code and often miss semantically equivalent segments beyond structural similarity. Such approaches require extensive hand-crafted rules to handle diverse cases, making them brittle under the intricate and irregular patterns of embedded code, *e.g.*, complex control flow and varied expressions. Moreover, some techniques are tied to specific platforms or IRs, limiting their applicability across heterogeneous embedded toolchains and leaving optimization opportunities unexplored.



(a) Simulink Model

```
void sample_step(void) {
  if (rtY.In1 < 0.0F) {
    rtW.Abs1 = fabsf(rtY.In1);
    rtW.Add1 = rtW.Abs1 + 3.0F;
    rtY.Out1 = rtW.Add1;
  } else {
    rtW.Math1 = expf(rtY.In1);
    rtW.Sub1 = rtW.Math1 - 3.0F;
    rtY.Out1 = rtW.Sub1;
  }
  if (rtY.In2 >= 0.0F) {
    rtW.Math2 = expf(rtY.In2);
    rtW.Sub2 = rtW.Math2 - 1.5F;
    rtY.Out2 = rtW.Sub2;
  } else {
    rtW.Abs2 = fabsf(rtY.In2);
    rtW.Add2 = rtW.Abs2 + 1.5F;
    rtY.Out2 = rtW.Add2;
  }
}
```

(b) Generated Code

```
sample_step():
  push    rdx
  movss   xmm0, DWORD PTR rtY[rip+12]
  xorps   xmm1, xmm1
  comiss  xmm1, xmm0
  jbe     .L10
  andps   xmm0, XMMWORD PTR .LC1[rip]
  movss   DWORD PTR rtDW[rip], xmm0
  addss   xmm0, DWORD PTR .LC2[rip]
  movss   DWORD PTR rtDW[rip+4], xmm0
  jmp     .L4
.L10:
  call    expf
  movss   DWORD PTR rtDW[rip+16], xmm0
  subss   xmm0, DWORD PTR .LC3[rip]
  movss   DWORD PTR rtDW[rip+8], xmm0
.L4:
  movss   DWORD PTR rtY[rip], xmm0
  movss   xmm0, DWORD PTR rtY[rip+20]
  xorps   xmm1, xmm1
  comiss  xmm0, xmm1
  jb      .L11
  call    expf
  movss   DWORD PTR rtDW[rip+28], xmm0
  subss   xmm0, DWORD PTR .LC4[rip]
  movss   DWORD PTR rtDW[rip+12], xmm0
  jmp     .L7
.L11:
  andps   xmm0, XMMWORD PTR .LC1[rip]
  movss   DWORD PTR rtDW[rip+20], xmm0
  addss   xmm0, DWORD PTR .LC5[rip]
  movss   DWORD PTR rtDW[rip+24], xmm0
.L7:
  movss   DWORD PTR rtY[rip+4], xmm0
  pop     rax
  ret
```

(c) Assembly Code

Fig. 1: A sample Simulink model containing two subsystems, along with the generated code on the left and the corresponding assembly code on the right.

A. Illustrative Example

We introduce a sample Simulink model in Figure 1 to further illustrate the aforementioned problems. Figure 1a shows the internal structure of the sample Simulink model, which contains two subsystems, namely s_1 and s_2. The major differences between them arise in `Const` blocks and `Switch` blocks. First, the values of `Const` blocks are different, *i.e.*, `Const1` $= 3.0$ in s_1 versus `Const2` $= 1.5$ in s_2. Second, the conditions and upstream logic of `Switch` blocks are inverted. In subsystem s_1, `Switch1` checks whether the value of `In1` is less than or equal to 0. If the condition holds, the output of `Add1` is forwarded to `Out1`; otherwise, the output of `Sub1` is used. By contrast, in the subsystem s_2, `Switch2` checks whether the value of `In2` is greater than 0. If the condition is satisfied, the output of `Sub2` is forwarded to `Out2`; otherwise, the output of `Add2` is selected.

```
void shared(float in, float c, float *out) {
  if (in < 0.0F) {
    float Abs = fabsf(in);
    float Add = Abs + c;
    *out = Add;
  } else {
    float Math = expf(in);
    float Sub = Math - c;
    *out = Sub;
}}
```

Fig. 2: A sample shared function f_s for extracting subsystem s_1 and s_2 in Figure 1.

Although this is one of the simpler instances, existing approaches fail to extract these code segments. Figure 1b and Figure 1c show the C code generated by Simulink Embedded Coder and the corresponding assembly code compiled using GCC with the `-Oz` flag. Simulink Embedded Coder generates separate code for subsystems s_1 and s_2. Memory-related optimizations employed by Simulink Embedded Coder are not applicable, as they require all blocks, parameters, and configurations within the subsystems to be identical for extraction. Moreover, GCC with the `-Oz` flag, which aims to minimize assembly code size, fails to merge these code segments. This also applies to other compilers such as Clang. While compilers are effective in optimizing arithmetic patterns and eliminating syntactic redundancies, they lack the capability to address more complex cases which require deep semantic reasoning and involve compensating transformations.

Although subsystems s_1 and s_2 are functionally different, they can be extracted into a shared function f_s for invocation, as illustrated in Figure 2. Note that all variables stored in the `rtW` shown in Figure 1b, such as `rtW.Abs1`, are local variables used to store intermediate results. These variables are scoped within their respective subsystems and are not referenced elsewhere in the model. Accordingly, f_s takes three parameters: `in` for input data, `out` for output results, and `c` as the predicate. Specifically, subsystems s_1 and s_2 are contextually equivalent to function f_s under different predicates: $f_s \equiv_{c=3.0} f_{s_1}$ and $f_s \equiv_{c=1.5} f_{s_2}$, respectively. Then, by Corollary 1, subsystems s_1 and s_2 can be safely extracted into f_s without altering the observable semantics, provided the corresponding predicate values are respected.

To quantitatively measure the benefits of the shared function f_s, we replace the code segments in Figure 1b with calls to f_s with corresponding parameters, and then compile the modified code along with the definition of f_s in Figure 2, using GCC with the `-Oz` flag. Compared to the original assembly code shown in Figure 1c, the modified version achieves a 19% reduction in code size (229 bytes versus 259 bytes). Moreover, in practice, cases similar to subsystems s_1 and s_2 are far more common than isolated pairs. Such cases present substantial opportunities for consolidation into shared functions, amplifying the potential for code size reduction. This further underscores the importance and necessity to fully exploit the extractable segments, especially in embedded scenarios with limited resources.

B. Solution

Large Language Models (LLMs) have recently shown significant capabilities in understanding and generating code [13], [14]. Industrial tools such as ChatGPT [26] and Cursor [27] are increasingly used to assist with programming tasks [28]. Google also reported that AI integration has improved engineer productivity by 10% [29]. These results are encouraging and indicate the potential of LLMs to identify and optimize semantically equivalent code segments beyond conventional code generators and compilers.

However, directly applying LLMs in Simulink-based embedded scenarios remains problematic. Such systems are commonly deployed in safety-critical domains, where correctness and reliability are mandatory requirements rather than optional goals. Moreover, Simulink models contain structured and domain-specific semantics that are not fully preserved in the generated code, making it difficult for LLMs to accurately reason about equivalence at the model level. As a result, any code compaction technique must not only reduce code size, but also guarantee that the optimized code preserves the behavior of the original implementation. This motivates us to explore a new paradigm of code compaction for Simulink code generation: *Can we effectively leverage the capabilities of LLMs to shrink generated code while providing formal guarantees that the optimized code remains behaviorally correct?*

Basic idea of SECO. Our key insight is to Simulink semantics as the partitioning context to guide LLMs in identifying candidate code segments, while providing formal checking to rigorously verify their semantic equivalence. Specifically, Simulink semantics provide model-level structure, such as subsystem boundaries and variable attributes, that is otherwise obscured in the generated code, enabling LLMs to reason about candidate segments beyond syntactic similarity. Meanwhile, SECO does not trust LLM outputs directly; it validates candidate rewrites through equivalence checking and applies only transformations that preserve the original input-output behavior. By combining semantic guidance with formal verification, SECO enables practical LLM-assisted code compaction for safety-critical embedded systems.

