PAT-Agent: Autoformalization for Model Checking

Xinyue Zuo
National University of Singapore
Singapore
zuoxy@nus.edu.sg

Yifan Zhang
National University of Singapore
Singapore
yifan.zhang_@u.nus.edu

Hongshu Wang
National University of Singapore
Singapore
hongshu.wang@u.nus.edu

Yufan Cai*
National University of Singapore
Singapore
caiyf@nus.edu.sg

Zhe Hou
Griffith University
Australia
z.hou@griffith.edu.au

Jing Sun
University of Auckland
New Zealand
jing.sun@auckland.ac.nz

Jin Song Dong
National University of Singapore
Singapore
dcsdjs@nus.edu.sg

Abstract—Recent advances in large language models (LLMs) offer promising potential for automating formal methods. However, applying them to formal verification remains challenging due to the complexity of specification languages, the risk of hallucinated output, and the semantic gap between natural language and formal logic. We introduce PAT-Agent, an end-to-end framework for natural language autoformalization and formal model repair that combines the generative capabilities of LLMs with the rigor of formal verification to automate the construction of verifiable formal models. In PAT-Agent, a *Planning LLM* first extracts key modeling elements and generates a detailed plan using semantic prompts, which then guides a *Code Generation LLM* to synthesize syntactically correct and semantically faithful formal models. The resulting code is verified using *the Process Analysis Toolkit (PAT) model checker* against user-specified properties, and when discrepancies occur, a *Repair Loop* is triggered to iteratively correct the model using counterexamples. To improve flexibility, we built a web-based interface that enables users, particularly non-FM-experts, to describe, customize, and verify system behaviors through user-LLM interactions. Experimental results on 40 systems show that PAT-Agent consistently outperforms baselines, achieving high verification success with superior efficiency. The ablation studies confirm the importance of both planning and repair components, and the user study demonstrates that our interface is accessible and supports effective formal modeling, even for users with limited formal methods experience.

Index Terms—Model Checking, Large Language Models, Autoformalization, LLM Agent

I. INTRODUCTION

Formal methods provide mathematically rigorous techniques for specifying and verifying the correctness of software and hardware systems. Despite their well-established benefits in safety-critical and concurrent domains, their broader adoption remains limited due to the steep learning curve, lack of accessible tooling, and the complexity of formal languages.

Recent advances in large language models (LLMs), such as OpenAI o3 [26] and LLaMA [13], have demonstrated strong capabilities in natural language understanding and code generation. These models present a promising opportunity to bridge the gap between informal requirements and formal specifications, particularly for users without formal methods expertise. By leveraging LLMs, it becomes feasible to lower the barrier to entry for formal modeling and make correctness assurance more broadly accessible.

Challenges. However, applying LLMs to formal model synthesis introduces significant challenges. LLMs are prone to hallucinations and logical inconsistencies [17], [39], especially when generating complex artifacts such as process algebra models or temporal logic specifications. Their outputs may violate syntactic rules, diverge from user intent, or overlook critical correctness properties. Existing works on synthesizing formal specifications from natural language instructions have a focus on fine-grained formulae, such as temporal logic [8], [11]. There are emerging attempts to synthesize system-level models using LLMs, but these works [5], [16] focus on simpler systems and exhibit limited success, relying solely on direct generation from natural language. In contrast, PAT-Agent integrates structured planning and verification-guided repair, enabling the synthesis of more complex and verifiable models. Meanwhile, traditional rule-based synthesis pipelines are labor-intensive and difficult to generalize across diverse systems and domains. These limitations underscore the need for a unified framework that combines the flexibility and automation capacity of LLMs with the rigor of formal verification to produce reliable, verifiable models at scale.

Our Solution. We propose **PAT-Agent**, a novel framework for natural language autoformalization and formal code repair. PAT-Agent combines the generative power of LLMs with the rigor of formal verification to automate the construction of verifiable system models. Built on top of the Process Analysis Toolkit (PAT), our approach differs from direct natural language–to–code generation and instead adopts a structured intermediate representation based on semantic prompts. These prompts capture reusable modeling constructs, such as variables and guarded actions, and are parameterized to guide code generation in a controlled, interpretable manner.

