A Sampling-and-Filtering Framework with Offline Policy Exploration for Certified Loop Invariant Generation

Anonymous
Anonymous Institution
anonymous@example.com

Abstract

Automatically synthesizing loop invariants remains a central challenge in formal software verification. We present SAM2INV, a system built around three contributions. First, LoopFactory, a probabilistic DSL that generates structured numeric loop programs with controllable arithmetic complexity, nesting depth, and variable count, serves as the *task input generator*: it supplies the programs that the downstream pipeline annotates and that ultimately populate the training corpora. Second, a *three-stage sampling-and-filtering pipeline* combines large language model (LLM) candidate generation with progressively more expensive filters to produce *certified* ACSL loop invariant annotations for C programs, formally verified by Frama-C/WP. N parallel LLM calls are issued and all returned candidate expressions are *unioned* into a single pool, which is then filtered through (1) an *ACSL syntax filter* that rejects ill-formed expressions, (2) an *execution-trace semantic filter* that discards candidates falsified by any observed loop state, and (3) *Houdini formal pruning* via Frama-C/WP that removes non-inductive candidates, converging to the strongest certified inductive conjunction the union supports. Crucially, the filtering pipeline alone—before any fine-tuning—already yields substantial improvements in success rate and invariant quality over a single-call baseline. Third, the pipeline naturally produces *offline SFT and DPO training data* at no extra cost: every certified annotation becomes an SFT record, and every rejected candidate—labelled by the stage at which it was filtered—forms a DPO preference pair. This stage-stratified labelling avoids reward hacking and requires no online reinforcement learning. Running LoopFactory and this pipeline together at scale constitutes the training data generation workflow.

1 Introduction

Loop invariants are the cornerstone of deductive program verification. Given a loop annotated with an appropriate invariant, a verification condition generator such as Frama-C/WP [16] can reduce the correctness of the entire program to a set of first-order proof obligations dischargeable automatically. In practice, however, *finding* the right invariant is the bottleneck: even expert users spend substantial effort crafting invariants that are simultaneously *inductive* (preserved by each iteration), *sufficient* (strong enough to imply the postcondition), and *syntactically admissible* (expressible in the annotation language).

Classical approaches fall into two broad camps. *Static* techniques such as abstract interpretation [5], template-based constraint solving [4], and Houdini-style iterative weakening [9] offer soundness guarantees but are confined to fixed abstract domains or templates. *Dynamic* techniques infer candidates from execution traces using polynomial fitting [8], machine learning [29], or data-driven enumerative search [10], but must still close the gap between empirical observation and formal proof.

Recently, large language models (LLMs) have demonstrated remarkable ability at code understanding and generation. Several studies have explored LLM-driven invariant synthesis [3, 15, 25], showing that LLMs can produce plausible candidates even for non-trivial programs. However, existing approaches suffer from two structural weaknesses. First, they rely on a single LLM call or a simple repair loop without *systematic multi-stage filtering*, so many returned candidates are syntactically ill-formed, semantically inconsistent with program behaviour, or formally non-inductive. Second, the verification signal produced by these systems is discarded after each run; no mechanism exists for using that signal to *improve* the LLM policy that generates the candidates. The result is that each new program is treated as if the model had learned nothing from prior successes and failures.

Why invariant quality is hard to learn from. A fundamental obstacle to improving LLM-based invariant generation through reinforcement learning is that *evaluating candidate quality is structurally different here from other code generation tasks.* The correct annotation for a loop is not a single program but a *conjunction* of multiple invariant clauses, each capturing a different arithmetic relationship. This has two consequences. First, the binary verifier signal is *non-attributable*: the final correct annotation is typically assembled from contributions of several parallel LLM calls, so no individual call can be scored as simply correct or incorrect. Second, the binary signal is susceptible to *reward hacking*: trivially weak annotations (e.g., `loop invariant true;`) can formally pass a checker on programs with loose postconditions yet carry no mathematical insight about the loop. These two properties together explain why online reinforcement learning methods such as GRPO [6]—which score individual completions against each other via binary verifier feedback—are poorly suited to this domain.

Our approach. We present SAM2INV (**Sam**pling **to Inv**ariant), a system that addresses both problems through a two-stage pipeline: first generate a large, diverse corpus of loop programs; then for each program, generate, filter, and certify invariant candidates—logging every accepted and rejected expression as structured training data.

Contribution 1: LoopFactory—a Probabilistic DSL for Task Input Generation. Existing benchmarks contain only a few dozen loop programs, which is insufficient for training. We design LoopFactory, a probabilistic domain-specific language (DSL) that generates structured numeric loop programs with tunable arithmetic complexity (linear vs. nonlinear), nesting depth, variable count, and loop-control mode. LoopFactory is the *task input generator*: it produces the programs that the invariant-generation pipeline annotates, and hence the programs that ultimately populate the SFT and DPO training corpora. Without a scalable supply of programs with known structure, offline training data cannot be generated at the scale needed for fine-tuning.

Contribution 2: A Three-Stage Sampling-and-Filtering Pipeline. Given a program (from LoopFactory or any source), SAM2INV issues N parallel LLM calls and *unions* all returned candidate invariant expressions into a single pool, rather than selecting among them. The pool is then filtered through three progressively more expensive stages: (i) an ACSL *syntax filter* that rejects ill-formed expressions without running any code; (ii) an *execution-trace semantic filter* that discards candidates falsified by any observed loop state; and (iii) *Houdini formal pruning* that uses Frama-C/WP to remove non-inductive candidates, converging to the strongest inductive conjunction the union supports. Every survivor is *certified* by Frama-C/WP. Importantly, *the filtering pipeline alone—before any fine-tuning—already yields substantial improvements in both success rate and invariant quality* over a single-call baseline. Union assembly makes the final annotation stronger by combining conjuncts that different calls discover independently; the syntax and trace filters ensure that only well-formed, semantically consistent candidates reach Frama-C, eliminating parse failures and reducing the cost of Houdini pruning; and Houdini finds the maximal inductive conjunction the pre-screened union can support. The pipeline is therefore a strong standalone tool for invariant generation, and the offline training data collection is an additional improvement layer built on top of it. Running LoopFactory and this pipeline together at scale constitutes the training data generation workflow: each program yields one certified annotation and a set of stage-labelled rejected candidates.

Contribution 3: Offline SFT and DPO Data from Pipeline Logs. The union-then-filter design makes online RL inapplicable for the reasons above, but it naturally produces structured training data *at no extra cost*: the pipeline already logs every candidate and the stage at which it was filtered. SAM2INV turns these logs into SFT records (program paired with its certified annotation) and DPO preference pairs (certified annotation as chosen; each rejected candidate as rejected, labelled by its rejection stage). This stage-stratified labelling provides a richer signal than binary pass/fail, avoids reward hacking, and requires no human annotation or online reward model.

The remainder of this paper is organised as follows. Section 2 surveys related work. Section 3 gives a high-level overview of the SAM2INV pipeline. Section 4 details each component. Section 5 describes LoopFactory. Section 6 presents experimental evaluation. Section 7 concludes.

