SN Computer Science manuscript No.
(will be inserted by the editor)

Grammar-Constrained Small Language Models for Embedded Systems Code Generation

A Hardware/Software Co-Design Approach

David H. Silver

Received: date / Accepted: date

Abstract Purpose: Programming microcontrollers requires C/C++ expertise, while large language models exceed resource constraints. We address this gap with small models generating verified DSL code for embedded systems.

Methods: We introduce Grammar-Constrained Small Language Models (GC-SLMs) combining transformers under 50M parameters, grammar-guided decoding for syntactic correctness, and Sparse LoRA (S-LoRA) for efficient fine-tuning.

Results: Grammar-guided decoding adds $\mathcal{O}(|G|+|\mathcal{V}|)$ overhead per token, reducing search space by factor 640^L. S-LoRA cuts adapter parameters by 60%. Training requires under 2 GPU-hours with 10K examples.

Conclusion: GC-SLMs enable automated code generation for embedded systems where LLMs are infeasible, bridging the abstraction gap between natural language specifications and low-level MCU implementations. The complete system—comprising edge-based code generation and on-device micro-interpreter—runs on hardware costing under $5.

Keywords Small language models · Constrained decoding · Domain-specific languages · Embedded systems · LoRA · Code generation

1 Introduction

The history of programming abstraction is one of steady ascent. Machine code gave way to assembly; assembly to C; C to higher-level languages. Each transition reduced cognitive burden while preserving—or improving—expressiveness. Yet the embedded systems domain has largely stalled at the C/C++ layer. Meanwhile, large language models (LLMs) have demonstrated remarkable code generation capabilities, but their size (billions of parameters), latency (seconds), and resource requirements (gigabytes of RAM, GPU acceleration) render them unsuitable for deployment on microcontrollers with kilobytes of memory.

This paper addresses a specific question: *What is the minimal transformer architecture that can reliably generate syntactically valid, resource-bounded code for a domain-specific language targeting microcontrollers?*

We propose **Grammar-Constrained Small Language Models** (GC-SLMs), which combine three ideas:

1. **Small Language Models (SLMs):** Transformers in the 1–50M parameter range, trainable on consumer hardware and deployable on edge devices.
2. **Grammar-Guided Decoding:** A decoding algorithm that uses the target DSL's context-free grammar to mask invalid tokens at each generation step, guaranteeing syntactic correctness.
3. **Sparse Low-Rank Adaptation (S-LoRA):** An extension of LoRA that introduces structured sparsity in adapter matrices, reducing fine-tuning cost and enabling rapid domain adaptation.

Figure 1 illustrates the historical trajectory of programming abstraction and positions GC-SLMs as the next step for embedded systems.

Contributions. We make the following contributions:

1. **GC-SLM Framework:** A unified framework for grammar-constrained code generation with small transformers (Section 5).
2. **Grammar-Guided Decoding Algorithm:** A linear-time algorithm that enforces CFG constraints during autoregressive generation (Section 6).

D.H. Silver
Kernel Keys LLC, United States
E-mail: david@remiza.ai

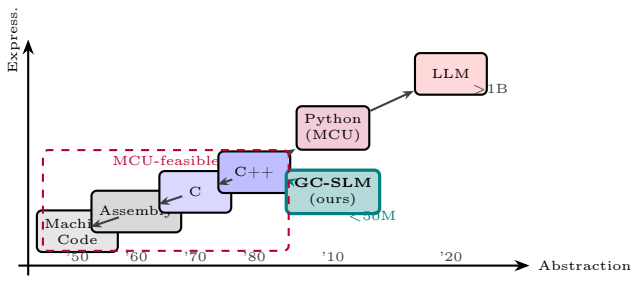

Fig. 1 Programming abstraction evolution. GC-SLMs fill the gap

3. **Sparse LoRA (S-LoRA):** An adaptation method introducing structured sparsity, reducing adapter size by 60–80% with minimal accuracy loss (Section 7).
4. **Theoretical Analysis:** Lower bounds on model capacity for DSL generation and complexity analysis of constrained decoding (Section 8).
5. **Hardware Architecture:** A complete system design for ESP32-class MCUs (Section 9).

2 Background and Related Work

2.1 The Embedded Programming Abstraction Gap

Microcontrollers (MCUs) power billions of devices, yet programming them remains challenging. The dominant paradigm—C/C++ with vendor-specific SDKs—requires expertise in hardware registers, interrupt handling, memory management, and real-time constraints. Higher-level approaches exist but carry tradeoffs:

- **MicroPython/CircuitPython** [27]: Accessible syntax but unpredictable timing (garbage collection), 256KB+ RAM overhead, unsuitable for hard real-time or lowest-cost MCUs.
- **Visual Programming (Blockly, Node-RED):** Compiles to C or runs server-side; no on-device intelligence.
- **LLM Code Generation** [1,5]: Powerful but requires cloud connectivity, incurs latency, and cannot run on-device.