The pipeline operates in modular stages. A Planning LLM first analyzes the natural language description and extracts

the modeling elements. These elements are further used to populate the structured semantic prompt that guides the generation of a detailed plan for formal code synthesis. A Code Generation LLM then converts the plan into syntactically valid, semantically aligned PAT models. The resulting model is verified using the PAT model checker, and if any user-defined requirements are not satisfied, a Repair Loop is triggered to iteratively revise the model based on counterexamples and failure traces. This decomposition of tasks across multiple LLMs is motivated by prior works [18], [9], which show that breaking down complex generation into several sub-tasks can improve effectiveness.

To support usability, accessibility, and iterative development, we further develop a web-based interface that supports guided modeling, verification, and repair. Users can input natural language descriptions, inspect and edit the extracted intermediate representations, review verification results, and apply automatic corrections through a seamless feedback loop. All code, datasets, and the interface are publically available[1].

This paper makes the following contributions:

- We propose PAT-Agent, a *fully automated end-to-end* framework for natural language autoformalization and formal code repair. It uses a semantic prompt-based decomposition strategy to extract structured modeling elements and incorporates a verification-guided repair loop to iteratively correct invalid models based on model checker feedback. An ablation study confirms the effectiveness of both components in achieving high verification success.
- We conduct a comprehensive evaluation of PAT-Agent on 40 diverse modeling tasks curated from three sources, demonstrating its high effectiveness and efficiency in generating correct and verifiable models compared to direct LLM baselines and alternative model combinations.
- We design a user-centric interface that enables human-in-the-loop development, complementing the automated pipeline. It allows non-experts to build and verify formal system models with minimal effort. A user study confirms the usability and accessibility of the interface.

II. PRELIMINARY

This section gives an overview of the Process Analysis Toolkit (PAT) and its modelling language *CSP#*.

A. The PAT Framework

Model checking is a method for checking whether a finite-state model of a system meets a given specification. We build our autoformalization framework on PAT (Process Analysis Toolkit [33]), an existing extensible model checker that supports state-space exploration and LTL verification for concurrent and real-time systems. Its underlying formalism, *CSP#*, is a dialect of Communicating Sequential Processes (CSP) enriched with imperative C# statements and typed variables. PAT has been successfully applied in domains such as industrial protocols, real-time controllers, and cyber-physical systems [32], [2].

Nevertheless, building a correct model remains an expertise-intensive task. Our method tackles this bottleneck by prompting large language models (LLMs) to synthesize CSP# code directly from informal descriptions, and then verifying it against requirements.

B. Syntax and Semantics of CSP#

The subset of CSP# used in our study can be generated by the grammar:

$$P ::= \textbf{Stop} \mid \textbf{Skip} \mid e \to P \mid e\{prog\} \to P \mid P \square Q$$
$$\mid \textbf{if}(b)\{P\} \textbf{ else}\{Q\} \mid P; Q \mid P \parallel_A Q \mid P \mid\mid\mid Q$$

A *process* is a labelled transition system (LTS)

$$\langle S, Act, \to, s_0 \rangle, \text{ where}$$

- S is the set of program states—valuations of a finite set of typed variables;
- Act is the alphabet of *observable events*; internal (silent) actions are denoted τ and excluded from Act;
- $\to \subseteq S \times (Act \cup \{\tau\}) \times S$ is the transition relation;
- $s_0 \in S$ is the initial state.

A step $s \xrightarrow{\alpha} s'$, with α being an event, communication, or τ, represents an *atomic* state change.

Event primitives: Besides basic events e, CSP# supports imperative C# statements written as `{prog}`, which execute atomically and may update multiple variables between two adjacent states.

Compositional operators: Complex systems are built from smaller processes via the operators below:

- *Sequential composition* $P; Q$ runs P and then runs Q.
- *External choice* $P \square Q$ offers the union of initial events; the environment selects a branch.
- *Parallel composition* $P \parallel_A Q$ forces synchronization on the alphabet A and interleaves all other events.
- *Interleaving* $P \mid\mid\mid Q$ executes both operands independently without event synchronization.

C. Assertions and Verification in PAT

A directive `#assert` embeds a decision problem about the generated LTS. We rely on three built-in classes:

Deadlock-freedom: `#assert P deadlockfree` checks whether every reachable state enables a transition.

Reachability: `#assert P reaches cond` checks whether a state satisfying predicate `cond` is reachable.

Linear temporal logic (LTL): `#assert P ⊨ F` checks whether all execution traces of P satisfy the LTL formula F.

Detailed explanations of the syntax and semantics of CSP# and PAT's assertion language are available in the literature [30].