IV. SECO Design

Figure 3 illustrates the workflow of SECO. SECO mainly consists of three phases: ① **Optimization-Aided Generation**: Given a Simulink model, SECO generates the embedded code, augments the generated code by embedding subsystem metadata and parameterizing the constants, and constructs dataflow graph, laying the foundation for downstream optimization. ② **LLM-Assistant Extraction**: SECO designs equivalence extraction agent to identify candidate segments and generate shared functions. Besides, SECO includes constraint checking agent that validates the specifications and syntax, and iteratively repairs errors. ③ **Semantic Equivalence Checking**: SECO generates harness functions for each candidate segments, and uses the SMT solver to verify the semantic equivalence between harness functions and shared functions. If the verification fails, SECO records the failed candidate segments, corresponding shared functions, and counterexamples provided

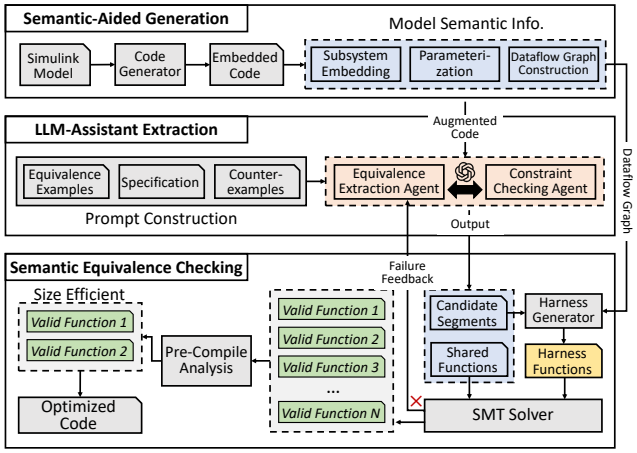


Fig. 3: Workflow of SECO: (1) Optimization-Aided Generation, which augments generated code with subsystem metadata and parameterization, and constructs the dataflow graph; (2) LLM-assistant Extraction, which identifies candidate segments, generates shared functions, and repairs invalid outputs; and (3) Semantic Equivalence Checking, which validates candidate rewrites using SMT-based harness checking and retains only size-reducing rewrites.

by SMT solver as failure feedback. The failure feedback is returned to Phase ② for refinement. After verification, SECO uses shared functions to rewrite corresponding extractable segments, and pre-compiles both the original code and rewritten code. Rewrites that lead to a reduction in code size are retained to synthesize the final optimized code.

A. Semantic-Aided Generation

SECO first uses Simulink Embedded Coder to generate code for the target Simulink model, which serves as the **semantic ground truth**: an optimized version of the code is considered correct if it is semantically equivalent to the code generated by Simulink Embedded Coder. Then, SECO collects semantic information from the target model. A Simulink model is internally stored as a `.zip` archive containing multiple `.xml` files, which encode the model configuration, blocks, connections, etc. SECO implements a customized parser to collect these contents.

Then, based on the collected model semantics, SECO augments the generated code and constructs dataflow graph. ① Subsystem Embedding: Using model structure and connections, SECO locates the inports and outports of each subsystem, and then embeds comments in the generated code to mark corresponding code regions. This metadata provides contextual information for locating extractable segments. ② Parameterization: Since different constants hinder code similarity identification, SECO defines a parameterization map M_p that associates each `Const` block with its value and replaces hard-coded constants in generated code accordingly. ③ Dataflow Graph Construction: SECO analyzes model connections to construct the dataflow graph, incorporating block properties, such as names, functionality, and data types, into graph nodes. Figure 4 shows an example of augmented code. Embedded

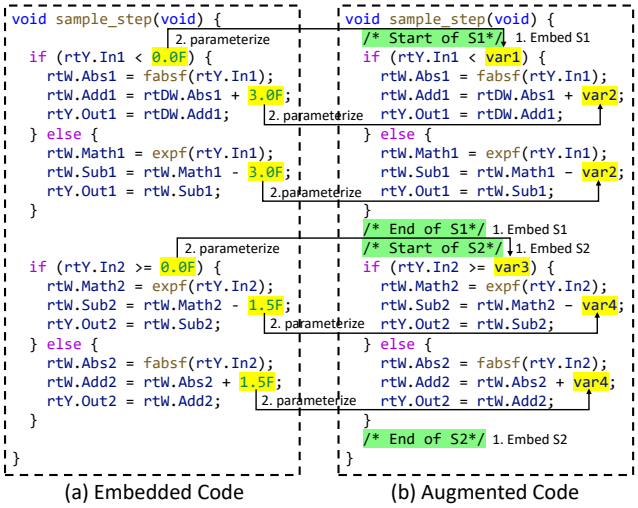


Fig. 4: An example of augmented code. Embedded subsystem metadata is highlighted in green, while parameterized constants are highlighted in yellow.

subsystem metadata is highlighted in green, where markers such as `/* Start of S1*/` and `/* End of S1*/` explicitly indicate the code region corresponding to subsystem `S1`. Parameterized constants are highlighted in yellow; for instance, constant value `3.0F` is replaced by `var2`.

B. LLM-Assistant Extraction

In this phase, SECO uses LLMs to identify candidate code segments for extraction and to validate both output specifications and syntax.

```
# Instruction
Given {Augmented C Code}, you should identify **equivalent code segments** and
define **shared functions** for merging them.
You can reason equivalent segments at two levels:
  - Treat subsystem as basic unit, and find equivalent code segments.
  - Directly find equivalent code segments.
# Examples
{Equivalence Examples} + {Counterexamples}
# Specifications
Subsystem semantic format; Shared function format; Identified segments format;
# Failure Feedback
Verification Failed: {Failure Feedback}
```

(a) Prompt template for equivalence extraction

```
# Instruction
Given {Shared Functions}, {Candidate Segments}, and {Specifications}, you should
check whether their formats are correct and repair errors.
```

(b) Prompt template for checking specification

```
# Instruction
Given {Shared Functions}, {Candidate Segments}, and {Compiler Feedback}, you
should repair syntactic errors.
```

(c) Prompt template for repairing syntactic errors

Fig. 5: Simplified prompt templates used by SECO.

Equivalence Extraction Agent. This agent, whose prompt template is shown in Figure 5a, identifies candidate equivalent segments and generates shared functions at two levels: (1) treat subsystems as basic units for finding equivalent segments; or (2) directly find equivalent segments. SECO provides representative equivalence examples, such as arithmetic and control flow patterns, along with counterexamples to illustrate non-equivalent cases. These examples serve as few-shot prompts

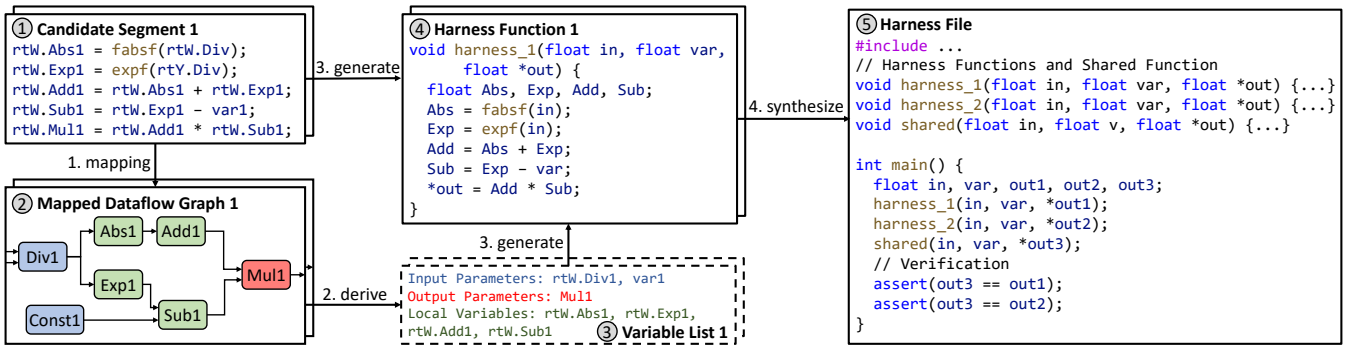


Fig. 6: An example illustrating the workflow of harness generation. A candidate code segment is first mapped to its corresponding dataflow graph, from which the input parameters, output parameters, and local variables are derived. Based on this information, SECO generates the corresponding harness function and then synthesizes a harness file for subsequent equivalence verification through output assertions.

to guide the agent's reasoning. Besides, SECO retrieves verification feedback (described in §IV-C) to iteratively refine the extracted candidates. SECO enforces specific formats for both shared functions and identified segments: (1) Shared function format: the return type must be `void`; parameter types must match the corresponding code segments; and output parameters must be declared as pointers. (2) Identified segment format: the segment must be labeled with the shared function to be invoked, and a corresponding rewrite call is generated.

