Guiding Likely Invariant Synthesis on Distributed Systems with Large Language Models

Yuan Xia, Aabha Shailesh Pingle, Deepayan Sur, Srivatsan Ravi, Mukund Raghothaman, Jyotirmoy V. Deshmukh
University of Southern California
Los Angeles, USA
{yuanxia, apingle, deepayan, srivatsr, raghotha, jdeshmuk}@usc.edu

***Abstract*—Deductive verification is an approach where the user expends effort in constructing a program invariant manually, or learns this invariant from program traces, and subsequently uses the program invariant for safety verification tasks. Most existing invariant synthesis tools still rely on model checking methods to certify invariants, use enumerative search to identify and refine candidate invariants, and user insights to start with a set of candidate predicates that may appear in the final (learned) invariant. On the other hand, pre-trained large language models (LLMs) have recently attracted considerable attention in various tasks related to generating program code (e.g., from natural language). There have been recent efforts at applying LLMs to the program verification context. In this paper, we investigate the capabilities of LLMs to synthesize program invariants by creating a specialized set of prompts. When LLM invariant synthesis with direct prompting fails, we introduce two revision frameworks for incorporating LLM calls. In the first framework PSyn, the LLM suggests atomic predicates based on counterexamples for reconstructing the invariants. The second framework ISyn adopts the LLM to generate invariants by itself within a revision process. We tested our methods on 8 distributed systems modeled in Promela and invariants are verified with Spin. Our results show that the state-of-the-art model GPT-o3, when it is used to generate invariants directly, is less effective than the symbolic invariant synthesis method RunVS. However, our integrated approach PSyn, which employs the LLM as a predicate prompter, significantly outperforms GPT-o3 and ISyn and has comparable performance with RunVS. This integrated technique also produces higher-quality invariants in general. Because we focus on runtime monitoring frameworks, we primarily consider system traces and hence likely invariants. However, our revision frameworks can also learn true invariants by using model checkers in the revision frameworks.**

***Index Terms*—Invariant Synthesis, Distributed Systems, Large Language Models**

I. Introduction

Formal verification of distributed computing systems is a computationally challenging problem, as the number of system states grows exponentially both in the number of program variables, and the number of processes participating in the distributed computation. This state-space explosion problem has long been a roadblock for automatic verification techniques like model checking, though approaches like partial order reductions, symmetric reductions, symbolic methods with SMT solvers, and context-bounding have sought to ameliorate this problem. An alternative approach to exhaustive exploration of the state-space is deductive verification, where the user proposes an invariant expression over the (global) program variables, which if valid, can help simplify proofs of system safety to a simple validity check of a propositional formula.

As manually deriving invariants is difficult, there has been considerable effort to automatically synthesize program invariants. There are two primary methods for this: deductive and automatic methods. Deductive methods involve manual or semi-manual proofs aided by theorem provers such as Coq [1] and Isabelle [2], and involve interaction with a proof system with explicit axiomatization of the program syntax and semantics. Deductive methods [3], [4] are able to capture complex invariants, but require significant human labor and deep domain knowledge. Automatic methods rely on tools such as model checkers or dynamic invariant detectors to learn invariants.

Model-checker-assisted invariant synthesis methods iteratively refine candidate invariants based on counterexamples generated by model checkers. Methods based on ICE-learning [5], [6] use decision trees to classify positive examples obtained from program traces from negative examples obtained from user-provided safety properties and implication counterexamples obtained from program transitions to enforce inductiveness of the invariant. Invariant verification is performed using the Z3 SMT solver. DistAI [7] leverages Ivy [8] as a model checker to ensure inductiveness of the invariant. While these approaches can give deterministic guarantees on the correctness of the invariant, they may face computational challenges in validating the invariant for large or infinite-state systems.

Dynamic invariant synthesis techniques infer invariants from just execution traces. Tools such as Daikon [9] collect runtime observations and detect likely invariants from them. DIDUCE [10] tracks certain types of invariants of Java programs. While dynamic methods may overlook rare edge cases and produce overly broad or imprecise invariants, they are generally faster and more scalable in predicting invariants from system traces. Most automatic invariant synthesis methods require user input in the form of annotations (e.g., in the form of assertions), identification of program variables that should be included in the invariant, and initial sets of predicates. In this paper, we will focus on automatic methods, while trying to both alleviate the burden imposed on users and scaling the process of validating the synthesized invariant.

A step in this direction was taken by our prior work [11] which developed the RunVS technique for dynamic invariant

https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_29

learning. RunVS learns decision tree-based invariants to separate positive examples (obtained from execution traces) from speculative negative examples. The approach uses a runtime monitoring technique to validate the learned invariant. As it does not provide deterministic guarantees on the learned invariant, we call its output a *likely invariant*. However, there are a number of limitations with RunVS:

1) RunVS requires the user to provide a list of global variables that may be included in the invariant.
2) RunVS requires a grammar for the predicate augmentation.
3) In many distributed systems, some global variables are indexed by the processes that update or read from them, and invariants can be quantified expressions such as $\forall i \exists j : P(x_i, y_j) \implies Q(x_i)$, where P and Q are predicates involving two variables and a single variable, respectively [7].

To address these challenges, we propose two frameworks that integrate Large Language Models (LLMs) into the invariant synthesizer RunVS. When we prompt the LLM with the source code for a distributed protocol along with other carefully crafted queries, the LLM can suggest predicates, reducing the space of predicates to be explored. When prompted with counterexamples and some of the previous context, it can revise invariants. In RunVS, we iteratively learn the invariant by alternating invariant inference (from examples) and runtime monitoring to generate counterexamples. Any counterexamples found are used to refine the invariant. We use the same basic framework, but use LLM-guidance at several steps during this process.