2 Related Work

Loop Invariant and Program Specification Generation. Classical invariant synthesis methods fall along a spectrum from fully static to fully dynamic. Abstract interpretation [5] computes fixed-point over-approximations in abstract domains (intervals, octagons, polyhedra) and underpins industrial analysers, but is restricted to the expressiveness of the chosen domain. Template-based constraint solving [4, 28] fixes a parametric invariant form and searches for satisfying coefficients via LP or SDP; the approach is complete within the template family but cannot discover structure outside it. The Houdini algorithm [9] starts from a large candidate set and iteratively removes non-inductive conjuncts, converging to the strongest inductive conjunction; we adopt this as the third stage of our filter pipeline. Daikon [8] and DIG [22] mine likely invariants from execution traces by instantiating templates over observed variable values or combining dynamic analysis with symbolic execution; like us, they rely on traces as evidence, but provide no formal inductiveness guarantee without a separate verifier pass. Beyond invariants, there is a growing body of work on broader *specification generation*: inferring pre/postconditions [7], separation-logic frame conditions [1], and ACSL contracts from source code. Our work targets loop invariant generation specifically, where the interaction between inductiveness and the loop guard makes the problem particularly challenging.

LLM-Based Formal Verification Assistance. Recent work has explored using LLMs to assist formal verification. Chakraborty and Lahiri [3] demonstrate that GPT-4 can generate loop invariants for simple C programs given appropriate prompts and a verification feedback loop. Pei et al. [25] study code-trained LLMs producing Hoare-style annotations, finding that iterative refinement with verifier feedback is essential. Kamath et al. [15] propose a "guess-and-check" framework combining LLM generation with bounded model checking. In the Lean and Coq proof assistant ecosystem, LLMs have been used to suggest proof tactics [13, 14] and to translate informal mathematical statements into formal proofs [30]. Compared to these works, SAM2INV introduces a *systematic three-stage filter pipeline* that pre-screens candidates before formal verification, dramatically reducing verifier load. Moreover, by combining multiple LLM candidates through the pipeline, the resulting invariant set is substantially stronger and more informative than any single-call approach: the diversity of parallel generation and the merging step allow the system to discover invariant conjuncts that no individual call would produce alone.

Automated Theorem Proving with Neural Models. Neural theorem proving has made rapid progress through tree-search over proof states using LLMs as tactic generators [13, 18, 26].

AlphaCode [20] and similar systems apply large-scale sampling and filtering for competitive programming, a paradigm closely related to ours. A key difference is that deductive program verification with Frama-C/WP does not admit a step-by-step proof search; instead, the verifier is an oracle that accepts or rejects a complete annotation in one shot, making the feedback signal coarser and the filtering pipeline more important.

Reinforcement Learning with Verifiable Rewards (RLVR).. Reinforcement learning from human feedback (RLHF) [24] and direct preference optimisation (DPO) [27] have become standard alignment techniques for LLMs, replacing scalar reward models with pairwise preference data. A natural extension replaces *human* preferences with *verifiable* preferences derived from automated oracles, a paradigm known as reinforcement learning with verifiable rewards (RLVR). DeepSeek-R1 [6] and related work [17] demonstrate that training on verifier-generated rewards for mathematical reasoning yields large capability gains with no human annotation. In code generation, execution-guided methods [11, 19] use test-case pass/fail signals to define preferences, and process reward models [21] assign credit to intermediate reasoning steps. Our approach inherits the RLVR spirit but addresses a distinctive challenge: in the loop invariant domain, binary verifier success is an especially sparse and *reward-hackable* signal, because a trivially weak invariant (e.g., `true`) may satisfy the verifier on programs without postconditions but is useless in practice. We address this by generating *structured* DPO preference pairs whose rejected members are annotated by the *stage* at which they fail (syntax, trace, or formal inductiveness), providing a much richer training signal than a flat pass/fail label. The preference labels are ground-truth correct by construction—a verified invariant truly satisfies Frama-C/WP's soundness conditions, and a rejected one truly does not—making our setting particularly well suited to offline preference optimisation.

Benchmark Generation and Training Data for Program Verification. Existing invariant synthesis benchmarks—such as the NLA (Non-Linear Arithmetic) suite from SV-COMP [2]—contain only a few dozen hand-crafted programs, making both systematic evaluation and large-scale training data generation difficult. LoopFactory addresses both needs: as the *task input generator* of SAM2INV, it generates large corpora of structured numeric loop programs that the pipeline annotates and logs as SFT/DPO training records; as a benchmark tool, it supports controllable ablation over program complexity dimensions. The design draws inspiration from grammar-based fuzzing [12] and probabilistic program synthesis [23].

3 System Overview

Figure 1 illustrates the end-to-end SAM2INV pipeline. The system takes as input a C function containing a `while` loop, a precondition (`requires`), and a postcondition (`assert`). It outputs the same function annotated with ACSL loop invariants and a `loop assigns` clause such that Frama-C/WP can fully verify the postcondition, together with SFT and DPO training records for offline policy fine-tuning.

SAM2INV Pipeline

❶ **Smart Sampling** — Execute the program with tiered inputs; collect execution traces recording variable values at every loop iteration.

❷ **Parallel LLM Generation** — Issue N parallel LLM calls with diverse prompts and temperatures; parse and union all candidate invariants into a single pool.

❸ **Stage 1 – ACSL Syntax Filter** — Reject candidates that violate ACSL syntax rules (unsupported quantifiers, forbidden operators, etc.).

❹ **Stage 2 – Execution-Trace Semantic Filter** — Discard candidates falsified by any observed loop state without invoking the verifier.

❺ **Stage 3 – Houdini Formal Pruning** — Iteratively remove non-inductive invariants via Frama-C/WP; the surviving set is *certified*.

❻ **Offline Data Collection** — Record verified candidates as SFT data; pair them with rejected candidates as DPO preference pairs. Output annotated C code (or report failure).

Figure 1: The SAM2INV sampling-and-filtering pipeline. Steps ❶–❷ form Phase I (Sampling & Generation); steps ❸–❺ form Phase II (Three-Stage Filtering); step ❻ constitutes Phase III (Certification & Offline Data Collection). Houdini pruning in step ❺ is guaranteed to terminate.

The pipeline operates in three broad phases.

Phase I: Sampling and Generation (Steps ❶–❷). The target program is compiled and executed under a set of carefully chosen inputs produced by the *smart tiered sampler* (Section 4.1). Execution traces—recording variable values at every loop iteration—are collected and formatted into structured text that exposes the mathematical relationships maintained by the loop. Multiple LLM instances are then queried in parallel, each receiving a prompt assembled from the source code, execution traces, and a system prompt encoding ACSL rules and an assertion-driven synthesis strategy. Prompt diversity is achieved by varying the template and the sampling temperature. The union of all parsed responses forms the initial candidate pool.

Phase II: Three-Stage Filtering (Steps ❸–❺). The candidate pool is subjected to three progressively more expensive filters. The *ACSL syntax filter* checks each candidate against a set of structural rules—no unsupported quantifiers, no forbidden operators, correct use of `\at` expressions—and discards violators immediately. The *execution-trace semantic filter* evaluates each surviving candidate against every recorded loop state; any candidate falsified by at least one observed state is discarded without invoking the verifier. Finally, *Houdini formal pruning* passes the pre-screened set to Frama-C/WP, iteratively removing candidates whose verification

conditions fail until the remaining set is fully inductive. Because at least one candidate is removed per Houdini iteration, termination is guaranteed (Theorem 4.2). Every candidate that exits Phase II is formally certified by Frama-C/WP.

Phase III: Certification and Offline Data Collection (Step ❻). The certified invariant set is written back into the source file as the verified output. Simultaneously, the pipeline records two complementary training signals. Candidates accepted through all three filter stages are recorded as *chosen* examples for supervised fine-tuning (SFT) of an invariant-generation policy. Candidates rejected at any earlier stage are paired with the corresponding chosen examples to form *direct preference optimisation* (DPO) training pairs, providing ground-truth preference labels derived entirely from formal verification—no human annotation required. This offline data collection adds negligible overhead: no extra LLM calls are made, and the records are produced as a natural byproduct of pipeline execution.