Algorithm 1: Constraint Checking Agent Workflow

```
1  Function ConstraintChecking(functions, segments):
2      # Stage 1: Validate specification and repair violation
3      functions, segments ← CheckSpec(functions, segments)
4      # Stage 2: Validate compilation and repair syntactic errors
5      step ← 0
6      while step < COMPILE_ITER do
7          feedback ← Compile(functions)
8          if feedback = ∅ then
9              return functions, segments
10         else
11             functions ← RepairSyntax(feedback, functions)
12         step ← step + 1
13     return ∅
```

Constraint Checking Agent. LLMs may generate the shared functions that are plausible but incorrect, violating specifications and syntactic errors. Such inaccuracies render shared functions unreliable or unusable. To mitigate these hallucination issues, SECO employs the constraint checking agent to validate both formatting specifications and syntactic correctness. Algorithm 1 outlines the workflow of the constraint checking agent. The agent first validates the specifications of shared functions and candidate segments and repairs any violations, using the prompt shown in Figure 5b (Line 3). The validated specifications mainly focus on the following formatting requirements:

- *Function Signature.* The return type of shared function must be `void`. The parameter list should follow the order of appearance in the function code, with input parameters listed before output parameters.
- *Input Parameters.* Each input parameter must have a type consistent with the code segment.
- *Output Parameters.* Output parameters should be declared as pointers, with types matching the corresponding code segments.
- *Local Variables.* Local variables must be declared as function domain to avoid redefinition conflicts.

Then, SECO validates the syntactic correctness of the shared function. It first invokes the compiler (GCC by default) to compile the shared function and collects the resulting diagnostics (Line 7). If compilation succeeds, *i.e.*, the function is free of syntactic errors, SECO returns the validated function (Line 9). Otherwise, SECO uses the compiler's error messages to iteratively repair the code with the prompt in Figure 5c (Line 11). This process is repeated until either all syntax errors are resolved or a predefined attempt limit (`COMPILE_ITER`) is reached. If the limit is exceeded, SECO returns $\emptyset$ to indicate validation failure (Line 13).

C. Semantic Equivalence Checking

In this phase, SECO verifies the extractability between candidate segments and shared functions, *i.e.*, whether the shared function can safely replace corresponding candidate segments without affecting program behavior. The failed candidate segments and shared functions are recorded as feedback sent to equivalence extraction agent for refinement.

Harness Generation. SECO first generates harness functions for candidate segments, and then synthesizes them with corresponding shared function into the harness file for verification. Figure 6 illustrates this process. SECO begins by mapping the operations in the candidate segment to the dataflow graph obtained in optimization-aided generation. For example, as shown in Subfigure ②, `Candidate Segment 1` maps to the blocks `Div1`, `Const1` (parameterized as `var1`), `Abs1`, `Exp1`, `Add1`, `Sub1`, and `Mul1`. Next, SECO analyzes the connections in the mapped subgraph to derive the input parameters, output parameters, and local variables of the harness function. Specifically:

- A variable is an input parameter if its associated block has no inport or receives data outside the subgraph.

- A variable is an output parameter if its associated block has no outport or is consumed outside the subgraph
- All remaining variables are classified as local variables.

For instance, in Subfigure ④ of Figure 6, `Div1` block and `Const1` block represent input parameters, and `Mul1` block represents the output parameter. SECO then orders the input and output parameters according to their first appearance in the candidate segment and rewrites the segment using the corresponding parameter or local variable names. After that, SECO integrates the generated harness functions with corresponding shared function into a harness file. In the main routine of the harness file, SECO declares identical variables for input parameters of harness functions and shared function, while declaring distinct variables for output parameters. Then, SECO employs assertions to assess the equality of output variables produced by the shared function and each harness function, as shown in Subfigure ⑤. Note that, since LLMs may generate multiple shared functions for extracting different candidate segments, SECO synthesizes a separate harness file for each shared function.

Equivalence Verification. For each synthesized harness file, SECO uses CBMC [30] to verify the semantic equivalence between the shared function and its corresponding harness function. CBMC symbolically encodes program variables and evaluates the inserted assertions using SMT solvers such as Z3 [31]. Specifically, SECO verifies whether the output variables of both functions are identical for all valid inputs. If the assertion holds, the harness function is equivalent to the shared function, and by Corollary 1, the corresponding candidate segments can be safely replaced without changing observable program behavior.

For non-equivalent candidates, SECO provides the equivalence extraction agent with failure feedback consisting of the candidate segments, shared function, and a reduced CBMC counterexample. When verification fails, CBMC generates verbose traces containing unnecessary details, such as temporary assignments and intermediate states. SECO compresses these traces by retaining only the violated property, failure-inducing inputs, and output discrepancies between the harness and shared functions. This feedback guides the next refinement iteration, where the agent either revises the shared function or explores alternative candidates. The verification-refinement loop continues until all candidates are verified or the attempt limit (`VERIFY_ITER`) is reached, after which unverified candidates are discarded to preserve correctness.

Pre-Compile Analysis. After verification, SECO rewrites extractable segments with the corresponding shared functions. However, equivalence checking only ensures behavioral correctness, not code-size improvement. The actual size impact depends on the target platform and compiler behavior, including function-call overhead and backend optimizations. Thus, SECO compiles both the rewritten code and the original code using the target compiler and compares their resulting code sizes. A verified shared function is retained only if it produces a smaller compiled result than the original version. The retained rewrites are then used to synthesize the final optimized code, ensuring that SECO preserves only those transformations that are beneficial for code compaction.

V. Evaluation

In this section, we comprehensively evaluate the effectiveness of SECO on real-world Simulink models and address the following research questions:

- **RQ1:** Can SECO reduce the assembly code size compared to other baselines? (§V-A)
- **RQ2:** What is the runtime overhead of the generated code introduced by SECO? (§V-B)
- **RQ3:** What are the end-to-end optimization time and token usage of SECO? (§V-C)
- **RQ4:** Is subsystem semantics useful for identifying extractable code segments? (§V-D)
- **RQ5:** How sensitive the results are to the choice of different LLMs? (§V-E)

Benchmarks. We evaluate SECO on ten Simulink models collected from SLNET [18] and industrial pipelines [32], [33]. Table I summarizes their functionality, block counts, subsystem counts, and cyclomatic complexity. The benchmarks range from 129 to 1440 blocks and cover diverse block types, including discrete and DSP blocksets. Their cyclomatic complexity ranges from 4 to 54, further indicating diverse control-flow complexity across the benchmarks. Overall, these models cover representative embedded-system scenarios, from safety-critical control to complex signal processing, enabling comprehensive evaluation of SECO in realistic applications.

TABLE I: Details of benchmark models.

Model	Functionality	#Block	#Subsystem	#CC*
AN	Multi-channel auditory system	452	26	12
ABS	Safety anti-lock braking system	382	17	6
BP	Bacterial population simulation	139	8	10
DL	Electronic doorlock control system	368	19	20
Hybrid	HybridPass filter model	215	12	15
BC	Bionic cochlea model	129	11	8
OHC	Hearing aid DSP system	242	16	10
RAC	Robot arm control system	179	8	4
Quat	UAV attitude control system	1041	16	16
BLDC	Brushless DC motors	1440	8	54

* CC denotes cyclomatic complexity, obtained from Simulink.