Contributions. To summarize, this paper makes the following contributions:

(i) We investigate prompt design for LLMs to discover and refine predicates that may appear in a program invariant.
(ii) We mitigate the problem of exploring a large space of predicates by combining LLMs with the RunVS invariant revision framework.
(iii) We boost LLM's predicate prediction capability with chain of thought reasoning.
(iv) We extend our previous RunVS approach to synthesize quantified invariants by using the LLM to replacing quantifier-free expressions with inferred quantifiers.
(v) We demonstrate the usefulness and scalability of our framework on distributed systems modeled in Promela [12].

II. Preliminaries

In this section, we formalize the terminology needed to explain our approaches. Besides, we will present the problem statement in this section.

A. Program structure and modeling assumptions

Program variables, program states, program execution traces, and invariants are all fundamental concepts in program synthesis. We give their definitions in this section so that we can precisely establish our problem statement.

Definition 1 (Variables, States)**.** *A type t refers to a finite or an infinite set of values. A program variable v of type t represents a symbolic name that contains values from t, denoted by $\texttt{type}(v) = t$. Let V represent the complete set of program variables. A* valuation $\nu : V \to \bigcup_{v \in V} \texttt{type}(v)$ *is a function that assigns to each variable $v \in V$ a concrete value $\nu(v) \in \texttt{type}(v)$. The* program state *at any execution point is denoted by the tuple $\nu(V)$, representing the collective valuation of all variables in V.*

Definition 2 (Labelled Transition System (LTS))**.** *A* labelled transition system *for a concurrent program is a tuple (S, L, T, Init), where:*
S is a set of system states,
L is a set of labels corresponding to program statements,
$T \subseteq S \times L \times S$ is the transition relation,
$\mathsf{Init} \subseteq S$ is a set of initial states.

Definition 3 (Reachable States)**.** *For a program $\mathcal{P}$ modeled as an LTS, its* reachable states *R are all states that can be reached from initial states s_0, $s_0 \in S$. $\mathsf{Reach}(\mathcal{P}, \mathsf{Init})$ is the minimal set R satisfying:*
1) $s_0 \in \mathsf{Init} \implies s \in R$ and,
2) $s \in R \wedge (s, s') \in T \implies s' \in R$.

Definition 4 (Program Trace)**.** *A* program trace *$\sigma = (s_0, s_1, \ldots, s_k)$ is a finite sequence of states where:$s_0 \in \mathsf{Init}$, and for all $j \in [1, k-1]$, $(s_{j-1}, s_j) \in T$.*

Definition 5 (Monitor)**.** *A* monitor *observes a finite prefix $\sigma = (s_0, \ldots, s_k)$ of a program trace and evaluates whether σ satisfies a property ϕ.*

Runtime monitoring refers to monitoring the system's states during its execution and detecting whether the behaviors align with predefined specifications. Runtime monitoring is a popular approach for dynamic verification [13]. The *offline monitoring* [14] collects runtime information during the system's execution and stores it for later analysis. The analysis is performed offline once the system has completed execution. In contrast, *online monitoring* [14] refers to the simultaneous observation and analysis of system states as it is running. The concurrent monitor enables early detection and response, which is essential for critical systems requiring immediate corrective actions.

Definition 6 (Invariants)**.** *An invariant I is a Boolean-valued expression over program variables that holds for all reachable program states, formally defined as follows:*

$$\forall s \in \mathsf{Reach}(\mathcal{P}, \mathsf{Init}),\ s \models I.$$

Remark. Likely invariants are hypothesized invariants that hold true for all states observed up to a given point.

B. Basics of LLM-based analysis

Generative Large Language Models (LLMs) are transformer-based neural networks pre-trained on large text/code corpora to generate human-like outputs. LLMs leverage self-attention mechanisms to capture long-range

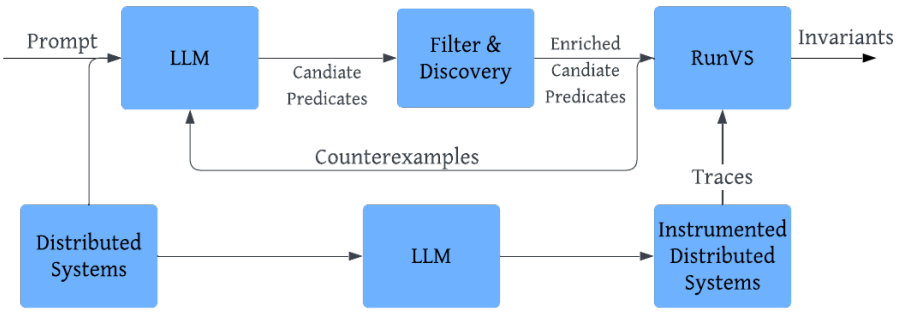

Fig. 1: Predicate LLM Synthesis Framework

dependencies in sequences. With fine-tuning, they are adaptable to domain-specific tasks, e.g., code synthesis [15]. In this work, LLMs act as assistants for invariant synthesis, helping revise predicates or invariants. Few-shot prompting (learning) is a form of prompt engineering in generative AI, where a few examples are provided in a prompt to perform a task.

Zero-shot prompting refers to the method of no examples given, and one-shot prompting means a single example crafted in a prompt to achieve a desired output. One-shot prompting is particularly useful when collecting a large amount of training data is impractical. We will use one-shot prompting to perform the task of invariant synthesis, denoted as one-shot invariant synthesis, as our baseline experiments.