Running Example. Consider the following program that computes the cube of n:

```
/*@ requires a>=n && n==0; */
int main1(int a, int n){
  int x, y, z;
  x=0; y=1; z=6;
  while(n <= a){
    n=n+1; x=x+y; y=y+z; z=z+6;
  }
  /*@ assert (n==a+1) && (y==3*n*n+3*n+1)
           && (x==n*n*n) && (z==6*n+6); */
}
```

The smart sampler executes this program with $a \in \{0, 1, \ldots, 10\}$ and collects traces such as:

iter	n	x	y
0	0	0	1
1	1	1	7
2	2	8	25

The parallel LLM generation step produces a pool of candidate invariants. Ill-formed candidates (e.g., those using exponentiation `^`) are eliminated by the syntax filter; candidates that evaluate to `false` on any observed row are eliminated by the trace filter. Houdini pruning then confirms the four inductive invariants $x = n^3$, $y = 3n^2{+}3n{+}1$, $z = 6n{+}6$, and $n \leq a{+}1$, and Frama-C/WP verifies the postcondition in a single pass. The verified invariants are stored as SFT data; every rejected candidate is paired with this verified set to form a DPO training record.

4 Method

We now describe each component of SAM2INV in detail, following the pipeline order introduced in Section 3.

4.1 Smart Dynamic Sampling

The purpose of dynamic sampling is to collect execution traces that expose the mathematical relationships maintained by the loop. Naive random sampling often produces inputs that are either too simple (all zeros) or too large (causing integer overflow or time-outs). We therefore adopt a *tiered* sampling strategy that prioritises simple, informative values while gradually introducing diversity.

Definition 4.1 (Value Tiers). Given a parameter variable v with domain $[l, u]$, we define four value tiers:

$$\mathcal{T}_0 = \{0, 1, -1\} \cap [l, u] \quad \text{(special values)}$$
$$\mathcal{T}_1 = \{2, 3, \ldots, 10\} \cap [l, u] \quad \text{(small integers)}$$
$$\mathcal{T}_2 \subseteq \{11, \ldots, 50\} \cap [l, u] \quad \text{(medium, sampled)}$$
$$\mathcal{T}_3 \subseteq \{51, \ldots, 100\} \cap [l, u] \quad \text{(large, sampled)}$$

Sampling proceeds in phases:

(1) **Phase 1:** Generate the Cartesian product over $\mathcal{T}_0$ for all parameters. These "corner cases" are ideal for fitting low-degree polynomials.
(2) **Phase 2:** Mix $\mathcal{T}_0$ and $\mathcal{T}_1$ to add coverage of small positive integers.
(3) **Phase 3:** Include all tiers, prioritising combinations with lower total tier index.
(4) **Phase 4:** Fill remaining quota with biased random sampling (80% probability of choosing values ≤ 20).

Each input is executed, and the variable state at every loop iteration is recorded. To keep prompts within LLM context limits, we retain only the first and last k iterations per run (default k=3) and group traces into at most G groups (default G=10).

4.2 Parallel LLM Generation

Prompt Design. Each LLM prompt is assembled from three components:

(1) A *system prompt* that defines the task (loop invariant synthesis for Frama-C/WP), lists allowed and forbidden ACSL constructs, and encodes the assertion-driven synthesis strategy (decompose postcondition conjuncts, add boundary invariants, preserve unmodified parameters).
(2) The *source code* of the target function.
(3) *Execution traces* formatted as structured variable–value tables (Section 4.1).

Assertion-Driven Strategy. A key insight is that the postcondition assertion directly suggests invariant candidates:

(1) Each conjunct of the assertion (split on `&&`) is a candidate invariant.
(2) For a loop guard of the form `i < n`, the weakened form `i <= n` is an inductive boundary invariant.
(3) Function parameters not modified in the loop body are preserved: `v == \at(v, Pre)`.

Parallel Diverse Generation and Union Assembly. To maximise the probability of finding the correct invariant set, we issue N parallel LLM calls (default N=5) using a thread pool, with temperatures τ_j sampled from $\{0.8, 1.0, 1.1, 1.2\}$ to encourage diverse outputs. Each call j produces a set of ACSL expression strings C_j. These sets are *unioned*, not

voted upon:

$$C_{\text{pool}} = \bigcup_{j=1}^{N} C_j \quad \text{(whitespace-normalised deduplication)}.$$

The union C_{pool} is then passed as a single flat set to the three-stage filter (Section 4.3).

The union-then-filter design is motivated by a structural property of loop invariants: the correct annotation is a *conjunction* of multiple clauses, and different clauses tend to be discovered by different calls. After Houdini pruning, the certified output C^* is assembled from whichever clauses in C_{pool} survive—it is a post-hoc collective result, not a selection from any individual response. This *non-attributability* property has direct consequences for training signal design; see Section 4.4 and Appendix B.4 for the formal analysis.

4.3 Three-Stage Filter Pipeline

The three-stage filter pipeline is the core engine of SAM2INV. It improves invariant generation in two independent ways.

First, *union assembly strengthens the invariant.* By pooling candidates from N parallel calls, the union contains conjuncts that no individual call would produce on its own. After Houdini pruning, the certified C^* is the strongest inductive conjunction the union supports—strictly stronger, in general, than the output of any single call (Section 4.2, Appendix B.4).

Second, *the lightweight filters ensure Houdini can work correctly and efficiently.* The syntax filter removes expressions that are syntactically malformed ACSL—parse errors, forbidden constructs, unbalanced brackets—which would cause Frama-C to fail entirely rather than returning a per-invariant result. The trace filter removes candidates that are *semantically contradicted* by observed execution states: any candidate falsified by at least one recorded loop state cannot be inductive and need never be sent to Frama-C. Together, these two stages deliver a pre-screened candidate set to Houdini, eliminating the risk of Frama-C parse failures and substantially reducing the number of formal verification calls.

Crucially, both improvements are *intrinsic to the pipeline itself*—they hold for any base LLM, before any fine-tuning. The offline training data collection (Section 4.4) is a separate improvement layer built on top of this already-strong baseline.

The design principle across stages is to discard as many incorrect candidates as possible *before* invoking the more expensive stage that follows.

Stage 1: ACSL Syntax Filter. Each candidate is checked against a set of structural ACSL rules before any execution or verification is attempted:

- No quantifiers (`\forall`, `\exists`), which are unsupported by Frama-C/WP for loop invariants.
- No custom definitions (`predicate`, `logic`, `lemma`).
- No ternary operator (`? :`).
- `\at(v, Pre)` permitted only on function parameters, not on local variables.
- No exponentiation (`^` is bitwise XOR in C/ACSL; repeated multiplication must be used instead).

Algorithm: Houdini Pruning
Input: Candidate set C, verifier $\mathcal{V}$
Output: Maximal inductive subset $C^* \subseteq C$
1: $C^* \leftarrow C$
2: **repeat**
3: $R \leftarrow \mathcal{V}(C^*)$ *// verify each invariant*
4: $F \leftarrow \{c \in C^* \mid R(c) = \mathit{fail}\}$
5: $C^* \leftarrow C^* \setminus F$
6: **until** $F = \emptyset$ **or** $C^* = \emptyset$
7: **return** C^*

Figure 2: Houdini-style pruning algorithm used in Stage 3 of the filter pipeline.

