A Appendix

A.1 Loop Invariant Refinement

To address the trade-off between accuracy and diversity in invariant generation, we propose a fine-grained iterative refinement framework (Algorithm 1) that combines verifier feedback with the LLM's self-correction ability. This framework is motivated by the observation that LLMs often produce loop invariants that are inaccurate or too weak, and that templates, while improving precision, reduce diversity.

During the refinement process, depending on the verifier's feedback, different strategies are applied: Syntax errors are repaired directly, weak invariants (passing Base and Preservation but not given Postcondition) are strengthened, invariants failing Base or Preservation but still implying the Postcondition are weakened or adjusted, and invariants failing all checks are discarded and regenerated. At each step, the LLM is guided by a specialized prompt that encodes these refinement strategies. If repeated attempts fail, a final elimination step removes consistently invalid invariants. The refinement continues until a correct invariant is achieved or an iteration limit is reached. Similarly, the refinement of a function specification can apply these strategies.

A.2 Full Prompt

Figure 1 guides the large language model to reason about the corresponding loop in natural language by answering a series of questions. This process builds the model's understanding of the loop and aids subsequent generation. Figure 2 illustrates a comprehensive prompt design used to guide LLMs in generating formal loop invariants in ACSL, within a program verification context. Figure 3 clearly defines a LLMs' role as a C and Frama-C expert, outlining specific tasks to repair, adjust, strengthen, or regenerate ACSL loop invariants based on Frama-C's verification feedback.

Algorithm 1: Iterative Loop Invariant Refinement LoopInvRefinement

Input: $\mathcal{F}$ with candidated loop invaraints $\mathcal{I}_0$, *LLM*
Output: $\mathcal{F}^*$ with verified invariants $\mathcal{I}^*$

```
I ← I_0;
F⁻ ← F − I_0
// Run verifier and collect errors
while Errors ← ¬Verify(F) do
    if reach iteration limit then
        // Find largest valid subset in I
        while I remains invalid do
            foreach e ∈ Errors do
                i ← associate loop invariant with e;
                I ← Elimination(I, i);
        F ← (F⁻, I);
        break;
    if Errors = syntax error then
        I ← RepairCallLLM(I);
    else
        guidance ← {} ;
        if Errors = verfication goals fail then
            // Only termination failed
            guidance ← guidance ∪ strengthGuide(I);
        else
            foreach e ∈ Errors do
                if e = loop invariant error then
                    i ← associate loop invariant with e;
                    if e not base then
                        guidance ← guidance ∪ WeakenGuide(i);
                    if e not preservation then
                        guidance ← guidance ∪ AdjustGuide(i);
                else
                    guidance ← guidance ∪ RegenGuide(i);
    I ← RefineCallLLM(I, guidance);
    F ← (F⁻, I);
return F
```

Prompt For Think In Natural Language

Role:
You are a C language static analysis expert. Your primary task is to formally verify C code by performing a detailed analysis of its behavior, with a specific focus on loop invariants.

Task: Loop Verification Analysis
Given a C code snippet, you must produce a comprehensive analysis covering the loop's properties, invariants, and pre/post-conditions.

```c{c_code}```

Your analysis must be structured with the following sections:

a. Loop Purpose and Structure
- Explain the purpose and intended outcome of the loop in natural language.
- Describe the loop's structure: its governing condition (`while(...)`), the operations performed in its body, and all variables relevant to its behavior.

b. Sample and Analyze Variable Values
- Pre-Loop Sampling: Before the loop begins, take the very first sample of all variables.
- Post-Iteration Sampling: After the first iteration of the loop body is complete, take a second sample. This process should be repeated for a total of five post-iteration samples (after iterations 1, 2, 3, 4, and 5).
- Post-Loop Sampling (if applicable): If the loop terminates within or after the five iterations, take a final sample immediately upon exiting the loop.

d. Loop Invariant Discussion
- The loop invariant must be true at the beginning and end of every loop iteration you sampling.
- Propose a valid loop invariant in natural language.
- Provide a detailed explanation of why this invariant is valid.

e. Establishment
- Explain how the proposed invariant is established.
- Describe how the given pre-condition guarantees that the invariant holds true before the first iteration of the loop.

f. Preservation
- Explain how the invariant is preserved.
- Demonstrate that if the invariant holds at the beginning of an iteration and the loop condition is true, it will still hold true at the end of that iteration.

g. Termination Analysis
- Identify the state of all relevant variables when the loop terminates (i.e., when the loop condition becomes false).
- Explain why the loop invariant remains valid under these termination conditions.

h. Post-condition Correctness
- Evaluate the provided post-condition. State whether it is correct or not.
- Explain how the invariant, in conjunction with the negation of the loop condition, proves that the post-condition is met.

Fig. 1. Prompt for Think in Natural Language
```

Prompt For Loop Invaraint Generation

Role:
You are a helpful AI software assistant specializing in reasoning about code behavior.
Your task is to analyze C programs and identify loop invariants that can be used to verify program properties using Frama-C.