Problem statement. Let $\mathcal{P}$ be a distributed program. Let $\mathsf{Reach}(\mathcal{P}, \mathsf{Init})$ be the set of reachable states of $\mathcal{P}$. Let G denote the grammar to specify (a possibly infinite) set of Boolean-valued expressions over program variables, and let $\mathcal{L}(G)$ denote this set. The objective of this paper is to design a runtime algorithm able to learn a program likely invariant ϕ without dependency on users by integrating LLMs. The invariant has the following properties:

1) Soundness of a likely invariant:

$$((s \in \mathsf{Reach}(\mathcal{P}, \mathsf{Init})) \Rightarrow (s \in \phi)) \tag{1}$$

2) Tightness of a likely invariant:

$$\phi = \underset{\phi \in \mathcal{L}(G)}{\arg\min} \; |\{s \mid s \notin \mathsf{Reach}(\mathcal{P}, \mathsf{Init}) \wedge s \in \phi\}| \tag{2}$$

III. Overview

In this section, we will first introduce our code instrumentation technique for tracking state changes of a distributed system simulated by Spin [16]. We then present two runtime monitoring frameworks integrating revision guidance for invariant synthesis from LLMs.

LLM-aided Code Instrumentation. Spin can monitor the state changes of a system modeled in Promela by capturing the system states after each system transition. However, its default approach treats an atomic block consisting of multiple statements, not as a single, simultaneous event, but as individual statements. Thus, Spin does not record global states after atomic blocks. In other words, atomicity is not held in Spin. The traces do not accurately show the states captured "before" and "after" snapshots of atomic blocks. We propose a refined instrumentation strategy that encodes a Promela system into an instrumented Promela system that is trackable by Spin, as shown in Figures 1 and 2. The implementation leverages an LLM to address this limitation. Our technique automatically inserts instrumentation code after every assignment outside atomic blocks while only considering one state update at the end of every atomic block when assignment statements are inside. The LLM produces updated Promela code given a sequence of required instrumentation tasks and constraints in the natural language specification. This method offers an automatic solution for capturing intermediate state transitions without sacrificing the atomic semantics of systems modeled by Spin. We will adopt Peterson [16] Promela program as a demonstration example throughout all prompting techniques. Peterson is a tiny concurrent algorithm that implements mutual exclusion for two concurrent processes.

In this paper, we introduce two frameworks for runtime invariant synthesis that integrate revision guidance from LLMs. Both frameworks rely on iterative refinement within runtime monitoring. The frameworks are supposed to detect incorrect invariants by observing system traces and refining them based on feedback, which is system states in observed traces that do not satisfy candidate invariants, which are denoted as counterexamples. However, two frameworks are distinguished regarding how an LLM is employed and what task the LLM is assigned.

Predicate LLM Synthesizer Framework. As shown in Figure 1, the first framework treats the LLM as a predicate hinter, which is considered a component responsible for supplying promising atomic predicates that are precise and potentially exist in the system invariants. Example prompts are presented in the section IV. We iteratively prompt the LLM with stored predicates from the previous iteration, the falsified invariant, and counterexamples observed from the system's runtime monitoring. Counterexamples are the system states that make the candidate invariant fail. We maintain a certain length of chat history as part of the context in LLM's input. This approach ensures that the LLM's context remains focused on

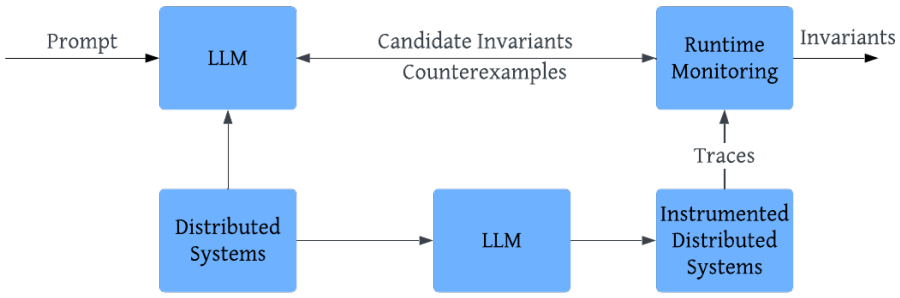

Fig. 2: Invariant LLM Synthesis Framework

relevant information, preventing it from exceeding the LLM's context window limit and maintaining a reasonable prompt length.

The new atomic predicates proposed by the LLM might augment the stored predicates. The stored predicates are increasingly augmented and refined over iterations. RunVS will construct an invariant that is sound and precise based on the input of predicates. Candidate invariants are expected to hold for every system trace. If the invariant fails to hold, the system extracts the state as a new counterexample, updates the counterexample set, and prompts the LLM again. This Predicate Prompter framework effectively offloads the challenging task of demanding potentially useful predicates from the user end to an LLM while keeping the invariant generation and monitoring rigorous through formal synthesis. We further filter out invalid predicates and enrich valid predicates, which is explained in the next section. Unlimited positive states will eventually exceed the context window of the LLM. Therefore, we focus on the validated predicates and a certain number of counterexamples during prompting.

Invariant LLM Synthesizer Framework. The self-revised LLM framework, as shown in Figure 2, adopts an LLM to generate entire invariants and iteratively refining them. In this framework, the LLM produces complete candidate invariants rather than atomic predicates. Example prompts are shown in section V. During the monitoring, once counterexamples are observed and a revision is triggered, these counterexamples are fed back to the LLM and prompted to revise the proposed invariant regarding the detected error states. Instead of incrementally adding predicates, this method explores the feasibility of the LLM's capability to generate and revise invariants by itself iteratively. In this framework, an LLM is guided by prior counterexamples and the information of partial successes of the candidate invariant in the previous iteration. While potentially more flexible, the self-revised LLM framework also leads to more concerns, e.g., the soundness and tightness guarantee of the guessed invariants.