Methodology. We compare SECO with two code generators, Simulink Embedded Coder [8] and Frodo [1], and a compiler-based optimizer, SalSSA [4]. For brevity, we use Simulink to refer to Simulink Embedded Coder in the remainder of this paper. These baselines represent generator-level and compiler-level code compaction approaches, which are the two dominant stages for existing code compaction. Unless otherwise specified, all methods use the `-Oz` optimization flag. To assess the impact on real-world deployment, we deploy optimized code on three embedded platforms: ARM (Cortex-A72), Tricore (Infineon TC387), and RISC-V (XuanTie C906). All LLM inference and verification processes run locally on an AMD Ryzen 7 5800X with 32 GB RAM.

Tool Implementation. We implement SECO in C++ and Python with 24K lines of code. By default, SECO uses `GPT-4o` (SECO-4o) and `GPT-5` (SECO-5) as lightweight and high-capacity LLM backends, respectively. SECO is

TABLE II: Comparison of the assembly code size of ARM, Tricore, and RISC-V architectures (Unit: Byte).

Model	Platform	Simulink	Frodo	SalSSA	SECO-4o	SECO-5
AN	ARM	2228	7230	2228	1744	**1684**
	Tricore	2654	3772	-	2056	**1998**
	RISC-V	2364	3809	-	1930	**1870**
ABS	ARM	1640	5626	1640	**1084**	**1084**
	Tricore	2792	3244	-	**1112**	**1112**
	RISC-V	1708	3197	-	**1112**	**1112**
BP	ARM	564	3069	564	**340**	**340**
	Tricore	792	1466	-	**374**	**374**
	RISC-V	392	1953	-	**292**	**292**
DL	ARM	1132	5685	1132	**764**	**764**
	Tricore	1708	2312	-	**870**	**870**
	RISC-V	950	2913	-	**582**	**582**
Hybrid	ARM	948	3842	948	572	**548**
	Tricore	1538	1998	-	662	**566**
	RISC-V	944	2397	-	586	**576**
BC	ARM	916	4521	916	**712**	**712**
	Tricore	1258	1736	-	**558**	**558**
	RISC-V	1014	2571	-	**698**	**698**
OHC	ARM	1780	5757	1780	1568	**1564**
	Tricore	2550	3244	-	1546	**1434**
	RISC-V	1830	3295	-	1484	**1450**
RAC	ARM	560	3289	560	**304**	**304**
	Tricore	984	1498	-	**352**	**352**
	RISC-V	576	2197	-	**292**	**292**
Quat	ARM	4128	5960	4128	3132	**3008**
	Tricore	7716	7914	-	6224	**5998**
	RISC-V	4492	5884	-	3408	**3326**
BLDC	ARM	4364	9652	4364	4120	**3464**
	Tricore	7684	8024	-	7052	**5202**
	RISC-V	4408	9214	-	4148	**3602**
Average Reduction	SECO-4o vs.	25.4% ↓	61.0% ↓	21.5% ↓	-	-
	SECO-5 vs.	31.3% ↓	64.1% ↓	26.2% ↓	-	-

backend-agnostic and supports any code-capable LLM for candidate identification and shared function generation. We further evaluate `Claude-Opus-4.6` (SECO-CC) and `DeepSeek-V4-Pro` (SECO-DS) in § V-E. Each equivalence check has a 300-second timeout. The default refinement limit is set to 3 for both compilation repair and verification (`COMPILE_ITER` and `VERIFY_ITER`), after which failed candidates are discarded.

A. Compared with Other Baselines

To evaluate the effectiveness of SECO, we compared the assembly code sizes produced by SECO-4o and SECO-5 against those produced by Simulink, Frodo, and SalSSA. For Simulink, we enabled the highest optimization level with the RAM-prioritized setting. On ARM, we use Clang for all generated code since SalSSA relies on LLVM IR, while GCC is used for Tricore and RISC-V due to their GCC-based toolchains. All platforms use the `-Oz` flag to prioritize code-size reduction, allowing us to evaluate whether SECO can provide additional compaction beyond compiler optimizations.

Table II summarizes the experimental results. Across the ARM, Tricore, and RISC-V platforms, SECO reduces the assembly code size by 5.6%–90.7% compared with Simulink and Frodo. Since SalSSA relies on LLVM IR and lacks GCC support, we evaluate it only on the ARM platform by applying it to Simulink-generated code. SECO further reduces the assembly code size by 5.6%–45.7% compared with SalSSA. These results demonstrate the effectiveness of SECO for code compaction, showing consistent size reduction across different models and target architectures.

TABLE III: Statistics of shared functions (SF), extracted code segments (ES), and discarded code segments (DS) of SECO-4o and SECO-5.

Model	SECO-4o			SECO-5		
	SF	ES	DS	SF	ES	DS
AN	3	11	3	3	10	0
ABS	4	13	0	4	13	0
BP	1	4	0	1	4	0
DL	2	9	2	2	9	2
Hybrid	1	4	2	2	6	0
BC	2	6	0	2	6	0
OHC	3	7	4	4	9	3
RAC	1	4	0	1	4	0
Quat	3	21	1	4	22	0
BLDC	2	11	2	7	27	2

In general, SECO-4o and SECO-5 outperform Simulink, Frodo, and SalSSA. Simulink employs memory-related optimizations to reduce code duplication, but these methods require the target subsystems to have identical blocks, parameters, and configurations. As a result, Simulink cannot capture extractable code segments that are semantically equivalent but structurally different. We also observe that Frodo produces significantly larger assembly code than the other tools, mainly because it generates separate functions for individual subsystems in the Simulink model, introducing additional function bodies and call overhead. SalSSA and compiler-based optimizations are also limited, because they operate after code generation and rely mainly on IR-level or syntactic redundancy without access to model-level Simulink semantics. Therefore, they are less effective at recognizing equivalence opportunities across structurally different but semantically related code segments. By leveraging Simulink semantics together with formal equivalence verification, SECO is able to exploit such opportunities and achieve better code compaction.

Comparing the two LLM configurations, SECO-4o and SECO-5 produce nearly identical improvements for most models, with SECO-5 offering marginally better reductions on a few cases (`OHC`, `AN`, `Quat` and `BLDC`). This suggests that SECO's semantic-guided optimization pipeline is robust even under a lighter-weight model, while higher-capacity reasoning provides slight additional gains. Table III helps explain this behavior. On several benchmarks, such as `ABS` and `DL`, SECO-4o and SECO-5 produce identical numbers of shared functions and extractable segments, and thus achieve the same final code size. By contrast, on complex models such as `Hybrid` and `OHC`, SECO-5 identifies more shared functions or extractable

```c
void harness_ne(float in, float c,
    float *out_harness)
{
    float Abs = fabsf(in);
    float Math = expf(in);
    float Add = Abs + c;
    float Mul = Add * Math;
    *out_harness = Mul / c;
}
```

(a) Candidate Segment

```c
void shared_ne(float in, float c,
    float *out_shared)
{
    float Abs = fabsf(in);
    float Math = expf(in);
    float Add = Abs * c;
    float Mul = Add * Math;
    *out_shared = Mul / c;
}
```

(b) Shared Function

Fig. 7: A rejected candidate due to non-equivalence. The candidate segment computes `Add = Abs + c`, whereas the generated shared function computes `Add = Abs * c`, resulting in different output values.

segments than SECO-4o, which leads to slightly better code compaction. For `AN`, although SECO-5 extracts fewer code segments than SECO-4o, the extracted segments are larger on average, which is consistent with its better compaction result. In addition, SECO-5 generally discards fewer candidate segments than SECO-4o, suggesting that its stronger reasoning capability helps generate more accurate extraction candidates that are more likely to pass verification. We further quantify this using the extraction success rate, defined as $\mathrm{ES}/(\mathrm{ES}+\mathrm{DS})$. Across all benchmarks, SECO-4o and SECO-5 achieve overall success rates of 86.5% and 94.0%, respectively. These results further demonstrate the effectiveness of SECO in identifying extractable code segments.