In our pipeline, users specify two inputs: properties in natural language and the expected verification outcomes (VALID or INVALID). These inputs together form the *requirements*. The properties are then autoformalized into corresponding PAT assertions. PAT evaluates the assertions, and the verification results are compared against the expected outcomes. A requirement is satisfied if the two match.

III. Approach

We propose **PAT-Agent**, a novel framework that tightly integrates LLMs with PAT to automate the synthesis and formal verification of system models from natural language specifications. As illustrated in Figure 1, the PAT-Agent framework comprises four core components:

- **Planning LLM** analyzes the natural language specification to extract key semantic elements, such as constants, variables, actions, and guarded conditions, and maps them into the parameterized slots of domain-specific semantic prompts, producing a detailed code generation plan that guides formal model construction.
- **Code Generation LLM** converts the detailed plan into formal models written in the PAT specification language, ensuring syntactic correctness and semantic fidelity to the original user intent.
- **Model Checker (PAT)** verifies whether the generated model satisfies the user-specified requirements, including safety, liveness, deadlock-freedom, and produces diagnostic feedback when a requirement is not satisfied.
- **Repair Loop** leverages feedback from PAT, such as counterexamples and violation traces, to iteratively revise the model. While LLMs assist in interpreting the feedback and producing corrections, the pipeline provides targeted guidance to ensure effective and controlled repair.
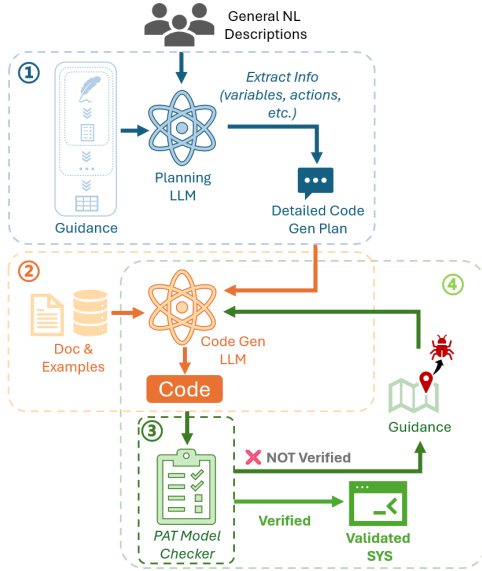

Fig. 1: PAT-Agent Framework Overview.

We formalize our pipeline as follows:

- $\mathcal{L}_{\mathrm{NL}}$ denotes the set of natural language specifications;
- Π is the space of *planning representations*—structured semantic prompt templates that are partially instantiated, with placeholders for constants, variables, and actions;
- $\mathcal{P}$ denotes the detailed model generation plan;
- $\mathcal{M}$ is the set of well-formed PAT models;

- $\mathcal{Q} = \{(\varphi_1, o_1), \ldots, (\varphi_m, o_m)\}$ denotes a finite set of user requirements, where each property φ_i is paired with an expected outcome $o_i \in \{\text{VALID}, \text{INVALID}\}$;
- $\mathcal{V}$ denotes the verification procedure of PAT;
- $\mathcal{C}$ denotes the set of all possible counterexample traces (diagnostic execution sequences) that PAT may return when a requirement is not satisfied;
- $\mathcal{R}$ denotes the repair instructions combined with PAT verification results.

Given a natural-language specification $N \in \mathcal{L}_{\mathrm{NL}}$ and a finite set of requirements $\mathcal{Q}$, our goal is to construct a *well-formed* PAT model $M^\star \in \mathcal{M}$ from N such that

$$\forall (\varphi, o) \in \mathcal{Q} : \mathcal{V}(M^\star, \varphi) = o.$$

Core Transformations. We formalize each pipeline stage as a possibly stochastic transformer:

$$\begin{aligned}
\mathcal{T}_{\mathrm{plan}} &: \mathcal{L}_{\mathrm{NL}} \longrightarrow \Pi \longrightarrow \mathcal{P}, \\
\mathcal{T}_{\mathrm{gen}} &: \mathcal{P} \longrightarrow \mathcal{M}, \\
\mathcal{V} &: \mathcal{M} \times \mathcal{Q} \longrightarrow \{\text{MATCH}, \text{MISMATCH}\}^m \times \mathcal{C}, \\
\mathcal{T}_{\mathrm{repair}} &: \mathcal{M} \times \mathcal{C} \times \mathcal{R} \longrightarrow \mathcal{M}.
\end{aligned}$$