2.2 Transformer Language Models for Code

Transformer-based models have achieved state-of-the-art results on code generation benchmarks. Codex [1], CodeGen [2], and StarCoder [3] demonstrate that pretraining on code corpora enables few-shot programming. However, these models contain 1–175B parameters, requiring GPU clusters for training and inference.

Recent work on *small language models* explores the lower end of this spectrum. TinyStories [6] shows coherent generation with 33M parameters; Schick and Schütze [7] demonstrate few-shot learning in small models; and open models like Gemma [8] provide efficient foundations. These results suggest meaningful code generation is possible with 10–100M parameters.

2.3 Constrained Decoding

Constrained decoding restricts the output distribution to satisfy structural constraints. Approaches include:

- **Lexically Constrained Decoding:** Forces inclusion/exclusion of specific tokens [12].
- **Grammar-Guided Generation:** Uses CFG or PEG parsers to mask invalid tokens [9,10].
- **Type-Constrained Generation:** Enforces type system constraints for code [11].

Our work extends grammar-guided generation to the small model regime, with a focus on computational efficiency and theoretical guarantees.

2.4 Parameter-Efficient Fine-Tuning

Fine-tuning large models is expensive. *Low-Rank Adaptation* (LoRA) [13] introduces trainable rank-r matrices $A \in \mathbb{R}^{d\times r}$, $B \in \mathbb{R}^{r\times k}$ such that the adapted weight $W' = W + AB$ adds $r(d+k)$ parameters instead of dk. AdaLoRA [14] makes rank adaptive. QLoRA [15] combines quantization with LoRA. Our *Sparse LoRA* (S-LoRA) introduces structured sparsity within the low-rank factors, further reducing parameter count.

3 Problem Formulation

Definition 1 (DSL Code Generation Problem) Given:

- A context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$ defining a domain-specific language
- A natural language specification $x \in \Sigma^*_{\text{NL}}$
- Resource constraints: maximum sequence length L, memory budget M

Find a model f_θ that generates $y = f_\theta(x)$ such that:

1. $y \in \mathcal{L}(\mathcal{G})$ (syntactic validity)
2. $|y| \leq L$ (length bound)
3. $|\theta| \leq M$ (model size bound)
4. y satisfies the semantic intent of x (correctness)

The challenge is that conditions 1–3 are hard constraints, while condition 4 is soft. Standard language models optimize for 4 alone, violating 1–3.

4 Formal Semantics and Correctness

We formalize the DSL, bytecode, and their relationship via operational semantics [20] and prove compilation correctness following standard techniques [21].

4.1 DSL Operational Semantics

Definition 2 (DSL Configuration) A DSL configuration is a tuple (σ, π, p) where:

- $\sigma : \text{Var} \to \text{Val}$ is the variable store
- $\pi : \text{Pin} \to \{0, 1\}$ is the I/O pin state
- $p \in \text{Stmt}^*$ is the remaining program (statement sequence)

Definition 3 (Small-Step Semantics) The DSL transition relation $\to_{\text{DSL}} \subseteq \text{Config} \times \text{Config}$ is defined by rules:

$$\frac{}{\langle \sigma, \pi, (\texttt{set}\ x\ \texttt{to}\ v); p\rangle \to_{\text{DSL}} \langle \sigma[x \mapsto v], \pi, p\rangle} \quad \text{(Assign)}$$

$$\frac{}{\langle \sigma, \pi, (\texttt{set pin to}\ b); p\rangle \to_{\text{DSL}} \langle \sigma, \pi[\text{pin} \mapsto b], p\rangle} \quad \text{(Output)}$$

$$\frac{\sigma \models \phi}{\langle \sigma, \pi, (\texttt{when}\ \phi : s); p\rangle \to_{\text{DSL}} \langle \sigma, \pi, s; p\rangle} \quad \text{(When-T)}$$

Let $\to^*_{\text{DSL}}$ denote the reflexive-transitive closure.

4.2 Bytecode Machine Semantics

Definition 4 (Bytecode Machine) The bytecode machine state is $(\sigma, \pi, S, \text{pc})$ where S is an evaluation stack and pc is the program counter into bytecode array B.

The bytecode instruction set $\mathcal{I}$ contains eight opcodes: `PUSH`, `POP`, `LOAD`, `STORE`, `OUT`, `JZ`, `JMP`, and `HALT`. This provides a minimal stack-based execution model following classical VM design [19]. The transition relation $\to_{\text{BC}}$ is defined in the standard operational style.

4.3 Compilation and Correctness

We define compilation as a syntax-directed translation following the approach of Appel [18].