Case Study of Rejected Candidates. We further analyze the discarded code segments in Table III. In general, these candidates are rejected for two reasons: they either fail equivalence checking due to semantic mismatches, or remain unresolved because the verifier times out on complex candidates. Figure 7 shows a representative case of semantic mismatch. The candidate segment computes `Add = Abs + c`, whereas the generated shared function computes `Add = Abs * c`. This difference changes the output value, so the candidate cannot be safely replaced by the shared function. Such cases indicate that LLMs may still generate plausible but semantically incorrect shared functions, and equivalence checking is necessary to filter them out. Meanwhile, the counterexample is packaged as failure feedback and returned to the equivalence extraction agent for refinement. Guided by this feedback, the agent revises the shared function and candidate segments, allowing SECO to verify a corrected extraction in the following refinement round.

Figure 8 shows a candidate that remains unresolved during verification. For readability, the figure shows part of the computation and omits several additional floating-point statements in the candidate region. Even in this simplified view, the candidate already contains nonlinear floating-point operations, including `expf`, `sqrtf`, division, and branch-dependent computations. The omitted statements further increase the size and complexity of the symbolic formula that the SMT solver needs to reason about. As a result, the verification burden exceeds the 300s timeout used by SECO. This timeout does not indicate that the candidate is semantically incorrect; rather, it means that the verifier cannot establish equivalence within the given time budget. Rather than applying a rewrite whose equivalence is not formally established, SECO conservatively discards this candidate for soundness.

```c
void harness_timeout(float in1, float in2, float in3,
    float in4, float in5, float *out)
{
    float Abs = fabsf(in1);
    float Exp = expf(in2);
    float Div = Exp / (Abs + fabsf(in3));
    if (Div > 1.0F)
    {
        Div = sqrtf(Div * Div + in3);
        ... // complex floating-point computations
    }
    else
    {
        Div = expf(Div - in3);
        ... // complex floating-point computations
    }
    *out = Div / (fabsf(Mux) + in5);
}
```

Fig. 8: A candidate discarded due to verification timeout. The candidate contains nonlinear floating-point operations and control flow, making SMT-based equivalence checking expensive. SECO conservatively discards it when verification does not finish within the timeout.

B. Runtime Overhead

To evaluate the overhead introduced by shared functions, we deployed the generated code of SECO-4o, SECO-5, Simulink, and Frodo on the aforementioned embedded architectures, and measured their execution time. Consistently, the compiler optimization flag was set to `-Oz`. To minimize statistical errors, each generated code was executed 100,000 times to obtain average execution times.

Table IV presents the experimental results. Overall, SECO introduces only marginal runtime overhead while achieving substantial code-size reduction. Compared with Simulink, SECO-4o and SECO-5 incur an average execution-time overhead of only 0.6% and 1.4%, respectively, across the evaluated platforms. Compared with Frodo, however, SECO-4o and SECO-5 are on average 18.8% and 18.1% faster, respectively. On the ARM platform, where SalSSA is applicable, SECO-4o and SECO-5 show only 0.1% average overhead over SalSSA.

In general, the code generated by Simulink is faster than that produced by SECO on most benchmark models. This runtime overhead is primarily due to the use of shared functions, which introduce additional costs related to function calls and context preservation. However, the overhead is acceptable: SECO achieves a reduction of 5.6% – 64.2% in assembly code size, while introducing only 1.0% average runtime overhead across all models. More importantly, given the tight memory constraints of embedded devices, the space savings enabled by SECO can be used to support more applications and accommodate critical features such as diagnostic modules and fault-tolerance mechanisms.

Interestingly, we observed that on certain models, *e.g.*, `RAC` model, SECO-4o and SECO-5 outperform than Simulink. To investigate the underlying reason, we used `perf` [34] to analyze the executable files of `RAC` model on the ARM architecture. We found that, compared to Simulink, the number of `L1-dcache-loads` is reduced by 60.7% with SECO-4o and SECO-5 (120M versus 305M). These results suggest that

TABLE IV: Comparison of the execution time of ARM, Tricore, and RISC-V architectures (Unit: Second).

Model	Platform	Simulink	Frodo	SalSSA	SECO-4o	SECO-5
AN	ARM	1.054	1.087	1.054	**0.892**	0.931
	Tricore	**0.998**	1.273	-	1.083	1.071
	RISC-V	4.163	4.307	-	**3.634**	3.716
ABS	ARM	**0.667**	0.883	**0.667**	0.725	0.731
	Tricore	1.110	1.201	-	1.018	**1.013**
	RISC-V	**3.232**	4.479	-	3.601	3.601
BP	ARM	0.125	0.161	0.125	**0.122**	**0.122**
	Tricore	**0.286**	0.337	-	0.298	0.298
	RISC-V	**0.578**	0.791	-	0.634	0.634
DL	ARM	**0.205**	0.265	**0.205**	0.219	0.221
	Tricore	**0.523**	0.624	-	0.553	0.553
	RISC-V	**1.135**	1.516	-	1.221	1.221
Hybrid	ARM	0.336	0.342	0.336	**0.314**	0.328
	Tricore	0.607	0.643	-	**0.526**	0.530
	RISC-V	**1.695**	2.038	-	1.833	1.874
BC	ARM	**0.287**	0.346	**0.287**	0.318	0.319
	Tricore	**0.384**	0.516	-	0.424	0.420
	RISC-V	**1.116**	1.473	-	1.216	1.216
OHC	ARM	**1.291**	1.382	**1.291**	1.297	1.293
	Tricore	1.756	1.815	-	**1.630**	1.656
	RISC-V	**5.281**	6.114	-	5.436	5.341
RAC	ARM	0.166	0.184	0.166	**0.151**	**0.151**
	Tricore	0.388	0.425	-	**0.351**	**0.351**
	RISC-V	0.867	0.901	-	**0.857**	**0.857**
Quat	ARM	**3.011**	4.735	**3.011**	3.137	3.114
	Tricore	2.996	3.305	-	**2.630**	2.656
	RISC-V	**17.829**	21.176	-	18.742	18.669
BLDC	ARM	4.077	6.295	4.077	4.058	**4.021**
	Tricore	**2.688**	3.703	-	2.885	2.930
	RISC-V	19.441	24.595	-	**18.935**	19.553
Average Overhead	SECO-4o vs.	0.6% ↑	18.8% ↓	0.1% ↑	-	-
	SECO-5 vs.	1.4% ↑	18.1% ↓	0.1% ↑	-	-

consolidating extractable code segments into shared functions can improve data locality and reduce pressure on the data cache, thereby offsetting part of the function call overhead.

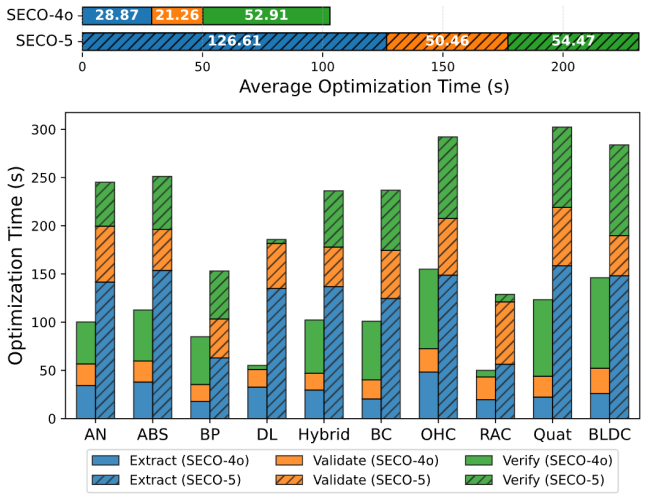


Fig. 9: The average optimization time of SECO-4o and SECO-5 for the benchmark models (top) and the optimization time for each model (bottom).

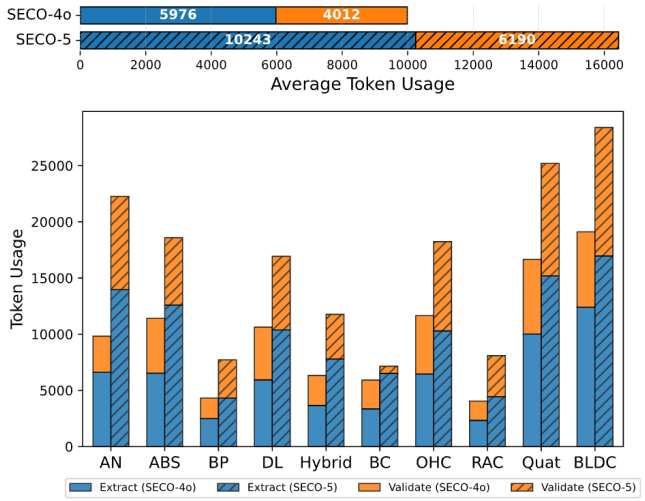


Fig. 10: The average token usage of SECO-4o and SECO-5 for the benchmark models (top) and the token usage for each model (bottom).