A. Planning LLM

To enhance both reliability and interpretability, PAT-Agent creates a semantic prompt-based synthesis strategy in place of end-to-end free-form code generation. Rather than directly mapping natural language to formal specifications, the framework introduces an intermediate planning step built around reusable, parameterized modeling prompts. Each modeling prompt has four components:

- A concise role specification that instructs the LLM to perform a targeted analysis task (e.g., extracting constants and variables from the system description).
- Supporting context, including the relevant user-provided natural language description or modeling constructs extracted in earlier stages to inform the current analysis.
- Structural guidance for the specific modeling element being extracted. These include:
  - Constant: name, value, and description
  - Variable: name, type, possible values, initial value, and description
  - Process: name, actions, guard conditions, and associated state changes
- Targeted instructions and validation rules for each construct. For example, when processing constants and variables:
  - Constants should include three properties: name, integer value, and description. Variables should include five: name, type (e.g., integer, array), possible values (which must be defined as constants if needed), initial value, and description.
  - All constants must be declared before variables. Names across constants, variables (including their possible values), and processes must be unique and non-conflicting.

The complete set of instructions is provided online [1].

Formally, $\mathcal{T}_{\mathrm{plan}}$ leverages *semantic prompts* $\mathrm{Prompt}_{\mathrm{var}}$, $\mathrm{Prompt}_{\mathrm{act}}, \ldots$ to extract (I) typed variables and constants,

and (II) guarded actions. The output is JSON-serializable. These extracted elements are used as inputs to subsequent semantic prompts (Π) to generate a detailed natural language annotation — referred to as the model generation plan $\mathcal{P}$. The prompt to generate the coding plan has three components: (1) role assignment, (2) the previously extracted structured elements, and (3) an instruction to formulate detailed, plan-level annotations that guide formal code generation.

B. Code-Generation LLM

Given the detailed modeling plan produced by the Planning LLM that specifies constants, variables, and state transitions, the Code-Generation LLM synthesizes a complete and executable model in the PAT formal specification language. Users also provide properties and expected outcomes that describe the intended system requirements. As system descriptions and requirements are specified as separate inputs in our pipeline, PAT-Agent first constructs the model from the description and then verifies it against the requirements to ensure consistency with the intended system behavior.

The modeling plan takes the form of a line-by-line natural language annotation that maps directly to formal constructs. To enhance robustness and code quality, we incorporate two forms of prompt augmentation. First, we embed syntactic guidance in the form of a compact documentation excerpt, summarizing PAT syntax and common errors observed during early experimentation (e.g., missing semicolons, incorrect process synchronization, malformed assertions). These cues help the model avoid structural mistakes and adhere to PAT's specification conventions. Second, we provide semantic guidance via a retrieval-augmented generation (RAG) mechanism. For each modeling task, we compare the newly generated plan against a small curated database of <plan, code> pairs and retrieve the most similar example based on plan content. The retrieved example's plan-code correspondence is then embedded into the prompt, illustrating how a comparable plan was translated into formal PAT code. If no closely similar plan exists, the retrieved example will serve as a one-shot illustration to guide code structure, syntax, and generation behavior.

Formally, $\mathcal{T}_{\text{gen}}$ converts each annotation in $\mathcal{P}$ into PAT syntax. Two auxiliary signals improve robustness: (a) a *syntax cue* σ_{PAT} summarising common mistakes, (b) a retrieval-augmented exemplar $(\mathcal{P}', M')$ with highest cosine similarity to $\mathcal{P}$.

C. Model Checker (PAT)

Once a formal model is generated, it is automatically verified against user-specified requirements using the PAT model checker. For each φ_i we invoke PAT's built-in engines: deadlock detection, reachability, or LTL. PAT returns a raw verification outcome for each φ_i, which is compared against the user-provided expected outcome o_i to produce a verdict $s_i \in \{\text{MATCH}, \text{MISMATCH}\}$. Collecting over all m properties yields the verdict vector $\vec{s} \in \{\text{MATCH}, \text{MISMATCH}\}^m$. When $s_i = \text{MISMATCH}$, we also retrieve from PAT a minimal counter-example trace $C = (t_0 \to t_1 \to \cdots \to t_\ell) \in \mathcal{C}$, where $t_i, 0 \le i \le \ell$, are events in the model, to guide repair.