Candidates failing any rule are discarded immediately. This filter runs in microseconds per candidate and eliminates a substantial fraction of the pool before any program execution is needed.

Stage 2: Execution-Trace Semantic Filter. Each surviving candidate is evaluated as a boolean expression over every recorded loop state in the collected execution traces. A candidate that evaluates to `false` on any observed state is *inconsistent* with the program's actual behaviour and is discarded. While trace evaluation does not guarantee inductiveness—the traces are finite and cannot cover all reachable states—it eliminates candidates with grossly incorrect arithmetic relationships cheaply, without invoking Frama-C. In practice, this stage reduces the candidate pool by a further significant fraction before the formal verifier is engaged.

Stage 3: Houdini Formal Pruning. After the two lightweight filters, the surviving candidates are submitted to Frama-C/WP for formal verification. We apply the Houdini algorithm [9]:

Theorem 4.2 (Termination). *Algorithm 2 terminates in at most $|C|$ iterations, since each iteration removes at least one candidate.*

The output C^* is the maximal subset of the pre-screened candidates that is simultaneously inductive and collectively sufficient to discharge the Frama-C/WP verification conditions. Every invariant in C^* is formally certified. If C^* is non-empty and sufficient to prove the postcondition, the pipeline reports success. If the pruned set is non-empty but insufficient, the system may optionally enter a brief iterative repair loop (at most K iterations; default K=3) in which verification error messages are fed back to the LLM for targeted strengthening; this repair step is not the primary mechanism and is not counted as part of the three-stage filter.

4.4 Offline Policy Exploration

A distinctive property of the SAM2INV pipeline is that it produces not only a certified invariant annotation but also high-quality training data for an invariant-generation policy—as a byproduct of every pipeline run.

Why not GRPO.. A natural baseline is group-relative policy optimisation (GRPO) [6], which scores each individual completion via binary verifier feedback. Two structural properties make this unsuitable here. First, *non-attributability* (Section 4.2, Appendix B.4): the correct annotation C^* is assembled post-hoc from the union of N calls, so a call that contributes only one essential conjunct will fail binary verification on its own yet be indispensable to the final answer—penalising it degrades the diversity that makes the union effective. Second, *binary reward hackability*: trivially weak annotations such as `loop invariant true;` can formally pass the verifier on programs with loose postconditions, making binary pass/fail an unreliable quality signal.

DPO data collection. Both problems are avoided by *offline* preference optimisation. The pipeline logs every candidate and the stage at which it was filtered; this log is converted into DPO training records at no extra cost. Let C^* be the certified set and P^+ the annotated program obtained by inserting C^*. For each rejected candidate r eliminated at stage $k \in \{1, 2, 3\}$, we form the record $(P,\ P^+,\ P_r^-,\ k)$ where P_r^- inserts $\{r\}$ as the sole annotation. The preference label is ground-truth correct by construction. Rejected candidates are weighted by stage: syntax failures (k=1) receive lower weight than inductiveness failures (k=3), since the latter represent the hardest and most informative contrasts. The SFT record $(P,\ P^+)$ is generated alongside each successful pipeline run. Formal definitions and the stage-stratified loss are given in Appendix C.

Policy improvement cycle. Fine-tuning on the accumulated SFT and DPO data produces a model that is deployed in place of the base LLM in Step ❷. The filter pipeline certifies every output regardless of model quality—a weaker model yields a smaller C^* but never an uncertified one—so the system remains sound throughout the improvement cycle.

5 Probabilistic Loop Synthesis

Offline training of a specialised invariant-generation policy requires a scalable supply of loop programs together with their certified invariant annotations. Existing benchmarks (e.g., the NLA suite from SV-COMP) contain only a few dozen hand-crafted programs, which is insufficient for fine-tuning data generation or for studying how synthesis success varies with program complexity. We design LoopFactory, a probabilistic domain-specific language (DSL) for generating structured numeric loop programs with controllable complexity, as the *task input generator*: it produces the programs that the SAM2INV pipeline annotates, thereby populating the SFT and DPO training corpora at scale. LoopFactory also supports large-scale evaluation beyond fixed benchmarks.

5.1 Hyperparameters

The generator is parameterised by:

$$\theta = (m,\ p,\ n,\ k,\ D_{\max},\ \pi_{\text{op}},\ \pi_{\text{cmp}},\ \pi_{\text{const}},\ \pi_{\text{self}},\ \pi_{\text{nest}})$$

Table 1: Hyperparameters of the LoopFactory DSL.

Symbol	Meaning
m	Max variables
p	Parameter (immutable) variables
n	Top-level loops
k	Max assignments per loop body
$D_{\max}$	Max nesting depth
π_{op}	Distribution over $\{+, -, \times, /, \text{mod}\}$
π_{cmp}	Distribution over $\{<, \leq, >, \geq, =, \neq\}$
π_{const}	Prob. of sampling a constant operand
π_{self}	Prob. of self-update ($v_i := v_i$ `op` x)
π_{nest}	Prob. of generating a nested sub-loop

where m is the maximum number of variables, p the number of (immutable) parameter variables, n the number of top-level loops, k the maximum assignments per loop body, $D_{\max}$ the maximum nesting depth, and the remaining symbols are probability distributions governing operator choice, constant injection, self-update assignment, and loop nesting (Table 1).

5.2 DSL Syntax

Expressions. Arithmetic expressions are of the form:

$$e \ ::= \ v \ \mid \ c \ \mid \ v \ \texttt{op} \ x$$

where $v \in \mathcal{V}$ is a variable, $c \in \mathbb{Z}$ is an integer constant, $x \in \mathcal{V} \cup \mathbb{Z}$, and $\texttt{op} \in \{+, -, \times, /, \text{mod}\}$. If $\texttt{op} = \text{mod}$, the right operand must be a positive integer constant.

Boolean Guards. Loop guards are comparisons e_1 `cmp` e_2 with $\texttt{cmp} \in \{<, \leq, >, \geq, =, \neq\}$.

Assignments. An assignment $v := e$ requires $v \in \mathcal{L}$, where $\mathcal{L} = \mathcal{V} \setminus \mathcal{P}$ is the set of writable (non-parameter) variables.

5.3 Program Structure

A generated program has the form:

$$\textbf{Prog} \ ::= \ \textbf{Init} \ ; \ \textbf{LoopForest}$$

The **Init** block assigns every writable variable exactly once, using expressions that depend only on parameter variables, ensuring a well-defined initial state.

The **LoopForest** is a sequence of loop trees $\mathcal{F} = [T_1, \ldots, T_n]$, where each tree node

$$T = \langle\, b,\ S,\ \mathcal{C}\, \rangle$$

consists of a guard b, an assignment list S with $|S| \leq k$, and a (possibly empty) list of child loops $\mathcal{C}$. Nesting depth is bounded by $D_{\max}$.

5.4 Semantic Loop Templates

To ensure that generated loops exhibit mathematically interesting behaviour (rather than trivial or divergent patterns), the DSL includes twelve *semantic templates* that encode common loop idioms:

(1) Cubic growth recursion (x, y, z coupled updates)
(2) Geometric recursion ($x = x \cdot z + 1$)

(3) Cumulative sum / dot product
(4) Power-sum accumulation
(5) Quotient–remainder counting
(6) Binary multiplication style
(7) Linear dual-variable shifting
(8) Linear decrement to zero
(9) Modulo bucket counting
(10) Euclidean-style subtraction
(11) Nested triangular sums
(12) Nested affine updates

Each template defines a fixed update pattern but allows the generator to sample concrete operators and constants from the probabilistic model, yielding a family of programs per template.