Task:
Given a C program with a loop, generate the necessary loop invariants in ACSL (ANSI/ISO C Specification Language) annotations. These invariants will help Frama-C verify the post-condition of the program.
A loop invariant is a condition that is true at the beginning and end of every loop iteration.
A loop invariant must satisfy the following conditions to be inductively invariant:
- Establishment: The invariant must be true before the loop begins execution.
- Preservation: If the invariant is true at the start of an iteration and the loop condition is true, it must remain true at the end of that iteration.
- Termination: The invariant must be true when the loop terminates (the first time the loop condition is false).
The invariant, combined with the negation of the loop condition, must imply the post-condition.

Examples:
You must use these follow examples as a reference to complete the task, with the following requirements:
- You may directly use the predicates or functions defined in these examples.
- You may refer to the patterns or ideas from these examples to create new predicates or functions.
- You may use the invariant generation logic from these examples as a guide for your own invariant.
``` {examples} ```

Inputs:
- The pre-condition before the loop begins execution.
- A full C loop program with invariant annotations containing `PLACE_HOLDER` that need to be filled.

Outputs:
Provide the same complete C loop program with invariant annotations where all `PLACE_HOLDER` are filled in within a ` ```c ``` ` block.

Rules:
- Only use keywords and constructs supported in ACSL annotations for loops.
- Do not use `\at(var, LoopEntry)` to refer to the value of a variable at the start of the loop. Instead, use the value specified in the pre-condition.
- Do not add any natural language explanations after ACSL annotations.
- When `unkonwn()` used as the loop condition, the number of loop iterations can be any non-negative integer, and the invariant must hold for all cases.
- Do not modify the structure or wording of the existing annotations. You are only allowed to fill in the placeholders `PLACE_HOLDER_TO_FILL` Before loop with appropriate logical expressions to make the invariants meaningful and valid
- Generate loop invariants with equality constraints as comprehensively as possible.
- If the invariant you need requires a logical function or a predicate, please fill `PLACE_HOLDER_PREDICATE_OR_LOGIC_FUNCTION`
- Please first try to directly use the verification goal as the loop invariant at `PLACE_HOLDER_VERFICATION_GOAL`. Often, the verification goal (assertion) also holds throughout the loop; in that case, it can be used directly as the invariant.

Consider the following C loop:
Pre-condition: `{pre_cond}`
Loop program: `{content}`

Fig. 2. Prompt for Loop Invariant Generation
```

Prompt For Loop Invaraint Refinement

Role:
You are an expert in the C language and the Frama-C static analysis tool.

Task:
(repair) Your task is to correct syntactically incorrect ACSL annotations in a given C program, using the provided error messages from Frama-C.
(adjust) Your task is to fix an incorrect loop invariant in a given C program based on the provided error messages from Frama-C.
When the "Goal Assertion" is correct, but "Goal Preservation" is incorrect, it means the current loop invariant can verify the postcondition but is
flawed. You need adjust the invariant to make sure it remains valid after each iteration and holds at the end of the loop.
(weaken) Your task is to fix an incorrect loop invariant in a given C program based on the provided error messages from Frama-C.
When the "Goal Assertion" is correct, but "Goal Establishment" is incorrect, it means the current loop invariant can verify the postcondition but is flawed. You need weaken the invariant to be valid under initial preconditions.
(strengthen) Your task is to fix an incorrect loop invariant in a given C program based on the provided error messages from Frama-C.
If only the "Goal Assertion" is incorrect, it indicates that the current loop invariant is correct but not strong enough. You need to strengthen it or add new invariants to ensure the postcondition can be verified.
(regeneration) Your task is to regenerate incorrect loop invariants in a given C program based on the provided error messages.
A loop invariant is a condition that holds true at the beginning and end of every iteration of a loop.
For a loop invariant to be inductively valid, it must satisfy the following conditions:
1. Establishment: The invariant must be true before the loop starts executing.
2. Preservation: If the invariant is true at the beginning of a loop iteration and the loop condition is true, it must remain true at the end of that iteration.
3. Termination: When the loop terminates (the loop condition becomes false for the first time), the invariant, combined with the negation of the loop condition, must imply the post-condition.
When the "Goal Assertion," "Establishment," and "Preservation" are all incorrect, it signifies that the current loop invariant is fundamentally wrong. You need to regenerate the entire loop invariant, ensuring that the postcondition can be verified. You are only allowed to regenerate the incorrect invariant.

Inputs:
1. Error List: ```{error_str}```
2. C Code with Incorrect ACSL Annotations: ```c{c_code}```

Outputs:
1. Error Analysis:
    Provide a detailed analysis of the error and the rationale behind your modification.
2. Fixed C Code:
    Provide the complete corrected C code with the fixed ACSL annotations, based on the error message and the incorrect annotations in the input C code.

Rules:
1. Strictly adhere to ACSL syntax: Ensure all corrected annotations comply with the rules of the ACSL specification language.
2. Do not modify the original C code: Only make changes to the ACSL annotations.
3. Do not add any natural language explanations after ACSL annotations

Fig. 3. Prompt for Loop Invariant Refinement