D. Verification-Repair Loop

Starting from an initial model $\mathcal{M}_0 = \mathcal{T}_{\text{gen}}(\mathcal{P})$, we perform iterative repair based on verification results:

$$(\vec{s}_k, C_k) \leftarrow \mathcal{V}(M_k, \mathcal{Q}),$$
$$\mathcal{M}_{k+1} \leftarrow \begin{cases} \mathcal{M}_k, & \text{if } \vec{s}_k \equiv \text{MATCH}^m \\ \mathcal{T}_{\text{repair}}(\mathcal{M}_k, C_k, R_k), & \text{otherwise.} \end{cases}$$

The loop halts at the first iteration $k^\star$ where the verdict vector $\vec{s}_{k^\star}$ indicates that all verification results match the expected outcomes specified by the users (i.e., $\vec{s}_{k^\star} \equiv \text{MATCH}^m$), thereby yielding the verified model $M^\star = M_{k^\star}$. We empirically cap k at $K_{\max} = 5$; if no satisfactory model is found within this limit, the user is given the latest model and counterexample trace. This choice balances repair effectiveness with computational cost, as increasing $K_{\max}$ further yields diminishing returns in our experiments.

Each iteration leverages structured feedback from PAT in the form of a counterexample trace C_k. It is interpreted to generate repair directives $R_k \in \mathcal{R}$, which (I) rank the actions appearing in C_k using a locality-of-fault heuristic, prioritizing nodes closer to the violation point, and (II) generate targeted edit commands such as *tighten guard of* START_DRIVING.

Directives are fed back to $\mathcal{T}_{\text{gen}}$ *in-context*, guiding code-level edits without rerunning the full planning pipeline and ensuring locality of change.

E. Interactive Interface

While PAT-Agent is fully capable of end-to-end automation, we also provide a user-friendly web-based interface to support controllable and interactive formalization when desired. This interface is designed to make formal modeling accessible to non-experts by supporting the entire pipeline through guided, modular interactions. It allows users to review and customize system components without requiring prior knowledge of formal specification languages or model checking. Key components of the interface include:

Chatbot: Users begin with a high-level natural language description of the system they wish to build, which the chatbot uses to detect reuse opportunities. It checks the database of verified systems—stored with both natural language descriptions and formal implementations—and either redirects users to *Verified Example Customization* when a similar system exists, or to *Information Gathering & Processing* for new synthesis when none match. If the high-level description is ambiguous (i.e., insufficient to determine whether a similar system exists), the chatbot requests clarification. Detailed descriptions and requirements are only collected systematically later in the *Information Gathering & Processing* stage.

Verified Example Customization: The interface offers access to historically synthesized and verified systems. When similarities exist, users can adjust key parameters or reuse components, enabling rapid adaptation, like code reuse in programming. If no similarities exist, users proceed with fresh synthesis. To support long-term scalability, PAT-Agent maintains this self-evolving

database of verified formal models, indexed by semantic descriptors such as system goals and requirements. Newly synthesized models are added upon successful verification, enriching a growing repository of reusable implementations.

Information Gathering & Processing: If a new model is needed, the interface first collects natural language system descriptions from users. It then displays the Planning LLM's interpreted results in structured tables, each corresponding to a distinct modeling construct. These include constants and variables (with types, value ranges, and initial values), and actions (with associated conditions and state transitions). Users can review and revise the extracted content.

Requirements Specification: Users specify requirements (including intended outcomes) via a guided interface with dropdowns and selection tools for defining states using familiar concepts like variable-value pairs. Linear temporal logic formulae can also be created without requiring users to manually write formal syntax, as system translation internally ensures syntactic correctness. Users can revise their specifications at any time if they notice a misalignment with their intentions.

Plan, Code, and Verification Viewers: The Plan and Code viewers allow users to inspect the natural language modeling plan and property descriptions, and the synthesized PAT model and formalized assertions, correspondingly. This enables users to confirm correctness or make edits. The Verification viewer presents each requirement with its verification outcome, and, if not satisfied, counterexamples that highlight discrepancies between user expectations and actual system behavior.

Repair Dashboard: For models that fail to satisfy all specified requirements, the interface displays the current implementation and unsatisfied requirements. Users can edit code or trigger automatic repair, and automated updates include human-readable summaries. If verification passes, the implementation is guaranteed to meet all user-confirmed requirements.

A full demonstration of the interface, along with an illustrative video, is available in our GitHub repository [1]. To evaluate the effectiveness and usability of the interface, we conducted a user study involving participants from both formal-methods and non-expert backgrounds, as detailed in Section V.