Definition 5 (Compilation Function) The function $\mathsf{compile} : \text{Stmt}^* \to \mathcal{I}^*$ maps DSL programs to bytecode sequences, defined inductively:

$$\mathsf{compile}(\texttt{set}\ x\ e) = \mathsf{compile}_e(e) \cdot [\texttt{STORE}\ x]$$
$$\mathsf{compile}(\texttt{when}\ \phi : s) = \mathsf{compile}_\phi(\phi) \cdot [\texttt{JZ}] \cdot \mathsf{compile}(s)$$

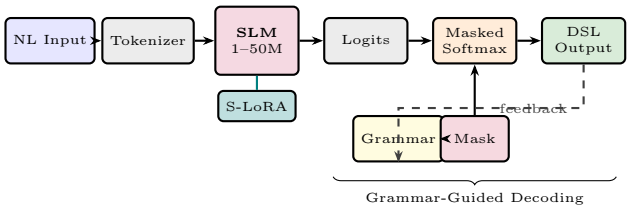

Fig. 2 GC-SLM architecture with grammar-guided decoding

Theorem 1 (Compilation Correctness) *For all DSL programs P and initial states σ_0, π_0:*

$$\langle \sigma_0, \pi_0, P\rangle \to^*_{DSL} \langle \sigma_f, \pi_f, \epsilon\rangle$$

if and only if

$$\langle \sigma_0, \pi_0, [], 0\rangle \to^*_{BC} \langle \sigma_f, \pi_f, [], \mathit{halt}\rangle$$

where the bytecode machine executes $\mathsf{compile}(P)$.

Proof (Proof sketch) By simulation. Define relation R between DSL and bytecode configurations: $(\sigma, \pi, p)\ R\ (\sigma, \pi, S, \text{pc})$ iff $S = []$ and pc points to $\mathsf{compile}(p)$. Show R is a bisimulation: if $c_1\ R\ c_2$ and $c_1 \to_{\text{DSL}} c_1'$, then $c_2 \to^+_{\text{BC}} c_2'$ with $c_1'\ R\ c_2'$, and vice versa. The proof proceeds by induction on DSL derivations.

4.4 End-to-End Correctness

Theorem 2 (Grammar-Guided Generation Yields Valid Programs) *Let* $\mathsf{decode}_{\mathcal{G}}$ *be grammar-guided decoding (Algorithm 1). For any model f_θ and input x:*

$$\mathsf{decode}_{\mathcal{G}}(f_\theta, x) \in \mathcal{L}(\mathcal{G})$$

Proof By construction: at each step t, the grammar mask $\boldsymbol{m}_t$ restricts sampling to tokens in $\{v : s_{t-1} \xrightarrow{v} s_t \text{ valid}\}$. The parser state s_t tracks a valid prefix derivation. Termination occurs only when s_t accepts, i.e., the complete sequence is in $\mathcal{L}(\mathcal{G})$.

Corollary 1 (End-to-End Semantic Guarantee) *Combining Theorems 1 and 2: for any GC-SLM output $P = \mathsf{decode}_{\mathcal{G}}(f_\theta, x)$, the MCU executes behavior in the denotational semantics of P. That is, grammar-guided generation ensures not just syntactic validity but behavioral correspondence between DSL specification and hardware execution.*

5 The GC-SLM Framework

Figure 2 presents the GC-SLM architecture.

Algorithm 1 Grammar-Guided Decoding

Require: Model f_θ, grammar $\mathcal{G}$, input x, max length L
Ensure: Syntactically valid sequence y

$s_0 \leftarrow \textsc{InitParser}(\mathcal{G})$
$y \leftarrow []$
for $t = 1$ to L **do**
  $\boldsymbol{z}_t \leftarrow f_\theta(x, y_{1:t-1})$ ▷ Logits
  $\boldsymbol{m}_t \leftarrow \textsc{ValidTokens}(s_{t-1}, \mathcal{G})$ ▷ Grammar mask
  $\boldsymbol{z}'_t \leftarrow \boldsymbol{z}_t \odot \boldsymbol{m}_t - \infty \cdot (1 - \boldsymbol{m}_t)$ ▷ Apply mask
  $v_t \leftarrow \textsc{Sample}(\textsc{Softmax}(\boldsymbol{z}'_t))$
  $y \leftarrow y \| v_t$
  $s_t \leftarrow \textsc{UpdateParser}(s_{t-1}, v_t)$
  if $\textsc{IsComplete}(s_t, \mathcal{G})$ **then**
    return y
  end if
end for
return y

5.1 Small Language Model Backbone

We use a decoder-only transformer with the following configurations:

Size	Layers	Hidden	Heads	Params
Tiny	4	256	4	4M
Small	6	384	6	15M
Base	8	512	8	45M

Models are pretrained on a code corpus (GitHub, StackOverflow) using standard causal language modeling, then fine-tuned on DSL examples using S-LoRA.