C. Optimization Time and Token Usage

For benchmark models, we recorded the optimization time of the three main phases: equivalence extraction agent (Extract Phase), constraint checking agent (Validate Phase), and semantic equivalence verification (Verify Phase). We omit precompile analysis from measurements, as it incurs negligible overhead due to the fast compilation process. Moreover, we measured the token usage for equivalence extraction agent and constraint checking agent.

Figure 9 and Figure 10 present the optimization time and token usage of SECO-4o and SECO-5 for the benchmark models. On average, SECO-4o takes 28.87s, 21.26s, and 52.91s for the equivalence extraction, constraint checking, and semantic equivalence verification, respectively. In comparison, SECO-5 takes 126.61s, 50.46s, and 54.57s for the corresponding phases. We found that, the verification time for `DL` and `RAC` models is significantly shorter. For `DL`, SECO-4o and SECO-5 require 4.22s and 4.06s, respectively, while for `RAC`, the verification time is 6.92s for SECO-4o and 7.7s for SECO-5. This disparity arises because these models operate on integer inputs, whereas others involve floating-point operations, which impose considerably higher computational overhead on the SMT solver during semantic equivalence checking. As for token usage, SECO-4o averagely uses 5,976 tokens for equivalence extraction and 4,012 tokens for constraint checking, while SECO-5 uses 10,243 tokens and 6,190 tokens, respectively. As expected, SECO-5 exhibits increased optimization time and token usage compared to SECO-4o, which can be attributed to the more complex reasoning chains required by the higher-capacity model. Based on these statistics, SECO-4o is generally more efficient, as it achieves similar reductions in assembly code size with lower time and token consumption.

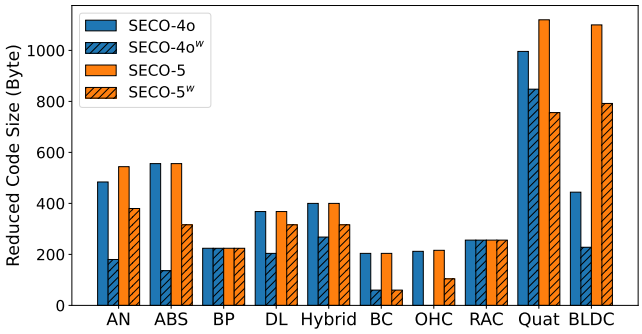


Fig. 11: Reduced assembly code size of SECO-4o, SECO-4o^w, SECO-5, and SECO-5^w compared to Simulink Embedded Coder on the ARM architecture.

D. Ablation Study

To investigate the contributions of subsystem semantics in guiding LLMs for identifying extractable segments, we implemented SECO-4o^w and SECO-5^w which disable subsystem embedding module. We then measured the reduced assembly code size of SECO-4o, SECO-4o^w, SECO-5, and SECO-5^w compared to Simulink across benchmark models.

Figure 11 shows the experimental results on ARM architecture. As the generated code is the same, the trends for Tricore and RISC-V architectures are consistent. The statistics illustrate that, SECO-4o and SECO-5 reduce more code size than their ablated counterparts on most benchmark models. Notably, on the `OHC` model, SECO-4o^w fails to achieve any code size reduction, while SECO-5^w still achieves some reduction, albeit less than SECO-5. These findings highlight the importance of embedding subsystem metadata, which partitions the generated code according to subsystem boundaries. Such metadata provides critical contextual information that guides LLMs in identifying extractable segments and uncovering optimizations beyond the immediate code structure.

For the majority of the models, SECO-5^w outperforms SECO-4o^w in terms of reduced assembly code size, due to higher-capacity reasoning. However, SECO-4o still outperforms SECO-5^w in most cases, which further underscores the importance of subsystem embedding for identifying extractable code segments. Exceptions arise in the `BP` and `RAC` models, where all four variants of SECO yield identical code size reductions. Further analysis reveals that both models are relatively simple. For instance, `RAC` contains only four extractable code segments, all of which are consolidated into a single shared function. In such scenarios, SECO-4o^w and SECO-5^w achieve comparable optimization outcomes.

E. Evaluation on Other LLMs

To evaluate the dependence of SECO on specific LLM backends, we instantiate the equivalence extraction and constraint checking agents with two additional code-capable LLMs: `Claude-Opus-4.6` (SECO-CC) and `DeepSeek-V4-Pro` (SECO-DS). All other components, including semantic-aided generation, harness generation, equivalence checking, and precompile analysis, remain unchanged. We evaluate different backends in terms of token usage, optimization time, extraction success rate, and assembly code-size reduction compared with Simulink, Frodo, and SalSSA.

TABLE V: Sensitivity of SECO to different LLM backends in terms of token usage, optimization time, extraction success rate, and average assembly code-size reduction over Simulink, Frodo, and SalSSA.

LLM	Token usage	Opt. time	Success rate	Size reduction		
				Simulink	Frodo	SalSSA
SECO-4o	8,012	95.1s	84.1%	37.9% ↓	70.5% ↓	31.3% ↓
SECO-5	13,842	216.1s	92.4%	38.4% ↓	71.1% ↓	31.6% ↓
SECO-CC	15,713	241.2s	94.3%	42.5% ↓	72.1% ↓	33.4% ↓
SECO-DS	18,209	417.8s	92.8%	40.8% ↓	71.6% ↓	32.8% ↓

Table V summarizes the experimental results. Overall, SECO remains effective across different LLM backends, while the offline optimization cost varies. Across the four evaluated LLMs, SECO achieves extraction success rates of 84.1%–94.3%, with average token usage ranging from 8,012 to 18,209 and average optimization time ranging from 95.1s to 417.8s. It reduces the assembly code size by 37.9%–42.5% compared with Simulink, 70.5%–72.1% compared with Frodo, and 31.3%–33.4% compared with SalSSA. These results indicate that SECO is not limited to a specific LLM backend; its workflow remains effective when different code-capable LLMs are used in the equivalence extraction agent and constraint checking agent.

The detailed metrics vary across LLM backends. Compared with SECO-4o and SECO-5, SECO-CC and SECO-DS generally achieve slightly better optimization effectiveness. Specifically, SECO-CC achieves the highest extraction success rate of 94.3%, while SECO-DS also reaches 92.8%, both outperforming SECO-4o and SECO-5. In terms of code-size reduction, SECO-CC reduces assembly code size by 42.5%, 72.1%, and 33.4% over Simulink, Frodo, and SalSSA, respectively, while SECO-DS achieves corresponding reductions of 40.8%, 71.6%, and 32.8%. This is mainly because stronger reasoning models can identify more accurate candidate segments and generate shared functions that are more likely to pass equivalence checking.

However, these gains come with higher offline cost. Compared with SECO-4o and SECO-5, SECO-CC and SECO-DS require more token usage and longer optimization time. For example, SECO-DS consumes 18,209 tokens on average and takes 417.8s, which is the largest among all evaluated backends. Further analysis shows that the optimization time of SECO-DS is mainly spent in candidate identification, where the equivalence extraction agent needs to reason over augmented code and generate candidate segments and shared functions. In addition, compared with GPT and Claude Code, `DeepSeek-V4-Pro` has lower generation throughput, which further increases the end-to-end optimization time. Overall, these results show a trade-off between optimization effectiveness and offline cost: more capable LLMs can expose more compaction opportunities, while lightweight models such as SECO-4o provide a better efficiency trade-off.