F. Running Example

We illustrate each step with a four-process car model (driver, key, door, motor). The four processes model four interacting elements — driver position, key position, door status, and engine behavior — that collectively define the car's operational logic. A valid system design must satisfy key safety and functionality requirements, such as preventing the car from entering a driving state without a licensed driver, avoiding key lock-in scenarios, and ensuring overall deadlock-freedom.

Description of Engine Behavior

This process models the motor in the car. The engine can be turned on and off, the car can start and stop driving, the fuel is gradually consumed after driving, and it can be refueled.

Planning with Semantic Prompts. Users only need to provide high-level natural language descriptions of the system, such as the engine behavior description shown above. These descriptions are automatically formatted into a structured supporting context (as illustrated in Figure 2a), which is then used to form the initial semantic prompt for extracting constants and variables.

With the full semantic prompt containing the supporting context, the Planning LLM identifies and extracts relevant constants and variables (e.g., `carMotion`, `fuel`, and `engineStatus`), as demonstrated in Figure 2b.

```
1  {
2    "modelName": "car",
3    "modelDesc": ...,
4    "interaction": ...,
5    "subsystemCount": 4,
6    "subsystems": ...
7  }
```

```
1  {
2    "processes": [
3      {
4        "name": ...,
5        "constants": ...,
6        "variables": ...
7      },
8      ...
9    ]
10 }
```

(a) Car System Context 1   (b) Car System Context 2

Fig. 2: Two Context JSONs of the car system in the running example. These contexts serve as the "Guidance" input in Figure 1.

The extracted elements, along with the relevant system descriptions, are subsequently used as the supporting context in the semantic prompt to reason about possible state transitions, which we refer to as actions. This involves not only identifying explicitly stated behaviors but also applying commonsense reasoning to infer plausible conditions and effects. For instance, even if not stated verbatim, the LLM may infer that the action *start_driving* should depend on conditions such as the engine being on, sufficient fuel being available, the car being stationary, and the owner being present. These inferred behaviors are essential for generating context-aware and semantically valid formal representations.

The structured templates within the semantic prompts, such as Figure 2a and Figure 2b, correspond to the input "Guidance" in Figure 1, which supports the Planning LLM to analyze systems. With all modeling constructs extracted, a detailed code generation plan will be formed through the prompt introduced in Section III-A.

Plan-Guided Formal Model Generation. For example, an annotation in the plan and corresponding PAT code for an action such as `start_driving` are as follows:

```
1  //If "engineStatus" is "engineOn", "carMotion" is "stop",
       "fuel" is above 0, and "ownerPos" is "inCar", the
       action "start_driving" make "carMotion" become "drive
       " (drive started)
2  [engineStatus == engineOn && carMotion == stop && (fuel >
       0) && ownerPos == inCar] start_driving{carMotion =
       drive;} -> motor()
```

The "Doc & Examples" in Figure 1 consists of syntax documentation and curated <plan, code> pairs, which guide the Code Generation LLM to produce syntactically correct and

TABLE I: Example Assertions for the Simplified Car System. Each assertion is autoformalized in PAT syntax and paired with the user-specified expected outcome and the verification outcome returned by PAT. Green indicates a match between the two. Red indicates a mismatch, meaning the requirement is not satisfied.

ID	Type	Formalized Assertion	User-Specified Outcome	Verification Outcome
P_1	Deadlock	`#assert car deadlockfree;`	VALID	VALID

P_2	Reachability	`#define keyInsideDoorLocked (key == inCar && door == locked`
		`                          && ownerPos != inCar);`
		`#assert car reaches keyInsideDoorLocked;`
			INVALID	INVALID

P_3	Reachability	`#define driveWithoutDriver (carMotion==drive && ownerPos != inCar);` `#assert car reaches driveWithoutDriver;`	INVALID	VALID

semantically consistent models. The full resources are available in our GitHub repository[2].

Verification Feedback and Iterative Repair. In the car system, key safety and functionality requirements, such as ensuring deadlock-freedom, preventing key lock-in scenarios, and disallowing driving without a licensed driver, are formalized as assertions and used to test model implementations against user-specified outcomes. Table I shows these assertions alongside both the expected outcomes and the verification results. The implementation meets P_1 and P_2, but fails to meet P_3.