Comparisons. The predicate LLM synthesizer framework builds invariants formally based on predicates, which reduces LLM's burden of struggling with invariant expression format correctness and logic correctness. We allow an LLM integration but still rely on RunVS to construct likely invariants, which are sound and precise in terms of the observed states. The invariant LLM synthesizer framework aims for a more holistic view, which may converge quickly if the LLM can combine multiple relevant constraints effectively. However, it can also require more sophisticated prompt engineering to avoid losing key information and giving too general invariants. Both frameworks consider incorporating LLM-assisted invariant generation with counterexample-guided refinement.

IV. Predicate LLM Synthesizer

As shown in Algorithm 1, PSyn collects counterexamples and samples a certain number of negative states in each iteration at Line 7, and ensures the recall is 1 and the precision is higher than a predefined threshold at Line 8. Otherwise, a revision will be triggered to learn a sound and more precise invariant. As Line 10, our predicate LLM synthesizer is fed with predicates, counterexamples, and the previous candidate invariant. It is responsible for drafting a new set of candidate predicates P in each required revision.

For an ablation study, we have included three types of prompting techniques, i.e., zero-shot, one-shot, and chain-of-thought (CoT) prompting, for testing and improving the inference capability of the LLM. For zero-shot prompting, in the first iteration, we supply the LLM with the full list of global variables, a Promela program, and the acceptable operators, e.g., +, -, ==, <=, and, or, true. For one-shot learning , we adopt the Peterson promela program along with an answer of predicates as one demonstration example, which is inserted into the prompt body. This biases the model towards catching the correct syntax and semantics of predicates from a given code. For CoT prompting, we prepare a reasoning template instructing the model to "think aloud" of predicates from the following five aspects:

1) Assertions embedded in the Promela model,
2) Branch conditions, e.g., guards in *if, do, and goto* statements,
3) Template predicates obtained from the above information,
4) Analysis of observed system states combined with variable types to infer variable ranges and replace template predicates with concrete values, e.g., $0 \leq ncrit \leq 1$,

Algorithm 1: PSyn

input : Distributed/Concurrent program $\mathcal{P}$
The number of points randomly sampled l
The precision threshold δ
Maximum depth of decision tree k
$\frac{|\text{speculated}|}{|\text{reached}|}$ threshold α
Total runs to monitor MaxTraces
LLMs generate or user input atoms
Max runs to confirm final invariant
MonitorBudget
output: likelyInv ϕ

1 reached $\leftarrow \{\}$, speculated $\leftarrow \{\}$, $\phi \leftarrow \mathit{false}$, $m \leftarrow 1$, $n \leftarrow 0$
2 **repeat**
3 $T \leftarrow$ SampleTrace($\mathcal{P}$) // sample a random trace
4 ce $\leftarrow \{s \mid s \in T \cap [\![\neg\phi]\!]\}$ // reached state in $\neg\phi$
5 ce $\leftarrow$ ce $\cup \{s \mid s \in T \cap$ speculated$\}$ // reached state assumed unreachable
6 reached $\leftarrow$ reached $\cup$ ce // update reached states
7 speculated $\leftarrow$ (speculated$\backslash$ce) $\cup$ randomSample($S \backslash$ ((speculated$\backslash$ce) $\cup$ reached), ℓ)
8 s.t. $\frac{|\text{speculated}|}{|\text{reached}|} < \alpha$
9 **if** ce $\neq \emptyset$ **then**
10 predicates $\leftarrow$ LargeLanguageModel(atoms, ce, ϕ)
11 atoms $\leftarrow$ PredicateDiscovery(BreakDown(predicates))
12 $\phi \leftarrow$ Learner(atoms, reached, speculated, d, k, δ)
$n \leftarrow 0$
13 **else** $n \leftarrow n + 1$
14 $m \leftarrow m + 1$
15 **until** $m \geq$ MaxTraces $\vee\, n \geq$ MonitorBudget
16 **return** ϕ

5) Counterexamples used to find new satisfying predicates to update the current predicate set.

We include only the validated predicates and the latest counterexamples so that the prompt length stays below the model's context limit. The model returns a list of candidate predicates $P = \{p_1, \ldots, p_k\}$. p_i is a Boolean expression. We apply predicate breakdown at Line 8 to enrich more predicates, where we parse the predicates into sub-clauses that can serve as atomic predicates. To ensure the format correctness of the generated predicates, we use a predicate filter function to filter out predicates in a wrong syntax. Besides, PredicateDiscovery at Line 8 enriches more concrete predicates by extracting the predicate patterns and replacing them with concrete values detected from system traces. The predicates coming from PredicateDiscovery will be the final predicates fed into the decision tree invariant learner Learner.

Algorithm 2: ISyn

input : Distributed/Concurrent program $\mathcal{P}$
The number of points randomly sampled l
The precision threshold δ
Max runs to confirm final invariant
MonitorBudget
output: Revised invariant ϕ

1 **repeat**
2 $T \leftarrow$ SampleTrace($\mathcal{P}$) // sample a random trace
3 ce $\leftarrow \{s \mid s \in T \cap [\![\neg\phi]\!]\}$ // reached state in $\neg\phi$
4 ce $\leftarrow$ ce $\cup \{s \mid s \in T \cap$ speculated$\}$ // reached state assumed unreachable
5 reached $\leftarrow$ reached $\cup$ ce // update reached states
6 speculated $\leftarrow$ (speculated$\backslash$ce) $\cup$ randomSample($S \backslash$ ((speculated$\backslash$ce) $\cup$ reached), ℓ)
7 s.t. $\frac{|\text{speculated}|}{|\text{reached}|} < \alpha$
8 **if** ce $\neq \emptyset$ **then**
9 $\phi \leftarrow$ LargeLanguageModel(ce, ϕ)
10 $n \leftarrow 0$
11 **else** $n \leftarrow n + 1$
12 $m \leftarrow m + 1$
13 **until** $m \geq$ MaxTraces $\vee\, n \geq$ MonitorBudget
14 **return** ϕ