VI. Discussion

Integration with existing model-driven design tools. SECO can be integrated into the embedded development workflow as a backend code-compaction optimization. For Simulink, the integration mainly requires a model parser to extract model semantics, including subsystem boundaries, block connections, and model-to-code mappings. SECO then uses the extracted semantics to augment the generated code, identify equivalent candidate segments, verify candidate rewrites through equivalence checking, and emit optimized embedded code for deployment.

Beyond Simulink, SECO can be extended to other model-driven design tools, such as Ptolemy-II [35] and SCADE [36], where subsystem-like or operator-level abstractions are common. Such extension primarily requires implementing tool-specific parsers to recover the corresponding model semantics and code-to-model mappings. For example, supporting SCADE would require extracting operator boundaries, dataflow semantics, and their mappings to the generated code, so that these semantics can be embedded into the code and used for harness construction. Once this adaptation is available, the remaining components of SECO, including LLM-assisted extraction, constraint checking, semantic equivalence checking, and pre-compile analysis, can be largely reused.

Support for embedded platforms. SECO is largely platform-agnostic, because it performs optimization before backend lowering and does not rely on platform-specific ISA features. Accordingly, most components of SECO, including LLM-assistant extraction, constraint checking, harness generation, and semantic equivalence checking, can be reused across different embedded targets. The platform-dependent part mainly lies in the pre-compile analysis, where target-specific compilers are needed to evaluate the actual code-size effect of rewrites. While our evaluation covers ARM, Tricore, and RISC-V, extending SECO to other platforms primarily requires integrating the corresponding cross-compilation toolchains and build configurations.

Compatibility with compiler toolchains. Our evaluation uses different compiler toolchains depending on the target platform. For Tricore and RISC-V platforms, we use GCC-based toolchains, which are the standard compilation environments for these embedded targets. For ARM platform, we use Clang, because SalSSA operates on LLVM IR and therefore requires an LLVM-based compilation flow for comparison. This choice is made for fair evaluation rather than as a requirement of SECO itself. Since SECO identifies and rewrites semantically equivalent code segments at the C level, it can operate with different downstream compiler toolchains as long as they support compilation and code-size measurement. Our results on both GCC-based and LLVM-based compilation flows consistently show the effectiveness of SECO.

Bounded checking. SECO uses CBMC [30], a bounded C model checker, to verify the semantic equivalence between shared functions and harness functions. This is practical in our setting because SECO verifies localized candidate segments rather than the whole generated program. When the generated code contains loops, SECO identifies extractable segments inside the loop body instead of extracting the whole loop. Therefore, the synthesized harness functions for CBMC verification are loop-free. The original loop structure, loop bounds, and loop-control logic are not rewritten; SECO only replaces the verified segments inside the loop.

In Simulink-generated code, memory- and stateful behaviors are represented through generated global/state variables and deterministic reads and writes. SECO handles such variables by explicitly splitting each access into reads and writes. If an extracted snippet reads global/state variables, SECO passes their current values as inputs to the shared function. If the snippet updates global/state variables, SECO returns the updated values through output parameters and assigns them back to the corresponding variables. Therefore, state updates are preserved explicitly during rewriting. Within the scope of Simulink-generated code, functional correctness is captured by equivalence checking between the harness function and the shared function.

Trade-offs. SECO makes semantic-equivalent segment extraction practical by augmenting LLMs with model semantics. This enables SECO to identify extractable segments beyond syntactic similarity, without relying on brittle hand-crafted rules. The cost is incurred offline, including token usage, LLM-query time, and verification time, while the deployed code benefits from substantial code-size reduction without manual optimization effort. SECO also trades completeness for soundness during verification. Complex candidates may make SMT solving expensive, so SECO verifies each harness independently, enables parallel checking, and applies a time-out (300s) to discard unresolved cases. As a result, SECO may miss some optimization opportunities, but every retained rewrite is verified before being applied. Finally, SECO trades small runtime overhead for substantial code-size reduction. The overhead mainly comes from shared-function calls introduced by code compaction. In our evaluation, SECO reduces code size by 38.2% over Simulink with only 1.0% runtime overhead on average.

VII. Related Work

Code generators. Code generation is of vital importance for model driven development, and many works have made efforts to improve the quality of the generated code. Specifically, Simulink Embedded Coder [8] implements various optimizations to generate production-quality code. On the one hand, it employs expression folding and loop-related optimizations to enhance runtime performance. On the other hand, it generates separate functions for identical subsystems to reduce code duplication. Mercury [32] improves the runtime efficiency of generated code from a microarchitectural perspective. It analyzes data dependencies between model actors, estimates instruction latency, and adjusts the translation order to reduce instruction-pipeline stalls during execution. Frodo [1] focuses on eliminating redundant calculations in generated code. When the model contains data-truncation blocks, Frodo recursively determines the precise calculation range of each block to avoid unnecessary computations. However, these approaches mainly target runtime efficiency or redundant computation

elimination, while our work focuses on code compaction by extracting semantically equivalent code segments.

Compiler-based code size optimizations. Reducing code size has long been an important objective in compiler optimization, especially for resource-constrained embedded systems. A representative direction is function merging, which exploits similarity between functions to eliminate duplicated or near-duplicated code. Early work [37] reduces code size by merging structurally similar functions and inserting control flow to resolve small differences. LLVM also provides the `MergeFunctions` pass [38], which identifies equivalent or similar functions and merges them at the compiler IR level. More recent work further improves this line of research. SalSSA [4] supports function merging directly in SSA form and uses sequence alignment to detect structurally similar functions more effectively. HyFM [39] reduces the time and memory overhead of function merging through block-level alignment and early profitability analysis. However, these approaches still rely primarily on structural similarity and compiler IR representations. As a result, they often fail when target cases deviate from predefined matching patterns, and their applicability is limited in heterogeneous embedded settings where toolchains and compiler infrastructures differ.

LLM-based approaches. Recently, many studies have explored the use of LLMs for code optimization. Meta trains a 7B-parameter model on LLVM IR and assembly code to recommend compiler optimization options [40]. Supersonic [41] formulates code optimization as a seq2seq generation task and uses diff-based output representation to produce optimized code edits. RAPGen [42] further introduces retrieval-augmented generation to improve the quality of LLM-generated optimized code. These approaches demonstrate the potential of LLMs in compiler and code optimization, but they target general-purpose programs rather than model driven embedded code generation. Moreover, they lack awareness of Simulink-specific semantics and do not provide formal correctness guarantees, which limits their applicability in safety-critical embedded systems.

VIII. Conclusion

In this paper, we propose SECO, a code compaction framework for Simulink models. SECO first augments the generated code by parameterization and subsystem embedding. Then, SECO introduces equivalence extraction agent and constraint checking agent to identify and validate candidate segments. Finally, SECO verifies semantic equivalence through generated harnesses and retains only rewrites that reduce compiled code size. Compared to Simulink Embedded Coder and Frodo, SECO achieves the pronounced assembly code size reduction, while introducing minimal runtime overhead. further verifies semantic equivalence through generated harnesses and retains only rewrites that reduce compiled code size.

IX. Acknowledgment

This research is sponsored in part by the NSFC Program (No. 92582101, 62372263, 62525207), and by the "Ganwei Action Plan" of Beihang University (No. WZ2026-QN-35).

References

[1] Z. Yu, Z. Su, Y. Jiang, A. Cui, and R. Wang, "Efficient code generation for data-intensive simulink models via redundancy elimination," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3656217

[2] E. Mezghani, E. Exposito, and K. Drira, "A model-driven methodology for the design of autonomic and cognitive iot-based systems: Application to healthcare," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 224–234, 2017.

[3] MathWorks, *Simulink Documentation*, 2025, https://www.mathworks.com/products/simulink.html.

[4] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Effective function merging in the ssa form," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 854–868. [Online]. Available: https://doi.org/10.1145/3385412.3386030

[5] D. Park, M. Jung, and J. Cho, "Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable iot devices," *Simulation Modelling Practice and Theory*, vol. 77, pp. 379–389, 2017.

[6] T. Ge, Z. Mo, K. Wu, X. Zhang, and Y. Lu, "Rollbin: reducing code-size via loop rerolling at binary level," in *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2022, pp. 99–110.

[7] A. Inc, *Building a Universal macOS Binary*, Apple, 2025, https://developer.apple.com/documentation/apple-silicon/building-a-universal-macos-binary.