This requirement is not satisfied due to potential ambiguities in natural language descriptions or minor generation inaccuracies (e.g., missing preconditions or incorrect behavior specifications), which can cause the initial model to fail some verification criteria. Not satisfying P_3 allows the car to enter a driving state without a person detected in the cockpit, a clear safety and regulatory violation. When such failures occur, the PAT model checker returns detailed diagnostic feedback in the form of violation traces, which act as counterexamples. These traces identify specific execution paths or conditions that trigger the violation, helping to isolate the root cause. For example, a trace revealing why driveWithoutDriver was reachable (`VALID`) for the current system implementation, rather than the desired `INVALID`, is as below:

```
1  <init -> owner_approach -> pickup_key_by_owner ->
2  unlock_door_by_owner -> open_door -> owner0_enter ->
3  start_engine_by_owner0 -> owner0_exit -> start_driving>
```

To revise the model to satisfy the safety requirement, PAT-Agent transforms the diagnostic feedback from PAT into a structured repair strategy and executes it through an iterative repair loop. Instead of re-running the full planning stage, it constructs an augmented prompt that combines information extracted from the violation trace (e.g., the actions or transitions leading to failure) with repair heuristics tailored to the property type of the unsatisfied requirement (e.g., tightening guard conditions for safety or loosening them for liveness), thereby giving targeted revision guidance to the Code Generation LLM.

When a trace implicates multiple actions, the prompt prioritizes those more likely responsible using two heuristics:

actions appearing later in the violation trace (i.e., closer to the failure point) and actions occurring more frequently. For example, in the above trace, `start_driving` appears last and is therefore considered more likely responsible, while other implicated actions are still considered with lower likelihood.

If `start_driving` is wrongly triggered when no driver is present, the prompt suggests tightening its preconditions (e.g., adding a check for driver position). If deadlock issues occur, it may recommend ensuring every state has at least one outgoing transition or relaxing overly strict conditions on related actions. This guided, localized repair preserves correct model components while accelerating convergence toward a verifiable implementation.

IV. EVALUATION

We conduct a comprehensive evaluation of **PAT-Agent** to assess its effectiveness and efficiency in synthesizing verified models from natural language descriptions. Our experiments are designed to answer the following research questions:

- **RQ1:** How effective is the proposed PAT-Agent pipeline in generating formally verified systems from natural language descriptions, compared to alternative generation strategies?
- **RQ2:** What is the impact of the key components comprising the Planning LLM and the Repair Loop on the overall performance of the PAT-Agent pipeline?
- **RQ3:** How does the processing time vary across different stages of the PAT-Agent pipeline and different model configurations?

A. Datasets and Baselines

We evaluate PAT-Agent on three datasets: **PAT** (26 pairs of natural language descriptions and requirements collected from the PAT library), **A4F** (8 pairs adapted from the Alloy4Fun dataset [22]), and **UCS** (6 pairs curated from the CSP textbook Understanding Concurrent Systems [28]), totaling 40 modeling examples. We curated the examples that can be formalized in the syntax defined in section II. Table II summarizes the datasets used in our evaluation. It reports the number of system instances and total number of verification assertions, as well as two statistics computed over the final verified models: the average of each system's maximum number of visited states and

TABLE II: Dataset Statistics.

Dataset	#Instances	#Assertions	States	Transitions
PAT	26	74	101253.62	342084.42
UCS	6	19	1164.17	14472.83
A4F	8	40	1223.00	7008.50
Overall	40	133	66234.07	225927.50

the average of each system's maximum number of transitions observed during verification.

In our study, we adopt OpenAI's *o3-mini-2025-01-31*[3] as the Planning LLM (referred to as o3) and Anthropic's *claude-3-7-sonnet-20250219*[4] as the Code Generation LLM (referred to as Claude). We denote this configuration as <o3, Claude>, which serves as the default setup of our PAT-Agent pipeline. To benchmark the effectiveness and efficiency of alternative model pairings, we also experiment with *DeepSeek-R1*[5] (referred to as R1).

B. Metrics

To evaluate the effectiveness of our framework, we employ three metrics commonly used in software engineering research [35], [7], [14]: Compilation Success Rate (CSR), Full-Pass Rate (FPR), and Average Pass Rate (APR). In our setting, "pass" means that the verification result produced by PAT matches the expected outcome specified by the user for an assertion, and "full-pass" means that a system satisfies all of its specified requirements. The metrics are defined as follows:

$$\text{CSR} = \frac{|S_{\text{succ}}|}{N}, \quad \text{FPR} = \frac{|S_{\text{full}}|}{N}, \quad \text{APR} = \frac{\sum_{i=1}^{N} p_i}{\sum_{i=1}^{N} A_i}$$

Let N denote the total number of systems for which formal models are generated. CSR measures the proportion of these systems that compile without syntax errors, where $S_{\text{succ}} \subseteq N$ denotes those successfully compiled.