5.5 Generative Distribution

The probability of a complete program factorises as:

$$\Pr(\mathbf{Prog}) = \prod_{v \in \mathcal{L}} \Pr(v := e_v) \cdot \prod_{T \in \mathcal{F}} \Pr(T) \tag{1}$$

where, recursively,

$$\Pr(T) = \Pr(b) \cdot \prod_{s \in S} \Pr(s) \cdot \prod_{T' \in \mathcal{C}} \Pr(T').$$

This factorisation ensures that the generator defines a proper probability distribution over the space of well-formed loop programs, enabling statistical analysis of benchmark difficulty and ablation over individual hyperparameters.

Expected Properties. Under the Bernoulli nesting model, the expected nesting depth is:

$$\mathbb{E}[\text{depth}] = \sum_{d=0}^{D_{\max}} d \cdot (1 - \pi_{\text{nest}})\, \pi_{\text{nest}}^{d}$$

and the expected number of loop nodes is $\sum_{d=0}^{D_{\max}} \pi_{\text{nest}}^{d}$. These closed-form expressions allow practitioners to tune π_{nest} and $D_{\max}$ to obtain a desired distribution of benchmark difficulty.

6 Experiments

7 Conclusion

We have presented SAM2INV, a three-contribution system for certified loop invariant generation.

LoopFactory, our probabilistic DSL, serves as the *task input generator*: it generates structured numeric loop programs with controllable arithmetic complexity, nesting depth, and variable count, supplying the programs that the pipeline annotates and that populate the offline training corpora. Its factorised generative distribution also supports principled ablation over program complexity dimensions, making it possible to characterise synthesis performance as a function of program structure.

The *three-stage sampling-and-filtering pipeline* is the core engine. It assembles a diverse candidate pool through smart tiered sampling and parallel LLM generation, then eliminates incorrect candidates through three successive filters: an ACSL syntax filter, an execution-trace semantic filter, and Houdini formal pruning via Frama-C/WP. Every invariant that exits the pipeline is formally certified. Crucially, the pipeline delivers substantial improvements even before any fine-tuning: union assembly of N parallel calls produces a stronger certified invariant than any single call, and the lightweight filters ensure that Houdini receives only well-formed, semantically consistent candidates, eliminating parse failures and reducing formal verification cost.

The *offline SFT and DPO data generation* is a byproduct of ordinary pipeline runs at no extra cost. Every certified annotation becomes an SFT record; every rejected candidate—annotated by the stage at which it was filtered—forms a DPO preference pair. This stage-stratified labelling avoids the reward hacking that plagues binary pass/fail signals, provides a richer training signal than online RL methods such as GRPO, and requires no human annotation or learned reward model. The preference labels are ground-truth correct by construction, making this setting particularly well suited to offline preference optimisation.

Several directions remain for future work. Most directly, the accumulated SFT and DPO datasets should be used to conduct systematic fine-tuning experiments, measuring how much the success rate of the filter pipeline improves as the policy model is updated over successive training rounds. A second priority is extending the system to heap-manipulating programs, where invariants must express separation-logic properties; our codebase already partially supports Coq/VST verification, and connecting this to the filter pipeline is a natural next step. Finally, integrating symbolic techniques—such as abstract interpretation for computing initial invariant bounds or interpolation for strengthening failing candidates—could provide a hybrid symbolic–neural architecture with stronger guarantees on programs that lie outside the current system's success region.

A LoopFactory DSL: Sampling Procedure and Concrete Example

A.1 Full Generation Procedure

Algorithm 3 shows the complete LoopFactory generation loop. Starting from a drawn hyperparameter vector θ (or a fixed configuration), the factory (i) selects parameters from a candidate pool, (ii) samples loop count and, for each loop, the NLA vs. linear family classification, (iii) allocates fresh single-letter variable names for each loop's counter and limit, (iv) samples loop control modes (increment, decrement, multiplicative, distance-to-limit, etc.), (v) fills each loop body with self-update and peer-update assignments drawn from π_{op}, and (vi) optionally inserts semantic-core assignments from the twelve hardcoded idiom templates. The rendered program is then wrapped in a function signature with a Frama-C `requires` clause and passed downstream for execution and invariant generation.

A.2 Loop-Control Mode Sampling

The function `SampleLoopControl` draws a loop-control mode from seven shapes (Table 2). Weights differ between the

Algorithm: LoopFactory Generation

Input: Hyperparameters θ, seed s
Output: A well-typed C function

$rng \leftarrow \text{Random}(s)$
$\mathcal{P} \leftarrow \text{sample}(\text{candidates}, p, rng)$ // *e.g.* $\{a,n\}$
$n_\ell \leftarrow \text{Uniform}[\text{min\_fuel}, \text{while\_fuel}]$ // *loop count*
Init block: for $v \in \mathcal{L}$, sample e_v over $\mathcal{P}$ only
for $i = 1, \ldots, n_\ell$ **do**
  Allocate counter ctr, limit lim via `NameAllocator`
  $nla \leftarrow \text{Bernoulli}(p_{\text{nonlinear}})$
  (inits, guard, step) $\leftarrow$ `SampleLoopControl`(ctr, lim, nla)
  Body $\leftarrow$ [step]
  repeat up to $k-1$ times:
    Sample assign $\leftarrow$ `SemanticAssign`(nla)
    Body $\leftarrow$ Body $\cup$ {assign}
  With prob. $p_{\text{semantic\_core}}$: inject one template idiom
  With prob. q_{nest}: recurse to depth $d{+}1 \le D_{\max}$
  Append `WhileLoop(guard, Body)` to forest
end for
return `Program`($\mathcal{P}$, inits, forest).render()

Figure 3: LoopFactory generation algorithm.

NLA and linear families: multiplicative and distance-to-limit modes are disabled for linear programs.

Table 2: Loop-control modes and their effect on counter, guard, and step.

Mode	Guard shape	Step
`inc1`	$ctr < lim$	$ctr \mathrel{+}= 1$
`dec1`	$ctr > 0$	$ctr \mathrel{-}= 1$
`inc_step`	$ctr < lim$	$ctr \mathrel{+}= d,\ d \in [2,5]$
`dec_step`	$ctr > d{-}1$	$ctr \mathrel{-}= d$
`mul_up`	$ctr < lim$	$ctr \mathrel{*}= m,\ m \in \{2,3\}$
`div_down`	$ctr > 0$	$ctr \mathrel{/}= 2$
`dist_to_limit`	$ctr > lim$	$ctr \mathrel{-}= d$

A.3 Concrete Generated Example

The following function was generated by LoopFactory with $\theta = (m{=}5,\ p{=}2,\ \text{while\_fuel}{=}2,\ k{=}4,\ D_{\max}{=}1,\ p_{\text{nonlinear}}{=}0.55)$:

```
/*@ requires a >= 0 && n >= 1; */
int f(int a, int n){
  int c, s, x;
  c=0; s=a; x=1;

  while (c < n) {        // inc1: ctr=c, lim=n
    c = c+1;
    s = s+c;             // semantic: cumulative
        sum
    x = x*c;             // NLA: factorial-style
  }

  while (x > 0) {        // div_down: ctr=x
    x = x/2;
  }
  /*@ assert s == a + n*(n+1)/2; */
}
```

This single program exercises: (i) an NLA loop with invariants $s = a + c(c+1)/2$ and $x = c!$; (ii) a geometric-decay loop requiring $x \ge 0$; (iii) a postcondition that tests only the sum invariant, so the SAM2INV pipeline must discover the correct invariant by combining candidates from multiple LLM calls.