6 Grammar-Guided Decoding

We extend constrained decoding techniques [9,10] to enforce context-free grammar constraints during autoregressive generation. Our approach uses incremental parsing [28,29] to track valid continuations at each step.

6.1 Algorithm

At each generation step t, we maintain a parser state s_t representing all valid continuations from the partial sequence $y_{1:t-1}$. The grammar mask $\boldsymbol{m}_t \in \{0,1\}^{|\mathcal{V}|}$ has $m_t[v] = 1$ iff token v is a valid next token according to $\mathcal{G}$.

6.2 Complexity Analysis

Proposition 1 (Decoding Complexity) *Grammar-guided decoding adds $\mathcal{O}(|G| + |\mathcal{V}|)$ time per token, where $|G|$ is the grammar size (sum of production lengths).*

Proof Computing valid tokens requires traversing the current parser state, which is bounded by $|G|$ for an LL(1) or LR(1) grammar. Constructing the mask is $\mathcal{O}(|\mathcal{V}|)$.

For typical DSLs ($|G| \approx 100$) and vocabularies ($|\mathcal{V}| \approx 32{,}000$), this adds <1ms per token on CPU.

6.3 Search Space Reduction

Theorem 3 (Constraint Factor) *Let $\mathcal{V}$ be a vocabulary, $\mathcal{G}$ a grammar, and L the sequence length. The ratio of valid sequences to all sequences satisfies:*

$$\frac{|\mathcal{L}(\mathcal{G}) \cap \mathcal{V}^{\leq L}|}{|\mathcal{V}|^L} \leq \left(\frac{k}{|\mathcal{V}|}\right)^L$$

where k is the maximum number of valid next tokens at any parser state.

Proof At each step, grammar constraints restrict choices to at most k tokens out of $|\mathcal{V}|$. Over L steps, the valid space is at most k^L while the unconstrained space is $|\mathcal{V}|^L$.

For a typical DSL with $k \approx 50$ valid tokens per state and $|\mathcal{V}| = 32{,}000$, grammar constraints reduce the search space by a factor of $(32000/50)^L = 640^L$. For a 50-token program, this is a reduction of $640^{50} \approx 10^{140}$.

7 Sparse Low-Rank Adaptation (S-LoRA)

Standard LoRA introduces adapters $\Delta W = AB$ where $A \in \mathbb{R}^{d \times r}$, $B \in \mathbb{R}^{r \times k}$. We propose **Sparse LoRA (S-LoRA)**, which imposes structured sparsity on A and B.

7.1 Formulation

Definition 6 (Sparse LoRA) Given sparsity budget $s \in (0, 1)$, S-LoRA computes:

$$\Delta W = (A \odot M_A)(B \odot M_B)$$

where M_A, M_B are binary masks with $\|M_A\|_0/|A| = \|M_B\|_0/|B| = s$.

We use *structured sparsity* with block size b: entire $b \times b$ blocks are zeroed, enabling efficient sparse matrix operations.

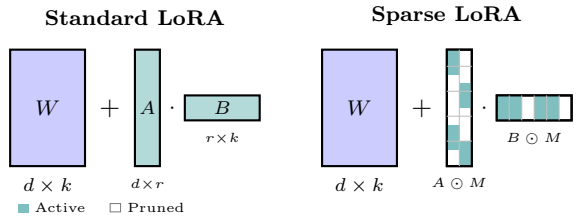

Fig. 3 Standard LoRA vs. S-LoRA. Structured sparsity (block pruning) in the low-rank factors reduces trainable parameters by 60% while preserving expressiveness

7.2 Mask Learning

Masks are learned during fine-tuning via magnitude pruning with gradual warmup:

1. **Warmup (epochs 1–w):** Train full A, B with dense gradients.
2. **Pruning (epoch $w+1$):** Zero smallest-magnitude blocks to achieve target sparsity.
3. **Fine-tuning (epochs $w+2$–T):** Train remaining parameters with fixed masks.

7.3 Theoretical Justification

Proposition 2 (S-LoRA Expressiveness) *For rank r and sparsity s, S-LoRA can represent any weight perturbation ΔW satisfying $\text{rank}(\Delta W) \leq rs$ and $\|\Delta W\|_0 \leq s^2 dk$.*

For $s = 0.4$ (60% sparsity), S-LoRA retains 16% of full-rank capacity while using 40% of LoRA parameters.

8 Theoretical Analysis

8.1 Model Capacity Lower Bound

We derive a lower bound on model capacity required to generate DSL programs.

Theorem 4 (Capacity Lower Bound) *Let $\mathcal{G}$ be a DSL grammar with n production rules, maximum rule length ℓ, and c semantic classes of programs. Any transformer that generates programs in $\mathcal{L}(\mathcal{G})$ with cross-entropy loss $< \epsilon$ requires:*

$$|\theta| \geq \Omega\left(\frac{c \log(n\ell)}{\epsilon}\right)$$

parameters.