A. Expansion to LLM-Guided Quantifier Synthesis

We also integrated an LLM for the expansion to quantified invariants. LLM is used to automate two critical tasks, i.e., detecting quantifiable variables and generalization to quantified invariants. Firstly, the LLM analyzes the Promela source code with a structured prompt to recognize quantifiable variables. LLM is able to identify the quantifiable variables and their names, types, and the corresponding domains of the variables. For instance of the distributed lock system [16], the LLM identified one quantifiable global variable $\mathit{current\_holder} \in \{-1, 0, \ldots, N-1\}$. Secondly, given a candidate invariant proposed by the symbolic synthesizer RunVS, the LLM synthesizes a quantified formula by replacing enumerative clauses, e.g., $\mathit{current\_holder} = 0 \vee \mathit{current\_holder} = 1 \vee \cdots$, with semantically equivalent quantified expressions, e.g., the existential quantifier $\exists\ \mathit{pid}\ (0 \leq \mathit{pid} < N \wedge \mathit{current\_holder} = \mathit{pid})$. For the distributed lock system, the LLM produces $(\mathit{lock\_available} \wedge \mathit{current\_holder} = -1) \vee (\neg \mathit{lock\_available} \wedge \exists \mathit{pid}(0 \leq \mathit{pid} < N \wedge \mathit{current\_holder} = \mathit{pid}))$. This generalization preserves the logic of invariants and tends to generate more precise invariants. This step enables reasoning across arbitrary system scales by eliminating explicit dependence on N and enables the length optimization for human understanding.

V. Invariant LLM Synthesizer

Our second framework, ISyn, assigns the invariant generation and revision task to an LLM to replace the decision tree learner in RunVS. Unlike PSyn, this LLM treats an invariant as a single boolean expression that is regenerated after every

revision. The framework is shown in Algorithm 2. We still adopt structured prompts and the ablation study for better instruction. We included the previous three chat histories and subsampled a fixed number of counterexamples to prompt the LLM. We only prompt the LLM with the latest counterexamples to avoid exceeding the context window limit, shown at Line 9. Before accepting the LLM-generated invariant, we run a syntactic check to ensure the format correctness of the invariant. We employed zero-shot learning, one-shot learning, and CoT prompting for the ablation study of the prompting in the second framework. For the zero-shot prompt, we directly prompt LLM to produce an invariant that always holds given the Promela program. With a one-shot prompt we provide the Peterson Promela code and its corresponding correct invariant as a demonstration example. For CoT prompting, besides the aspects described in the previous section, we design thinking steps that involve invariant inference and invariant revision.

Model-checker validation. Our frameworks can accept feedback from a model checker to prompt the LLM. Spin is able to verify an invariant if it is sound. If Spin confirms the soundness, the loop terminates. Otherwise, a reproducible trace with the violated state is returned. The next revision prompt includes the verification result, the error state, and the last monitored invariant. Our LLM synthesizers are able to revise invariants or predicates based on the model checker's verification results.

VI. Experimental Evaluation

A. Benchmarks and Measurements

To investigate our research goal, experiments were designed to address the following three research questions:

- **RQ1**: Do different prompting techniques affect the overall performance of runtime invariant generation?
- **RQ2**: Do PSyn and ISyn generate the likely invariants of higher quality than those generated by GPT-o3 and RunVS by runtime monitoring of these real-world distributed systems?
- **RQ3**: How PSyn and ISyn efficiently and effectively learn likely invariants on large-scale distributed systems?

We employ *gpt-3.5-turbo-0125* [17] as the component within our PSyn and ISyn frameworks. We configured the LLM API with temperature $T = 1$ to balance deterministic output with randomness and diversity. Counterexample sampling was constrained to 5 examples per iteration, optimizing the trade-off between context window utilization and informativeness. We adopt RunVS [11] as our symbolic invariant synthesizer, which is the only tool able to learn diverse invariants on Spin systems within a runtime monitoring framework. As baseline experiments, we evaluate invariants that are purely generated by RunVS within 500 traces. Besides, we adopt GPT-o3 [17] among the state-of-the-art models. We perform one-shot invariant synthesis by GPT-o3 with the context of program code but without a revision framework.

We adopt three metrics for evaluating the quality of learned invariants. The first two are soundness and tightness, whose definitions are defined in the problem statement. Soundness is checked by Spin with the corresponding system. Tightness is evaluated by counting the assignments of a boolean expression Z3 SMT solver. Tighter invariants indicate that the invariants are more concise. The third one is safety, which can be considered as proof of invariant usefulness. Safety is evaluated by implication checking between the invariant i and the safety property ϕ such that $i \rightarrow \phi$.

Our benchmarks include 8 distributed systems modeled in Promela, drawn from resources related to Spin and other distributed systems literature. Our frameworks PSyn and ISyn are general to other languages. Promela is chosen since it represents interesting finite-state, infinite-state, asynchronous message-passing-based distributed systems and is the input to the Spin model checker, which gives us fine-grained control over trace generation.