B Detailed Invariant Generation Algorithm

B.1 Data Structures

The pipeline maintains the following state for each outer loop index i:

- **record**$_i$: the structured descriptor extracted from static analysis, containing the loop guard, assignments (transition relation), initial variable values, invariable (unmodified) parameters, and function parameter names.
- $\mathcal{T}_i$: the execution trace set, a list of variable-state snapshots collected by the smart sampler.
- C_i: the live candidate pool (a set of ACSL expression strings).
- C_i^*: the certified invariant set (output).

B.2 Complete Per-Loop Algorithm

B.3 Role of Static Analysis

Before Phase 1, the sampler's `LoopAnalysis` module performs a *lightweight static pre-pass* on the input C file to populate **record**$_i$. This is *not* symbolic execution in the strict sense; it is a source-level analysis that:

(1) Identifies the loop guard (transition relation) by reading the `condition` field recorded in the JSON analysis file produced by the Frama-C preprocessing step.
(2) Extracts the set of variables that are never assigned inside the loop body (*unchanged variables*), which become candidates for $\backslash\texttt{at}(v, \text{Pre})$ invariants.
(3) Infers conservative integer bounds $[\ell_v, u_v]$ for each parameter via `LoopBoundAnalyzer`, which are used to constrain the smart sampler's input domain.
(4) Builds the **record** structure that encodes loop content, transition expression, pre-condition, and the names of function parameters (needed to gate `\at` usage in the syntax filter).

The Frama-C/WP oracle is *only* invoked during the Houdini pruning stage (Phase 3) and never during candidate generation or lightweight filtering.

B.4 Non-Attributability of the Certified Output

We formalise the key structural property stated informally in Section 4.2.

Algorithm: InvariantGenerator per loop i

Input: Source code P, record $\mathbf{r}$, traces $\mathcal{T}$, LLM $\mathcal{M}$, verifier $\mathcal{V}$

Output: Annotated code P^* or $\perp$

Phase 1 — Generation:
$P_{\text{tmpl}} \leftarrow \texttt{InsertTemplate}(P, i)$ *// PLACE_HOLDER annotations*
$(\sigma, \ell) \leftarrow \texttt{PreparePrompt}(\mathbf{r}, \mathcal{T})$ *// (system, user) prompts*
for $j = 1, \ldots, N$ **in parallel do**
  $R_j \leftarrow \mathcal{M}(P_{\text{tmpl}}, \sigma, \ell,\ \tau_j)$ *// diverse temperature*
  $C_j \leftarrow \texttt{ExtractInvariants}(R_j)$
end for
$C \leftarrow \bigcup_j C_j$ *// union of all candidates*

Phase 2a — Heuristic ACSL Syntax Gate:
$C \leftarrow \{c \in C \mid \texttt{SyntaxFilter}(c, \mathbf{r}) = \text{pass}\}$

Phase 2b — Execution-Trace Semantic Gate:
$C \leftarrow \{c \in C \mid \forall \mathbf{s} \in \mathcal{T} :\ \texttt{eval}(c, \mathbf{s}) = \text{true}\}$

Phase 2c — Conflict Detection:
$C \leftarrow \texttt{RemoveConflicts}(C)$

Phase 2d — Merge:
$C_{\text{merged}} \leftarrow \texttt{Dedup}(C)$ *// whitespace-normalised*

Phase 3 — Houdini Formal Pruning:
$C^* \leftarrow \texttt{Houdini}(C_{\text{merged}}, \mathcal{V})$ *// see Algorithm 2*
if $C^* = \emptyset$ **then return** $\perp$

Phase 4 — Postcondition Check & Optional Repair:
if $\mathcal{V}$ reports `assert` satisfied **then**
  return $\texttt{Annotate}(P, C^*)$
else run $\leq K$ repair iterations (feed error to $\mathcal{M}$)
return $\texttt{Annotate}(P, C^*)$ or $\perp$

Figure 4: Full per-loop invariant generation algorithm.

Definition B.1 (Attribution). Given N candidate sets $C_1, \ldots, C_N$ and a certified output C^*, we say C^* is *attributable to call* j if $C^* \subseteq C_j$.

Proposition B.2 (Non-attributability is typical). *Suppose the correct annotation for a loop requires a conjunction of $m \geq 2$ invariant clauses $c_1, \ldots, c_m$. If each clause c_i is discovered independently by each of the N calls with probability $q \in (0, 1)$, then the probability that* all *clauses appear in any single call C_j is q^m, while the probability that all clauses appear in the union $C_{pool} = \bigcup_j C_j$ is $1 - (1 - q)^{mN}$. For $q = 0.6$, $m = 4$, $N = 5$: single-call coverage $= 0.6^4 \approx 0.13$; union coverage $= 1 - 0.4^{20} \approx 1.00$.*

Proof. Follows directly from independence and the complement rule for union of events. □

Proposition B.2 shows that the gap between single-call and union coverage grows exponentially with the number of required conjuncts m. In practice, real loop invariants for arithmetic programs require $m \in [2, 6]$ conjuncts (boundary invariants, postcondition conjuncts, auxiliary equalities), making single-call attribution unlikely in typical cases.

Corollary B.3. *Under the conditions of Proposition B.2 with $m \geq 2$ and $q < 1$, the expected fraction of per-call outputs C_j for which $C^* \subseteq C_j$ is $q^m < 1$. Assigning a positive reward to calls satisfying $C^* \subseteq C_j$ and a negative reward to others would on average penalise $(1 - q^m) > 0$ fraction of calls that may have contributed essential clauses to the union.*

This corollary shows that any *per-call* binary scoring scheme — including GRPO — systematically mislabels calls that contribute essential clauses to the final answer, degrading the training signal for exactly the behaviours (diverse conjunct coverage) that make the union effective.

C DPO Data Construction: Formal Specification

C.1 Chosen Response Construction

Given a program P and the certified invariant set C^* for loop i, the chosen annotated program is:

$$P^+ = \texttt{Annotate}(P,\ i,\ C^*)$$

where `Annotate` inserts the ACSL block

```
/*@ loop invariant c_1; ...loop invariant c_k;
      loop assigns vars; */
```

immediately before the `while` statement of loop i. The `loop assigns` clause is derived from the set of variables assigned anywhere in the loop body. P^+ is guaranteed to satisfy Frama-C/WP because it survived Stage 3 (Houdini pruning); no additional verification is needed for the training record.

C.2 Rejected Response Construction and Stage Labelling

For each $r \in C_{\text{pool}} \setminus C^*$, we construct:

$$P_r^- = \texttt{Annotate}(P,\ i,\ \{r\})$$

and record the stage $k(r) \in \{1, 2, 3\}$ at which r was eliminated:

- $k(r) = 1$: r failed the ACSL syntax gate (Appendix E).
- $k(r) = 2$: r passed syntax but was falsified by some trace state $\mathbf{s} \in \mathcal{T}$, i.e. $\texttt{eval}(r, \mathbf{s}) = \text{false}$.
- $k(r) = 3$: r passed syntax and trace filters but was removed by Houdini because Frama-C/WP could not verify its establishment or preservation condition.