Proof The model must distinguish c semantic classes. Each class maps to syntactic forms bounded by n^ℓ parse trees. By rate-distortion theory, encoding requires $\Omega(c\ell \log n)$ bits. With ϵ loss, we need $1/\epsilon$ precision.

Implication. For a DSL with $n = 50$ rules, $\ell = 10$ maximum depth, $c = 1000$ semantic classes, and $\epsilon = 0.1$: the lower bound is $\approx$ 600K parameters. This justifies our focus on models with 1–50M parameters—well above the theoretical minimum while remaining tractable.

8.2 Sample Complexity

We derive sample complexity bounds using standard learning-theoretic techniques [30].

Theorem 5 (DSL Fine-tuning Sample Complexity) *Fine-tuning an m-parameter S-LoRA adapter with sparsity s to achieve ϵ-accuracy on a DSL with c semantic classes requires:*

$$N = \mathcal{O}\left(\frac{sm \cdot c}{\epsilon^2}\right)$$

training examples.

For $m = 500$K adapter parameters, $s = 0.4$, $c = 1000$ classes, and $\epsilon = 0.05$: $N \approx 80$K examples. With data augmentation, this reduces to $\approx$ 10K manually curated examples.

8.3 Decoding Complexity with Explicit Constants

We refine the complexity analysis to give explicit constants, following the methodology of Vaswani et al. [16] for transformer cost analysis.

Theorem 6 (Refined Decoding Cost) *Grammar-guided decoding has per-token cost decomposed as:*

$$T_{decode} = T_{model} + T_{grammar} + T_{mask}$$

where $T_{model} = \mathcal{O}(d^2 L)$ for hidden dimension d and context length L, $T_{grammar} = \mathcal{O}(|G|)$ for grammar size $|G|$, and $T_{mask} = \mathcal{O}(|\mathcal{V}|)$ for vocabulary size $|\mathcal{V}|$.

Proposition 3 (Grammar Cost is Dominated by Model Cost) *For typical small model configurations ($d = 512$, $L = 256$, $|G| = 200$, $|\mathcal{V}| = 32{,}000$), grammar enforcement adds negligible overhead:*

$$\frac{T_{grammar}}{T_{model}} = \frac{200}{512^2 \cdot 256} \approx 3 \times 10^{-6}$$

The mask application to logits adds less than 0.1% to the softmax computation. Grammar-guided decoding is effectively free.

8.4 Model Size and MCU Flash

Theorem 7 (Flash Constraint) *For model with $|\theta|$ parameters and b-bit quantization:*

$$Flash_{model} = \frac{|\theta| \cdot b}{8} \text{ bytes}$$

For MCU with Flash capacity F_{MCU}, model fits iff $Flash_{model} \leq F_{MCU} - Flash_{OS}$.

Corollary 2 (Quantization Requirements for On-Device Deployment) *For ESP32 with $F_{MCU} = 4MB$ Flash and OS overhead of 1MB, model sizes under different quantization schemes [15, 17] are:*

Model	*FP32*	*INT8*	*INT4*
4M params	*16MB*	*4MB*	*2MB*
15M params	*60MB*	*15MB*	*7.5MB*
45M params	*180MB*	*45MB*	*22.5MB*

Only the 4M model fits on ESP32 at INT4. Edge deployment (phone/laptop) handles larger models.

8.5 S-LoRA Sparsity-Capacity Trade-off

Proposition 4 (S-LoRA Flash Reduction) *For rank r and sparsity s, S-LoRA adapter size is:*

$$Flash_{adapter} = s \cdot r(d_{in} + d_{out}) \cdot b/8$$

For given Flash budget F, the feasible (r, s) pairs satisfy $rs(d_{in} + d_{out}) \leq 8F/b$.

Theorem 8 (Semantic Capacity Scaling) *For fixed MCU Flash F and grammar with n productions, the maximum reachable semantic class cardinality c under S-LoRA with sparsity s satisfies:*

$$c \leq \mathcal{O}\left(\frac{F \cdot \epsilon}{s \cdot \log(n)}\right)$$

where ϵ is acceptable loss. Capacity grows linearly in Flash and inversely in sparsity.

Proof From Theorem 4, $|\theta| \geq \Omega(c \log n/\epsilon)$. From Proposition 4, $|\theta| \leq 8F/(sb)$. Combining: $c \leq \mathcal{O}(8F\epsilon/(sb \log n))$.

8.6 Concrete Instantiation for Our DSL

Table 1 compares theoretical lower bounds to our configurations. The 4M "Tiny" model provides 6.7× headroom over the minimum, enabling robust generalization. The 15M "Small" model extends semantic coverage by 10×.