B. Prompt Engineering Ablation Study

In Tables I and II, we compared the performance of three prompting techniques, i.e., direct prompting, one-shot prompting, and chain-of-thought prompting [18]. In each case study, if a synthesized likely invariant is verified by Spin or implies the safety property, it is indicated by ✓in tables. The tightness is evaluated by calculating the model counts by Z3 solver [19]. Since we adopted Peterson as our demonstration example in our 1-shot and CoT prompting, Peterson was not experimented with using those two prompting techniques. We can see CoT tends to learn a sound and more precise invariant since LLM adopts reasoning to give more diverse predicates. Besides, one-shot and CoT can give a more useful invariant since their generated invariants mostly imply the safety property of the corresponding system. CoT especially helps ISyn to break down the whole reasoning steps of invariant synthesis compared to the direct prompting's useless invariants. However, we can see LLM prompting techniques may fail to predict all potential predicates that can generate a sound invariant to predict all reachable states for some system, i.e., producer-consumer, which involves complex mathematical equations to represent its reachable states.

C. Invariant Quality Evaluation

Based on the ablation study of different prompting techniques, PSyn and ISyn with CoT prompting perform relatively better than other two techniques. We adopt CoT PSyn and ISyn to compare with RunVS and GPT-o3. We can see PSyn under chain-of-thought prompting has comparable performance with RunVS symbolic tool with user-provided predicates. However, its usefulness declines when predicates demand intricate combinations of mathematical expressions. Even users must already have a comprehensive knowledge of the system to formulate such predicates. ISyn-generated invariants are more general than PSyn's, but compared to GPT-o3's, more invariants imply safety properties, which indicate the usefulness of generated likely invariants. Besides, for deeper analysis of ISyn results, we conduct root cause analysis over ISyn-generated invariants in Table IV.

System	Tightness			Soundness			Safety		
	0-shot	1-shot	CoT	0-shot	1-shot	CoT	0-shot	1-shot	CoT
Peterson [16]	10	—	—	✓	—	—	✓	—	—
Bakery [16]	2	2	2	✓	✓	✓	✓	✓	✓
Alternating Bit Protocol [16]	300	300	300	✓	✓	✓	✓	✓	✓
Leader Election [16]	2	2	2	✓	✓	✓	✓	✓	✓
UPPAAL Train/Gate [16]	202	16	14	✓	✓	✓	✗	✓	✓
Distributed Lock Server [20]	6	6	6	✓	✓	✓	✓	✓	✓
Producer Consumer [21]	505000	10200	5050	✗	✗	✗	✗	✓	✓
Smart Contract(Ethereum Transactions) [22]	7200	9700	9500	✓	✓	✓	✓	✓	✓

TABLE I: **Likely Invariant Quality Evaluation** of PSyn with different prompting techniques

System	Tightness			Soundness			Safety		
	0-shot	1-shot	CoT	0-shot	1-shot	CoT	0-shot	1-shot	CoT
Peterson [16]	16	—	—	✓	—	—	✓	—	—
Bakery [16]	200	2	199	✓	✓	✓	✗	✓	✗
Alternating Bit Protocol [16]	400	300	301	✓	✓	✓	✗	✓	✓
Leader Election [16]	3	3	3	✓	✓	✓	✓	✓	✓
UPPAAL Train/Gate [16]	300	400	399	✓	✓	✓	✓	✗	✓
Distributed Lock Server [20]	11	12	7	✓	✓	✓	✗	✗	✓
Producer Consumer [21]	250000	505000	1000000	✗	✗	✓	✗	✗	✗
Smart Contract(Ethereum Transactions) [22]	10000	10000	10000	✓	✓	✓	✓	✓	✓

TABLE II: **Likely Invariant Quality Evaluation** of ISyn with different prompting techniques

System	Tightness				Soundness				Safety			
	PSyn	ISyn	RunVS	GPT o3	PSyn	ISyn	RunVS	GPT o3	PSyn	ISyn	RunVS	GPT o3
Peterson [16]	10	16	30	16	✓	✓	✓	✓	✓	✓	✓	✓
Bakery [16]	2	2	2	2	✓	✓	✓	✓	✓	✓	✓	✓
Producer Consumer [21]	5050	1000000	10100	1000000	✗	✓	✓	✓	✓	✗	✓	✗
Alternating Bit Protocol [16]	300	301	300	400	✓	✓	✓	✓	✓	✓	✓	✗
Leader Election [16]	2	3	2	100	✓	✓	✓	✓	✓	✓	✓	✗
UPPAAL Train/Gate [16]	14	399	300	210	✓	✓	✓	✓	✓	✓	✓	✗
Distributed Lock Server [20]	6	7	6	5	✓	✓	✓	✗	✓	✓	✓	✗
Smart Contract (Ethereum) [22]	9500	10000	10000	10000	✓	✓	✓	✓	✓	✓	✓	✓

TABLE III: **Likely Invariant Quality Evaluation** on GPT-o3, RunVS, PSyn and ISyn

Root Cause	**Explanation**
Over-conservativeness	After many counter-examples the model collapses to an overly general invariant (e.g. `true`), which trivially holds but conveys no useful information.
Partial Description	An invariant is too precise or incomplete.
CE Violation	Generated invariants are not satisfied by counterexamples.
Specification Violation	Generated invariant omits or contradicts constraints explicitly stated in the synthesis instructions (e.g., operator misuse or redundant information).
Hallucination	The LLM introduces symbolic names (variables, channels) that are *absent* from the original Promela model, yielding syntactically valid but semantically meaningless predicates.

TABLE IV: Typical root causes of incorrect LLM-generated invariants.