The full DPO training record for program P, loop i, and rejected expression r is the tuple:

$$\mathcal{D}(P, i, r) = \big(P,\ P^+,\ P_r^-,\ k(r)\big).$$

C.3 Stage-Stratified Loss Weighting

Standard DPO minimises the loss:

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(P^+ \mid P)}{\pi_{\text{ref}}(P^+ \mid P)} - \beta \log \frac{\pi_\theta(P_r^- \mid P)}{\pi_{\text{ref}}(P_r^- \mid P)}\right)\right].$$

We extend this with stage-dependent weights w_k:

$$\mathcal{L}_{\text{S-DPO}} = -\mathbb{E}\left[w_{k(r)} \cdot \log \sigma\left(\beta \log \frac{\pi_\theta(P^+ \mid P)}{\pi_{\text{ref}}(P^+ \mid P)} - \beta \log \frac{\pi_\theta(P_r^- \mid P)}{\pi_{\text{ref}}(P_r^- \mid P)}\right)\right].$$

The rationale for $w_1 \leq w_2 \leq w_3$ is that Stage-3 rejections are the hardest preference distinctions: the rejected expression is consistent with all observed program states and was eliminated only by a formal inductiveness check, making the margin between P^+ and P_r^- the smallest and the training signal the most informative. Concretely, we use $w_1 = 0.5,\ w_2 = 1.0,\ w_3 = 2.0$ as default weights, reflecting the increasing difficulty of the corresponding rejection judgements.

C.4 SFT Records

Alongside DPO pairs, each successful pipeline run produces an SFT record $(P,\ P^+)$. SFT pre-training warms up the model towards the format and style of ACSL annotations before DPO alignment, following the standard two-phase fine-tuning recipe [24]. Because P^+ is formally verified, SFT on these records teaches the model what a *correct and informative* annotation looks like, providing a positive learning signal that complements the contrastive DPO signal.

D Prompts Used in Each Stage

D.1 System Prompt

The system prompt is loaded verbatim from `prompts/system_prompt.txt` and prepended to every LLM call. Its full content is reproduced below.

```
You are a formal verification expert specializing in loop
invariant synthesis for Frama-C with the WP plugin.

## TASK
Given a C function with a loop and a postcondition
(/*@ assert ... */), generate ACSL loop annotations
(invariants, assigns) that enable Frama-C/WP to verify
the assertion.

## CORE STRATEGY: Assertion-Driven Invariant Synthesis
1. Decompose the assertion: each conjunct of the assert
   (split on &&) is a candidate loop invariant.
2. Add boundary invariants: weaken the loop guard.
   - while(i < n) needs: i <= n
   - while(n <= a) needs: n <= a+1
3. Preserve parameter values: if a parameter is unmodified,
   assert: loop invariant x == \at(x, Pre);
4. Verify sufficiency: check that (invariants AND NOT guard)
   implies the postcondition.

## FORBIDDEN CONSTRUCTS
- \forall , \exists   (quantifiers not supported)
- predicate, logic, axiomatic, lemma
- Ternary ? :
- \at(local_var, Pre)  (\at only on function parameters)
- ^ for exponentiation  (write x*x*x for x cubed)

## ALLOWED OPERATORS
Comparison: ==, !=, <, <=, >, >=
Logical:    &&, ||, !, ==>, <==>
Arithmetic: +, -, *, /, %

## OUTPUT
Return ONLY the annotated C function. Do not include
explanations.
```

D.2 User Prompt (Loop Context Block)

The user prompt is assembled by `PromptFormatter` and passed as the second message in the conversation. It consists of two parts.

Part 1 | Loop context.

```
### Loop Context: ###

1. Loop Context
  A. Pre-Condition (Before Loop Entry): ‘<precondition>‘
  B. Loop Transition Relation: ‘<transition_expr>‘
  C. Loop Snippet:
‘‘‘c
<loop_code>
‘‘‘

2. Execution Traces
   Each trace shows the full sequence of conditional
   evaluations, step by step.

   WARNING: CRITICAL TEMPORAL SEMANTICS:
   - Each state is labeled as
     'BEFORE loop body execution' or 'AFTER loop terminates'
   - Loop invariants must hold at the START of each iteration
     (BEFORE body executes)
   ...
```

Part 2 — Execution traces. For each sampled input group (up to G=5), the formatter emits a trace block:

```
[TRACE 1]
    [ n==0 && x==0 && y==1 && z==6 ]
        (BEFORE loop starts) ->
    [ n==1 && x==1 && y==7 && z==12 ]
        (BEFORE iteration 1 body executes) ->
    [ n==2 && x==8 && y==25 && z==18 ]
        (AFTER loop terminates)
```

The temporal labels (`BEFORE iteration` k `body executes`) are essential: they tell the LLM that an invariant must hold at loop entry of each iteration, not at loop exit, which is a common source of error in naive prompting. Initial values use $\texttt{\textbackslash at}(v, \text{Pre})$ notation so the model can directly reference them in `loop invariant` lines.

Diversity via temperature. Each of the N parallel LLM calls uses an independently sampled temperature τ_j. In the default configuration, the primary call uses $\tau_1 = 1.0$ and subsequent calls are drawn uniformly from $\{0.8, 1.0, 1.1, 1.2\}$, ensuring that the candidate pool covers both conservative and more exploratory invariant conjectures.

E Heuristic ACSL Syntax Gate: Implementation Details

The syntax gate (Stage 1, Section 4.3) runs before any program execution or Frama-C invocation. It is implemented in `unified_filter.py` as a rule-based checker that operates directly on ACSL expression strings.

E.1 Rejection Rules

Each candidate invariant expression is checked against the following ordered rules (any single failure causes rejection):

(1) **Trivial/placeholder tokens.** Expressions equal to `true`, `false`, `...`, or containing only punctuation are rejected immediately.

(2) **Non-printable or non-ASCII characters.** Frama-C requires printable ASCII (U+0020–U+007E). Characters such as the Unicode inequality signs $\leq$, $\geq$, $\neq$, or control characters (e.g. BEL, U+0007 occasionally emitted by LLMs) cause rejection. The error message instructs the model to use `<=`, `!=`.

(3) **Nested `loop invariant` keyword.** When a LLM fills in a `PLACE_HOLDER` slot with a complete `loop invariant expr;` line instead of a bare expression,

Frama-C cannot parse the annotation and enters a pathological parsing state. A regex check (`/\bloop\sinvariant\b/i`) catches and rejects these.

(4) **Bracket balance.** Unmatched parentheses, square brackets, or braces are rejected, with the position of the first mismatch reported in the error message.

(5) **Forbidden quantifiers.** Presence of `\forall` or `\exists` triggers rejection; Frama-C/WP does not support quantifiers in loop invariant annotations.

(6) **Forbidden ACSL definitions.** The keywords `predicate`, `inductive`, `logic`, `axiomatic`, `lemma` may not appear in invariant expressions.

(7) **Forbidden ACSL math operators.** `\product`, `\sum`, `\min`, `\max` are rejected (not supported by the WP plugin for loop invariants).

(8) **Ternary operator.** The `? :` construct is not valid ACSL.

(9) **Undefined variable names.** Each identifier in the expression is compared against *known_vars*, the union of local variables, parameters, and ACSL built-ins extracted from **record**. Unknown identifiers are rejected with a diagnostic listing the candidates.

(10) **`\at` scope check.** Expressions of the form $\texttt{\at}(v, \text{Pre})$ are permitted only when $v \in \mathcal{P}$ (function parameters). Usage on local variables is rejected, since local variables have no pre-state binding in Frama-C/WP.