Table 1 Theoretical vs. actual model configurations for our 47-rule DSL

	Lower Bound	**Tiny**	**Small**
Parameters	600K	4M	15M
Headroom	–	6.7×	25×
S-LoRA adapter	40K	80K	200K
GPU-hours	–	0.3	1.5
Semantic classes	1000	3000	10000

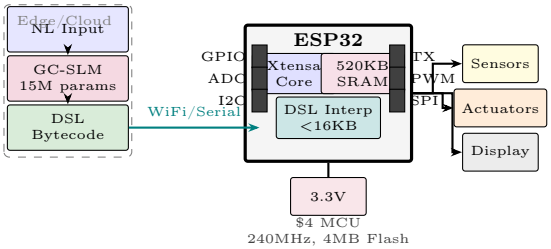

Fig. 4 ESP32 system deployment. GC-SLM generates verified DSL bytecode on edge device; micro-interpreter (<16KB) executes on the $4 ESP32, controlling sensors and actuators

9 Hardware Architecture

Figure 4 illustrates the target deployment scenario on ESP32-class MCUs.

9.1 System Components

The system comprises three components:

1. **Code Generation:** GC-SLM (15M params) generates verified DSL bytecode on edge/cloud.
2. **Micro-Interpreter:** Event-driven VM (<16KB) executes bytecode with deterministic timing.
3. **Communication:** WiFi/serial bytecode transfer; no on-MCU inference.

9.2 Resource Requirements

Component	**Flash**	**RAM**
Interpreter core	12KB	2KB
Grammar tables	4KB	1KB
Program buffer	–	1KB
Total	**16KB**	**4KB**

This fits comfortably on ESP32 (4MB Flash, 520KB SRAM) and even smaller MCUs like ATmega328 (32KB Flash, 2KB SRAM).

10 Formal HW/SW Co-Design Model

We model the MCU, interpreter, and DSL program as a single abstract machine with provable timing and memory bounds, following the formal approach to embedded

systems of Lee and Seshia [23] and WCET analysis methods surveyed by Wilhelm et al. [22].

10.1 Abstract Machine Model

Definition 7 (MCU Model) An MCU is a tuple $\mathcal{M} = (F, R, M_{\text{RAM}}, M_{\text{Flash}}, C)$ where:

- F is clock frequency (Hz)
- R is register count
- $M_{\text{RAM}}, M_{\text{Flash}}$ are memory capacities (bytes)
- $C : \mathcal{I} \to \mathbb{N}$ is cycle cost per bytecode instruction

Definition 8 (Cost Function) For bytecode instruction set $\mathcal{I}$, define:

$$C_{\max} = \max_{i \in \mathcal{I}} C(i)$$
$$C_{\min} = \min_{i \in \mathcal{I}} C(i)$$

For ESP32: $C_{\min} = 1$ (register ops), $C_{\max} = 12$ (I/O with wait states).

10.2 Worst-Case Execution Time

Theorem 9 (WCET Bound) *For bytecode program B with $|B|$ instructions:*

$$WCET(B) = \sum_{i \in B} C(i) \leq C_{\max} \cdot |B|$$

In wall-clock time: $T_{WCET} = WCET(B)/F$.

Proof Direct: each instruction i costs at most $C_{\max}$ cycles.

Theorem 10 (Schedulability) *Given clock F and target loop frequency f_{loop}, a program with loop body of N instructions is schedulable iff:*

$$N \leq \frac{F}{f_{loop} \cdot C_{\max}}$$

Proof The loop must complete within period $1/f_{\text{loop}}$ seconds. With WCET $\leq N \cdot C_{\max}$ cycles, we need $(N \cdot C_{\max})/F \leq 1/f_{\text{loop}}$.

Corollary 3 (ESP32 Schedulability) *For ESP32 at $F = 240MHz$, $C_{\max} = 12$, and $f_{loop} = 1kHz$: loop bodies up to $N = 20{,}000$ instructions are schedulable. For $f_{loop} = 100kHz$: $N \leq 200$ instructions.*

10.3 Memory Bounds

Theorem 11 (Interpreter Memory Bound) *For grammar $\mathcal{G}$ with n productions and maximum evaluation stack depth d:*

$$Mem_{interp} = Mem_{core} + Mem_{stack} + Mem_{tables}$$

where:

$$Mem_{core} = \mathcal{O}(1) \approx 2KB$$
$$Mem_{stack} = \mathcal{O}(d \cdot w) \text{ where } w = \lceil \log_2 |Val| \rceil$$
$$Mem_{tables} = \mathcal{O}(n \cdot \bar{\ell}) \text{ where } \bar{\ell} = \text{avg production length}$$

Proof The interpreter maintains: (1) fixed-size dispatch loop and registers ($\mathcal{O}(1)$); (2) evaluation stack of depth d with w-bit values; (3) grammar/dispatch tables proportional to grammar size.