D. Invariant Generation Efficiency Evaluation

To benchmark the *efficiency* of PSyn and ISyn, we measured the execution time, the ratio of observed states to the total reachable states of the program (determined either manually or through the Spin model checker), the average number of revision events, and the average longevity of final likely invariants. The outcomes for each distributed system are presented in Table V. We can see the likely invariants can be converged within a few revision events and can survive for a large number of traces. Furthermore, we show the average execution time of each LLM API call and each revision event. We can see the invariant can be revised efficiently without incurring much overhead per LLM call.

VII. Validity Threats

The performance of LLMs may be influenced by their training data. If the models were exposed to the Promela invariant corpus during pre-training, their outputs could reflect memorization rather than reasoning. To mitigate this, we included Promela programs crafted by humans to avoid a potential overlap with the LLM training dataset. Besides, we have shown the baseline LLM model's performance at the time of the experiments. The baseline LLM is not good at invariant synthesis with a direct prompt. Besides, our evaluation focuses on classical distributed systems modeled on Spin and has

Distributed Program	LoC	Shared Vars.	No. of Reachable States (R)	$\frac{\lvert\text{Visited}\rvert}{\lvert R\rvert}$	$\lceil f \rceil$		$\lceil r \rceil$		$T_r\,(s)$	$T_{\text{LLM}}\,(s)$	
					PSyn	ISyn	PSyn	ISyn	PSyn	PSyn	ISyn
Peterson [16]	20	3	16	1	400	498	1	2	0.051	26.77	1.22
Bakery [16]	24	2	8	1	497	498	2	2	0.729	3.54	1.32
Producer Consumer [21]	37	4	1.03M	0.00044	400	498	14	2	0.097	1.06	2.23
Alternating Bit Protocol [16]	42	3	∞	≈ 0	400	498	1	2	0.073	1.72	2.49
Leader Election [16]	127	3	26K	0.0024	499	498	3	2	0.053	0.59	2.93
UPPAAL train/gate [16]	78	7	16.8M	0.000024	494	498	2	2	0.069	1.39	3.91
Distributed Lock Server [20]	100	4	12.2K	0.015	497	498	3	2	0.054	30.47	1.14
Smart Contract (Ethereum Transactions) [22]	962	21	∞	≈ 0	499	499	1	1	0.182	23.14	3.46

TABLE V: **Evaluation results of PSyn and ISyn on large-scale distributed systems.** f: number of sample runs (out of 500) that the final likely invariant survives; r: average number of runs that trigger a revision; T_r: average revision time for PSyn; T_{LLM}: average LLM-interaction time per revision.

not experimented in the whole domain, e.g., systems in other languages, sequential programs. Future work should validate our frameworks on real-world systems.

Hyperparameters could affect the overall performance of the LLM. Hyperparameter tuning is not done in this work. Besides, experiments involving LLMs, e.g., GPT-3.5, are inherently non-deterministic due to API-level randomness and opaque model updates. While we report the parameters and prompts used for the experiments, exact replication may require access to fixed model versions. These threats are inherent to LLM-based approaches but do not invalidate our core findings. LLMs can aid in invariant synthesis when used properly, and structured revision frameworks have comparable performance with the symbolic solver and outperform direct prompting for invariant synthesis.

VIII. Related Work

Traditional Invariant Synthesis. Prior work on invariant synthesis has focused on dynamic analysis and symbolic reasoning. Daikon [9] pioneered dynamic invariant detection by inferring likely invariants from execution traces, while DIDUCE [10] tracks certain types of invariants for Java programs. Houdini [23] starts with candidate invariants and removes invariants with the ESC/Java checker. RunVS learns decision tree-based invariants with user-given predicates. These works are more scalable but require more of the given input. Tools like ICE/ICE-DT [5], [6], which use implication counterexamples and decision trees for inductive invariant synthesis. Horn-ICE [24] extended this to handle Horn clauses for sequential and concurrent programs. These methods rely heavily on formal solvers or model checkers, limiting scalability for complex systems. DIG [25] is limited to numerical invariants. DistAI [7] and DuoAI [26] are only limited to the Ivy language and platform. These works have a certain constraints on the invariants or systems.

LLM-Based Invariant Generation. Recent studies explore LLMs for invariant synthesis. Pei et al. [27] demonstrated that LLMs can predict invariants statically, outperforming Daikon when limited to at most five execution traces. Janßen et al. [28] showed that ChatGPT generates valid loop invariants for 75 out of 106 C programs, enabling Frama-C to verify previously unprovable tasks. Chakraborty et al. [29] proposed reranking LLM-generated invariants to reduce verification effort, achieving a median rank of 4 for correct invariants. SymbolicGPT [30] treats invariant learning as a symbolic regression task using a novel transformer-based language model. However, these approaches treat LLMs as passive generators without iterative refinement.

Besides, there are hybrid neuro-symbolic frameworks integrating LLMs with formal methods. Lemur [31] combines LLMs with automated reasoners, using LLMs to propose subgoals validated by SMT solvers. LaM4Inv [32] adopts LLMs with bounded model checking (BMC), where LLMs generate candidates and BMC provides counterexamples for revision. While Lemur leverages LLMs for invariant revision-specific subtasks, it does not fully exploit LLMs' iterative revision capabilities during synthesis. Unlike LaM4Inv, which relies on BMC for corrections and does not fully exploit the information output from LLM, our method enables LLMs to self-correct and extracts atomic predicate templates from LLMs. Besides, LaM4Inv filters out all predicates disproved by BMC without reconstruction, which reduces the diversity and the tightness of invariants. Our work introduces two novel invariant revision frameworks. The synthesizer improves its atomic predicate suggestions based on counterexamples, enabling the LLM to reconstruct its predicates iteratively.