[8] S. E. Coder, *Simulink Embedded Coder Documentation*, MathWorks, 2025, https://www.mathworks.com/solutions/embedded-code-generation.html.

[9] G. Team, *GNU Compiler Collection*, 2025, https://gcc.gnu.org/.

[10] C. Community, *Clang C Fanguage Family Frontend for LLVM*, 2025, https://clang.llvm.org/.

[11] N. G. Obbink, I. Malavolta, G. L. Scoccia, and P. Lago, "An extensible approach for taming the challenges of javascript dead code elimination," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 291–401.

[12] Infineon, *32-bit AURIX™ TriCore™ Microcontroller*, 2025, https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/.

[13] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models," 2025. [Online]. Available: https://arxiv.org/abs/2303.18223

[14] H. Lin, M. Maas, M. Roquemore, A. Hasanzadeh, F. Lewis, Y. Simonson, T.-W. Yang, A. Yazdanbakhsh, D. Altinbüken, F. Papa, M. N. Edmonds, A. Patil, D. Schwarz, S. Chandra, C. Kennelly, M. Hashemi, and P. Ranganathan, "Eco: An llm-driven efficient code optimizer for warehouse scale computers," 2025. [Online]. Available: https://arxiv.org/abs/2503.15669

[15] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1592–1604. [Online]. Available: https://doi.org/10.1145/3650212.3680384

[16] X. Ren, Q. Zang, and Z. Guo, "Simugen: Multi-modal agentic framework for constructing block diagram-based simulation models," 2025. [Online]. Available: https://arxiv.org/abs/2506.15695

[17] Z. Xu, S. Jain, and M. Kankanhalli, "Hallucination is inevitable: An innate limitation of large language models," 2025. [Online]. Available: https://arxiv.org/abs/2401.11817

[18] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "Slnet: a redistributable corpus of 3rd-party simulink models," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 237–241. [Online]. Available: https://doi.org/10.1145/3524842.3528001

[19] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, "Developing applications using model-driven design environments," *Computer*, vol. 39, no. 2, pp. 33–40, 2006.

[20] Y. Jiang, H. Zhang, H. Zhang, X. Zhao, H. Liu, C. Sun, X. Song, M. Gu, and J. Sun, "Tsmart-galsblock: A toolkit for modeling, validation, and synthesis of multi-clocked embedded systems," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 711–714.

[21] Y. Jiang, H. Liu, H. Song, H. Kong, R. Wang, Y. Guan, and L. Sha, "Safety-assured model-driven design of the multifunction vehicle bus controller," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 10, pp. 3320–3333, 2018.

[22] MathWorks, *MATLAB Documentation*, 2025, https://www.mathworks.cn/products/matlab.html.

[23] J. Friedman, "Matlab/Simulink for automotive systems design," in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 1. IEEE, 2006, pp. 1–2.

[24] L. Wang, *PID control system design and automatic tuning using MATLAB/Simulink*. John Wiley & Sons, 2020.

[25] S. Palli, A. Duppala, R. C. Sharma, and L. Rao, "Dynamic simulation of automotive vehicle suspension using MATLAB Simulink," *Int J Veh Struct Syst*, vol. 14, 2022.

[26] OpenAI, *Introducing ChatGPT*, 2025, https://openai.com/index/chatgpt/.

[27] Cursor, *The AI Code Editor*, 2025, https://cursor.com/.

[28] S. Quan, J. Yang, B. Yu, B. Zheng, D. Liu, A. Yang, X. Ren, B. Gao, Y. Miao, Y. Feng, Z. Wang, J. Yang, Z. Cui, Y. Fan, Y. Zhang, B. Hui, and J. Lin, "Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings," 2025. [Online]. Available: https://arxiv.org/abs/2501.01257

[29] H. Langley, *Sundar Pichai says AI is making Google engineers 10% more productive*, 2025, https://www.businessinsider.com/ai-google-engineers-coding-productive-sundar-pichai-alphabet-2025-6/.

[30] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker: (competition contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 389–391.

[31] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[32] Z. Yu, Z. Su, Y. Yang, J. Liang, Y. Jiang, A. Cui, W. Chang, and R. Wang, "Mercury: Instruction pipeline aware code generation for Simulink models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4504–4515, 2022.

[33] Z. Yu, Y. Yang, Z. Su, R. Wang, Y. Tao, and Y. Jiang, "Knight: Optimizing code generation for simulink models with loop reshaping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 2, pp. 444–457, 2025.

[34] L. perf wiki Contributors, *perf: Linux profiling with performance counters*, 2025, https://perfwiki.github.io/main/.

[35] C. Ptolemaeus, *System design, modeling, and simulation: using Ptolemy II*. Ptolemy. org Berkeley, 2014, vol. 1.

[36] G. Berry, "Scade: Synchronous design and validation of embedded control software," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 2007, pp. 19–33.

[37] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 85–94. [Online]. Available: https://doi.org/10.1145/2597809.2597811

[38] LLVM, *MergeFunctions pass, how it works*, 2025, https://llvm.org/docs/MergeFunctions.html.

[39] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, K. Hazelwood, and H. Leather, "Hyfm: function merging for free," in *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 110–121. [Online]. Available: https://doi.org/10.1145/3461648.3463852

[40] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve, and H. Leather, "Large language models for compiler optimization," 2023. [Online]. Available: https://arxiv.org/abs/2309.07062

[41] Z. Chen, S. Fang, and M. Monperrus, "Supersonic: Learning to generate source code optimizations in c/c++," *IEEE Transactions on Software Engineering*, vol. 50, no. 11, p. 2849–2864, Nov. 2024. [Online]. Available: http://dx.doi.org/10.1109/TSE.2024.3423769

[42] S. Garg, R. Z. Moghaddam, and N. Sundaresan, "Rapgen: An approach for fixing code inefficiencies in zero-shot," 2025. [Online]. Available: https://arxiv.org/abs/2306.17077



Zehong Yu received the B.S. degree in software engineering from Southeast University in 2021, and M.S.E. degree in software engineering from Tsinghua University in 2024. Currently, he is pursuing a Ph.D. degree in software engineering at Tsinghua University, Beijing, China. His research interests are in the areas of model driven development and embedded software engineering.



Zhuo Su received the B.S. degree in software engineering from Northeastern University, Shenyang, China, in 2018, and the Ph.D. degree in software engineering from Tsinghua University, Beijing, China, in 2023. He currently works as an associate professor at Beihang University, Beijing, China. His research interests are in the areas of model driven development and software security.



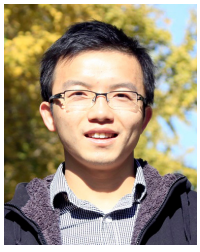
Yixiao Yang received the B.S. degree in software engineering from Nanjing University, Nanjing, China, in 2014. He received the Ph.D. degree in software engineering from Tsinghua University, Beijing, China. He is currently working as a assistant researcher in the College of Information Engineering, Capital Normal University, Beijing, China. His research interests include code completion, test case generation, model driven design and their applications to industry.



Haowei Qiu received the B.S. degree in software engineering from Tsinghua University, Beijing, China, in 2025. He is currently pursuing the M.S.E. degree in software engineering at the School of Software, Tsinghua University, Beijing, China. His research interests are in the area of model driven development.



Rui Wang Rui Wang received the B.S. degree in computer science from Xi'an Jiaotong University, Xi'an, China, in 2004, and the Ph.D. degree in computer science from Tsinghua University, Beijing, China, in 2012. She is currently a professor with the College of Information Engineering, Capital Normal University, Beijing, China. Her research interests include formal verification and their applications in embedded systems.



Yu Jiang received the B.S. degree in software engineering from Beijing University of Posts and Telecommunications in 2010, and the PhD degree in computer science from Tsinghua University in 2015. He was a Postdoc researcher with the Department of Computer Science, University of Illinois at Urbana Champaign, Champaign, IL, USA, in 2016, and is now an associate professor in Tsinghua University. His research interests include domain specific modeling, formal computation model, formal verification and their applications in embedded systems.