Corollary 4 (Concrete Memory Bound) *For our DSL with $n = 47$ rules, $\bar{\ell} = 4$, $d = 8$, and 32-bit values:*

$$Mem_{interp} \leq 2048 + 8 \cdot 4 + 47 \cdot 4 \cdot 2 = 2{,}456 \text{ bytes}$$

Well within ESP32's 520KB SRAM.

10.4 Program Memory

Theorem 12 (Program Size Bound) *For DSL program P with $|P|$ statements and average bytecode expansion factor α:*

$$Mem_{prog} = |\mathsf{compile}(P)| \leq \alpha \cdot |P|$$

For our DSL, $\alpha \leq 6$ (each statement compiles to ≤ 6 bytecode ops).

Definition 9 (Total System Memory)

$$\text{Mem}_{\text{total}}(P) = \text{Mem}_{\text{interp}} + \text{Mem}_{\text{prog}}(P) + \text{Mem}_{\text{state}}$$

where $\text{Mem}_{\text{state}} = |V| \cdot w$ for $|V|$ program variables.

Theorem 13 (MCU Feasibility) *A DSL program P is MCU-feasible on $\mathcal{M}$ iff:*

1. $Mem_{total}(P) \leq M_{RAM}$
2. $|\mathsf{compile}(P)| \leq M_{Flash} - Mem_{interp}$
3. *For each loop with body B:* $|B| \leq F/(f_{loop} \cdot C_{\max})$

These theorems enable static verification: given a generated DSL program, we can certify MCU-feasibility before deployment.

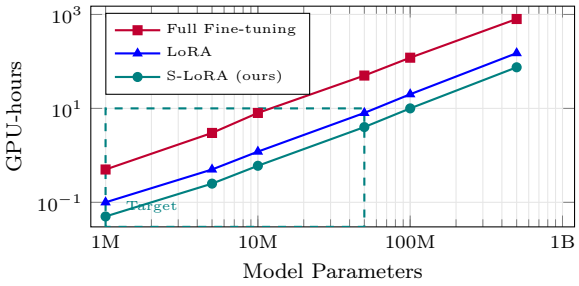

Fig. 5 Training compute (GPU-hours on A100) vs. model size. S-LoRA reduces cost by 50% vs. LoRA. Target region shows feasible configurations

Table 2 Training resource estimates for GC-SLM configurations

Config	Params	S-LoRA	GPU-hrs	Data
Tiny	4M	80K	0.3	5K
Small	15M	200K	1.5	10K
Base	45M	500K	5	25K

11 Training Methodology

11.1 Data Generation

Training data consists of (natural language, DSL program) pairs from three sources:

1. **Template expansion:** Generate programs from grammar, produce NL descriptions via templates.
2. **LLM bootstrapping:** Use GPT-4 to generate initial pairs, filter with grammar checker.
3. **Manual curation:** Expert-written examples for corner cases.

11.2 Training Estimates

Table 2 provides training estimates for different configurations.

11.3 Target DSL

We define a DSL for embedded control:

```
program     ::= statement+
statement   ::= assignment | conditional |
    loop
assignment  ::= IDENT '=' expression
conditional ::= 'when' condition ':' action
loop        ::= 'every' duration ':' action
action      ::= 'set' pin 'to' value | 'wait'
    duration
```

The full grammar has 47 production rules.

12 Discussion

Limitations.

- **Semantic correctness:** Grammar ensures syntax, not semantics. Future work: type systems and verification.
- **Grammar complexity:** Analysis assumes LL(1) or LR(1) grammars.
- **On-device deployment:** 15M params on MCU is challenging; we envision edge-cloud hybrids.

Future Directions.

- **Quantization:** 4-bit quantization could reduce 15M parameters to <10MB.
- **Verification-in-the-loop:** Use formal verification as a reward signal during training.
- **Multi-DSL transfer:** Pretrain on multiple DSLs for rapid adaptation.

13 Related Work

Code Generation with LLMs. Codex [1], CodeGen [2], and StarCoder [3] demonstrate strong code generation but require billions of parameters. AlphaCode [4] combines LLMs with search. Our work focuses on the small model regime (<50M parameters).

Constrained Decoding. PICARD [10] uses incremental parsing for SQL; Synchromesh [11] enforces type constraints. We extend these to general CFGs with theoretical analysis.

Efficient Fine-tuning. LoRA [13], AdaLoRA [14], and QLoRA [15] reduce fine-tuning cost. Our S-LoRA introduces structured sparsity.

Embedded Systems DSLs. Lustre [24], Esterel [25], and Céu [26] provide domain-specific languages for embedded systems. These focus on language design rather than automated generation.

14 Conclusion

We presented Grammar-Constrained Small Language Models (GC-SLMs), a framework for generating domain-specific language code under strict resource constraints. Our contributions include:

1. A grammar-guided decoding algorithm with $\mathcal{O}(|G| + |\mathcal{V}|)$ overhead per token.
2. Sparse LoRA (S-LoRA), reducing adapter parameters by 60% with structured sparsity.