IX. Conclusion

This work establishes that structured revision frameworks significantly enhance the scalability of invariant synthesis with LLMs and also the capability of LLMs in invariant synthesis. Our experimental evaluation demonstrates that iterative refinement on atomic predicates with counterexample guidance improves invariant quality across three critical metrics: soundness, tightness, and safety. We have shown the effectiveness of chain-of-thought prompting over few-shot learning. LLM in PSyn independently revises predicates using structured feedback, akin to human debugging. By integrating LLMs into the invariant synthesis loop, rather than treating them as standalone generators, our frameworks represent a collaborative neuro-symbolic invariant synthesis. Our contribution is to integrate LLM-proposed predicates into a runtime verification loop in a structured way. To our knowledge, this is the first work to combine LLM reasoning and invariant synthesis in

distributed systems. Specifically, two frameworks are new examples of combining LLM reasoning and lightweight formal techniques, such as dynamic invariant generation and runtime monitoring on distributed systems, which increases the scalability. Future work will investigate four directions, i.e., application to other platforms (e.g., P language), exploring code- or math-specialized LLMs and fine-tuning, extension to online monitoring framework, and extension to self-debugging or self-healing systems.

Acknowledgments. This work was partially supported by the National Science Foundation through the following grants: CAREER award 2048094, FMitF-2124431, and an Amazon Faculty Research Award.

REFERENCES

[1] A. Chlipala, *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013.

[2] M. Wenzel, "Isabelle as document-oriented proof assistant," in *International Conference on Intelligent Computer Mathematics*. Springer, 2011, pp. 244–259.

[3] I. Dillig, T. Dillig, B. Li, and K. McMillan, "Inductive invariant generation via abductive inference," *Acm Sigplan Notices*, vol. 48, no. 10, pp. 443–456, 2013.

[4] L. Kuczynski and K. Daly, "Qualitative methods for inductive (theory-generating)," *Handbook of dynamics in parent–child relations*, pp. 373–392, 2003.

[5] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Ice: A robust framework for learning invariants," in *In Proc. of 26th Int. Conf. on Computer Aided Verification*, 2014, pp. 69–87.

[6] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *ACM Sigplan Notices*, vol. 51, no. 1. ACM New York, NY, USA, 2016, pp. 499–512.

[7] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan, "DistAI: Data-Driven automated invariant learning for distributed protocols," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 2021, pp. 405–421.

[8] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: safety verification by interactive generalization," in *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 614–630.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," in *Science of computer programming*, vol. 69, no. 1-3. Elsevier, 2007, pp. 35–45.

[10] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th international conference on Software engineering*, 2002, pp. 291–301.

[11] Y. Xia, D. Sur, A. S. Pingle, J. Deshmukh, M. Raghothaman, and S. Ravi, "Llm-guided predicate discovery and data augmentation for learning likely program invariants," in *To appear in The 40th ACM/SIGAPP Symposium On Applied Computing*, 2025.

[12] G. J. Holzmann and W. S. Lieberman, *Design and validation of computer protocols*. Prentice hall Englewood Cliffs, 1991, vol. 512.

[13] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The journal of logic and algebraic programming*, vol. 78, no. 5, pp. 293–303, 2009.

[14] L. Gao, M. Lu, L. Li, and C. Pan, "A survey of software runtime monitoring," in *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2017, pp. 308–313.

[15] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[16] G. J. Holzmann, "The model checker spin," in *IEEE Transactions on software engineering*, vol. 23, no. 5. IEEE, 1997, pp. 279–295.

[17] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[18] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

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

[20] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, "I4: incremental inference of inductive invariants for verification of distributed protocols," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 370–384.

[21] G. Byrd and M. Flynn, "Producer-consumer communication in distributed shared memory multiprocessors," *Proceedings of the IEEE*, vol. 87, no. 3, pp. 456–466, 1999.

[22] Z. Yang, M. Dai, and J. Guo, "Formal modeling and verification of smart contracts with spin," *Electronics*, p. 3091, 2022.

[23] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *International Symposium of Formal Methods Europe*, 2001, pp. 500–517.

[24] P. Ezudheen, D. Neider, D. D'Souza, P. Garg, and P. Madhusudan, "Horn-ice learning for synthesizing invariants and contracts," in *Proc. of the ACM on Programming Languages*, vol. 2, 2018, pp. 1–25.

[25] T. Nguyen, K. Nguyen, and M. B. Dwyer, "Using symbolic states to infer numerical invariants," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3877–3899, 2021.

[26] J. Yao, R. Tao, R. Gu, and J. Nieh, "Duoai: Fast, automated inference of inductive invariants for verifying distributed protocols," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 485–501.

[27] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?" in *International Conference on Machine Learning*. PMLR, 2023, pp. 27 496–27 520.

[28] C. Janßen, C. Richter, and H. Wehrheim, "Can chatgpt support software verification?" in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2024, pp. 266–279.

[29] S. Chakraborty, S. K. Lahiri, S. Fakhoury, M. Musuvathi, A. Lal, A. Rastogi, A. Senthilnathan, R. Sharma, and N. Swamy, "Ranking llm-generated loop invariants for program verification," *arXiv preprint arXiv:2310.09342*, 2023.

[30] M. Valipour, B. You, M. Panju, and A. Ghodsi, "Symbolicgpt: A generative transformer model for symbolic regression," *arXiv preprint arXiv:2106.14131*, 2021.

[31] H. Wu, C. Barrett, and N. Narodytska, "Lemur: Integrating large language models in automated program verification," *arXiv preprint arXiv:2310.04870*, 2023.

[32] G. Wu, W. Cao, Y. Yao, H. Wei, T. Chen, and X. Ma, "Llm meets bounded model checking: Neuro-symbolic loop invariant inference," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 406–417.