(11) **Exponentiation (`^` or `\pow`).** A final safety sweep removes any surviving candidates containing these tokens, which represent bitwise XOR or undefined ACSL functions respectively.

E.2 Post-Merge Sanitisation

After the union of N candidate pools is formed (Algorithm 4, line 11), an additional normalisation pass is applied:

- $\texttt{\at}(v, \text{Pre})$ for local variable v is rewritten to plain v (conservative safe fallback).
- Any invariant re-introducing `^` or `\pow` is silently dropped.

This sanitisation step prevents the Houdini stage from encountering annotation parse errors that would stall the Frama-C subprocess.

F Smart Dynamic Sampling: Implementation Details

F.1 Overview

The smart sampler (`smart_sampler.py`) generates concrete integer inputs for the program's function parameters. Its design goal is to produce a small but informationally dense set of execution traces that expose the polynomial relationships maintained by the loop, while avoiding:

- All-zero inputs (which collapse polynomial terms and prevent coefficient identification).
- Excessively large inputs (which cause integer overflow or time-outs during dynamic execution).

F.2 Tiered Value Generation

For each parameter v with domain $[\ell_v, u_v]$ inferred by `LoopBoundAnalyzer`, the sampler constructs four tiers:

$\mathcal{T}_0(v) = \{0, 1, -1\} \cap [\ell_v, u_v]$	(special corner values)
$\mathcal{T}_1(v) = \{2, \ldots, 10\} \cap [\ell_v, u_v]$	(small positive integers)
$\mathcal{T}_{1'}(v) = \{-2, \ldots, -10\} \cap [\ell_v, u_v]$	(small negatives, if enabled)
$\mathcal{T}_2(v) \subseteq \{11, \ldots, 50\}$	(5 randomly drawn medium values)
$\mathcal{T}_3(v) \subseteq \{51, \ldots, 100\}$	(3 randomly drawn large values)

F.3 Phased Cartesian Sampling

Inputs are generated in four phases. Each phase fills part of the quota of at most $S_{\max}$ samples (default $S_{\max} = 20$). Already-seen input tuples are deduplicated via a hash set.

(1) **Phase 1 (corner cases).** Generate all tuples in $\prod_v \mathcal{T}_0(v)$. For p parameters each with up to 3 special values, this produces at most 3^p tuples (e.g. 9 for $p = 2$). These inputs are ideal for fitting degree-0 and degree-1 polynomial terms.

(2) **Phase 2 (mixed tier 0+1).** Enumerate combinations where each coordinate is drawn from $\mathcal{T}_0(v) \cup \mathcal{T}_1(v)$. Combinations are ordered by total tier index $\sum_v \text{tier}(v_j)$, so simpler combinations are generated first.

(3) **Phase 3 (all tiers).** Extend to include $\mathcal{T}_2$ and $\mathcal{T}_3$, again prioritised by total tier index. This covers medium-range inputs needed to disambiguate quadratic from linear relationships.

(4) **Phase 4 (biased random fill).** If the quota is not yet reached, fill with random tuples where each coordinate is drawn with probability 0.8 from $[0, 20]$ and 0.2 from $[21, 100]$. This phase ensures coverage of edge cases not reached by the structured phases.

F.4 Trace Collection and Formatting

Each accepted input tuple is passed to `DynamicExecutorConfigurable`, which instruments the C source with `printf` statements recording all variable values at the top of each loop iteration. The collected trace for a single input is a sequence of variable-assignment strings (e.g. `(n == 2) * (x == 8) * (y == 25) * (z == 18)`). These are parsed and formatted by `PromptFormatter` into the temporal-labelled trace blocks shown in Appendix D.

To prevent context-window overflow, the formatter applies two limits:

- At most $G = 5$ trace groups are included per LLM call.
- Within each group, at most $k_{\max} = 10$ iteration snapshots are kept; if a loop runs for more than $k_{\max}$ iterations, the first $\lfloor k_{\max}/2 \rfloor$ and last $\lceil k_{\max}/2 \rceil$ snapshots are retained, and the truncation is noted explicitly in the prompt.

These limits ensure that the total prompt length stays within the LLM's context window even for programs with deeply nested loops or large parameter ranges.

References

[1] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Proc. Int. Symp. on Formal Methods for Components and Objects (FMCO)*. Springer, 115–137.

[2] Dirk Beyer. 2023. Competition on Software Verification (SV-COMP). https://sv-comp.sosy-lab.org/.

[3] Saikat Chakraborty, Shuvendu K. Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2024. Towards Neural Synthesis for SMT-Assisted Proof-Oriented Programming. In *Proc. Int. Conf. on Learning Representations (ICLR)*.

[4] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *Proc. Int. Conf. on Computer Aided Verification (CAV)*. Springer, 420–432.

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL)*. 238–252.

[6] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948* (2025).

[7] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Write Specifications?. In *Proc. ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE)*.

[8] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. In *Science of Computer Programming*, Vol. 69. 35–45.

[9] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Proc. Int. Symp. on Formal Methods Europe (FME)*. Springer, 500–517.

[10] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. In *Proc. 43rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*. 499–512.

[11] Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen Carbonell, et al. 2024. RLEF: Grounding Code LLMs in Execution Feedback with Reinforcement Learning. *arXiv preprint arXiv:2410.02089* (2024).

[12] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 206–215.

[13] Jesse Michael Han, Floris Rabe, Yuhuai Wang, Christian Szegedy, and Christian Szegedy. 2022. Proof Artifact Co-Training. In *Proc. Int. Conf. on Learning Representations (ICLR)*.

[14] Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. 2023. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *Proc. Int. Conf. on Learning Representations (ICLR)*.

[15] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding Inductive Loop Invariants using Large Language Models. *arXiv preprint arXiv:2311.07948*.

[16] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A Software Analysis Perspective. In *Proc. Int. Conf. on Software Engineering and Formal Methods (SEFM)*. Springer, 233–247.

[17] Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengding Huang, Hamish Ivison, Faeze Brahman, Lester James Validad Miranda, Valentina Pyatkin, et al. 2024. TÜLU 3: Pushing Frontiers in Open Language Model Post-Training. *arXiv preprint arXiv:2411.15124* (2024).

[18] Guillaume Lample, Timothée Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. 2022. HyperTree Proof Search for Neural Theorem Proving. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*.

[19] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*.

[20] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-Level Code Generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097.

[21] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's Verify Step by Step. In *Proc. Int. Conf. on Learning Representations (ICLR)*.

[22] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using Dynamic Analysis to Discover Polynomial and Array Invariants. In *Proc. 34th Int. Conf. on Software Engineering (ICSE)*. 1076–1086.

[23] Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Viber. 2015. Efficient Synthesis of Probabilistic Programs. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 208–217.

[24] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training Language Models to Follow Instructions with Human Feedback. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*.

[25] Kexin Pei, David Biber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants?. In *Proc. Int. Conf. on Machine Learning (ICML)*.

[26] Sorin Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. 2023. Formal Mathematics Statement Curriculum Learning. In *Proc. Int. Conf. on Learning Representations (ICLR)*.

[27] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2023. Direct Preference Optimization: Your Language Model Is Secretly a Reward Model. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*.

[28] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Non-linear Loop Invariant Generation Using Gröbner Bases. In *Proc. 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*. 318–329.

[29] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*. 7751–7762.

[30] Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Christian Szegedy, and Christian Szegedy. 2022. Autoformalization with Large Language Models. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*.