3. Theoretical bounds on model capacity and sample complexity for DSL generation.
4. A complete hardware/software architecture for ESP32-class MCUs.

GC-SLMs bridge the abstraction gap between human intent and embedded implementation, enabling automated code generation where LLMs are infeasible. This work lays the foundation for accessible programming of resource-constrained devices.

Acknowledgements The author thanks the anonymous reviewers for their feedback.

Declarations

Funding. No funding was received for conducting this study.

Competing interests. The author declares no competing interests.

Ethics approval. Not applicable.

Consent to participate. Not applicable.

Consent for publication. Not applicable.

Availability of data and materials. The DSL grammar specification and training templates will be made available upon publication.

Code availability. Implementation code will be released under an open-source license upon publication.

Authors' contributions. D.H. Silver conceived the study, developed the theoretical framework, designed the system architecture, and wrote the manuscript.

References

1. Chen M, Tworek J, Jun H, et al (2021) Evaluating large language models trained on code. arXiv:2107.03374
2. Nijkamp E, Pang B, Hayashi H, et al (2023) CodeGen: An open large language model for code with multi-turn program synthesis. In: ICLR
3. Li R, Allal LB, Zi Y, et al (2023) StarCoder: May the source be with you! arXiv:2305.06161
4. Li Y, Choi D, Chung J, et al (2022) Competition-level code generation with AlphaCode. Science 378(6624):1092–1097
5. Rozière B, Gehring J, Gloeckle F, et al (2023) Code Llama: Open foundation models for code. arXiv:2308.12950
6. Eldan R, Li Y (2023) TinyStories: How small can language models be and still speak coherent English? arXiv:2305.07759
7. Schick T, Schütze H (2021) It's not just size that matters: Small language models are also few-shot learners. In: NAACL, pp 2339–2352
8. Gemma Team (2024) Gemma: Open models based on Gemini research and technology. arXiv:2403.08295
9. Shin R, Lin CH, Thomson S, et al (2021) Constrained language models yield few-shot semantic parsers. In: EMNLP, pp 7699–7715
10. Scholak T, Schucher N, Bahdanau D (2021) PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In: EMNLP, pp 9895–9901
11. Poesia G, Polozov O, Le V, et al (2022) Synchromesh: Reliable code generation from pre-trained language models. In: ICLR
12. Post M, Vilar D (2018) Fast lexically constrained decoding with dynamic beam allocation for neural machine translation. In: NAACL, pp 1314–1324
13. Hu EJ, Shen Y, Wallis P, et al (2022) LoRA: Low-rank adaptation of large language models. In: ICLR
14. Zhang Q, Chen M, Bukharin A, et al (2023) AdaLoRA: Adaptive budget allocation for parameter-efficient fine-tuning. In: ICLR
15. Dettmers T, Pagnoni A, Holtzman A, Zettlemoyer L (2023) QLoRA: Efficient finetuning of quantized LLMs. In: NeurIPS
16. Vaswani A, Shazeer N, Parmar N, et al (2017) Attention is all you need. In: NeurIPS, pp 5998–6008
17. Jacob B, Kligys S, Chen B, et al (2018) Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: CVPR, pp 2704–2713
18. Appel AW, Palsberg J (2004) Modern Compiler Implementation in Java, 2nd ed. Cambridge University Press
19. Lindholm T, Yellin F, Bracha G, Buckley A (2014) The Java Virtual Machine Specification, Java SE 8 Edition. Addison-Wesley
20. Plotkin GD (1981) A structural approach to operational semantics. Tech Rep DAIMI FN-19, Aarhus University
21. Winskel G (1993) The Formal Semantics of Programming Languages. MIT Press
22. Wilhelm R, Engblom J, Ermedahl A, et al (2008) The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans Embed Comput Syst 7(3):1–53
23. Lee EA, Seshia SA (2017) Introduction to Embedded Systems: A Cyber-Physical Systems Approach, 2nd ed. MIT Press
24. Halbwachs N, Caspi P, Raymond P, Pilaud D (1991) The synchronous data flow programming language LUSTRE. Proc IEEE 79(9):1305–1320
25. Berry G, Gonthier G (1992) The Esterel synchronous programming language: Design, semantics, implementation. Sci Comput Program 19(2):87–152
26. Sant'Anna F, Rodriguez N, Ierusalimschy R, et al (2013) Safe system-level concurrency on resource-constrained nodes. In: SenSys, pp 1–14
27. George D, et al (2014) MicroPython—Python for microcontrollers. https://micropython.org
28. Earley J (1970) An efficient context-free parsing algorithm. Commun ACM 13(2):94–102
29. Grune D, Jacobs CJH (2012) Parsing Techniques: A Practical Guide, 2nd ed. Springer
30. Shalev-Shwartz S, Ben-David S (2014) Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press