FPR captures the fraction of systems that pass all their assertions, with $S_{\text{full}} \subseteq N$ denoting such systems.

APR offers a fine-grained view of assertion-level correctness. Let p_i be the number of assertions passed by system i, and A_i the total assertions it must satisfy; APR then reflects the overall proportion of satisfied assertions across all systems — even when some systems fail a subset of checks.

C. Results of RQ1: Effectiveness of Formal Model Generation

Settings. To evaluate the effectiveness of our pipeline in generating formally verified models from natural language descriptions, we compare its performance against direct LLM-based generation methods, as well as against five alternative model combinations using the same pipeline structure. The baseline methods include individual LLMs (o3, Claude, R1) used to generate code directly without a structured planning step or repair loop. These direct generations are provided with

[3]https://platform.openai.com/docs/models/o3-mini
[4]https://docs.anthropic.com/en/docs/about-claude/models/overview
[5]https://api-docs.deepseek.com/news/news250120

TABLE III: Performance against Direct Generation with LLMs.

Dataset	Model	CSR	FPR	APR
PAT	R1	0.3846	0.3462	0.3243
	o3	0.7308	0.5769	0.7162
	Claude	0.7308	0.5385	0.6892
	Ours	**1.0000**	**1.0000**	**1.0000**
UCS	R1	0.3333	0.3333	0.2632
	o3	0.6667	0.5000	0.5789
	Claude	0.6667	0.1667	0.4211
	Ours	**1.0000**	**1.0000**	**1.0000**
A4F	R1	0.6250	0.5000	0.5750
	o3	0.7500	0.3750	0.5250
	Claude	0.8750	0.5000	0.6250
	Ours	**1.0000**	**1.0000**	**1.0000**
Overall	R1	0.4250	0.3750	0.3910
	o3	0.7250	0.5250	0.6391
	Claude	0.7500	0.4750	0.6316
	Ours	**1.0000**	**1.0000**	**1.0000**

TABLE IV: Verification Results of Formal Models Generated by Different Pipeline Configurations. Each entry in the Pipeline column is denoted as <model$_{\text{plan}}$, model$_{\text{gen}}$>, where model$_{\text{plan}}$ is used for planning and model$_{\text{gen}}$ for code generation. Best results are bolded; second-best results are underscored.

Dataset	Pipeline	CSR	FPR	APR
PAT	**<o3, Claude>**	**1.0000**	**1.0000**	**1.0000**
	<R1, Claude>	<u>0.8846</u>	<u>0.8462</u>	<u>0.8514</u>
	<R1, R1>	0.6923	0.6538	0.6081
	<R1, o3>	0.6923	0.6923	0.6351
	<o3, R1>	0.8462	0.8077	0.7297
	<o3, o3>	0.8846	0.8077	0.8649
UCS	**<o3, Claude>**	**1.0000**	**1.0000**	**1.0000**
	<R1, Claude>	**1.0000**	**1.0000**	**1.0000**
	<R1, R1>	0.6667	0.6667	0.6842
	<R1, o3>	0.6667	0.5000	0.6316
	<o3, R1>	1.0000	0.8333	0.9474
	<o3, o3>	0.6667	0.6667	0.6842
A4F	**<o3, Claude>**	**1.0000**	**1.0000**	**1.0000**
	<R1, Claude>	0.8750	0.7500	0.8000
	<R1, R1>	0.8750	0.7500	0.8250
	<R1, o3>	0.8750	0.7500	0.8000
	<o3, R1>	0.8750	0.7500	0.8250
	<o3, o3>	<u>1.0000</u>	<u>0.8750</u>	<u>0.9250</u>
Overall	**<o3, Claude>**	**1.0000**	**1.0000**	**1.0000**
	<R1, Claude>	<u>0.9000</u>	<u>0.8500</u>	<u>0.8571</u>
	<R1, R1>	0.7250	0.6750	0.6842
	<R1, o3>	0.7250	0.6750	0.6842
	<o3, R1>	0.8750	0.8000	0.7895
	<o3, o3>	0.8750	0.8000	0.8571

the same RAG-extracted exemplars and syntax documentation as our pipeline to ensure a fair comparison.

For model combination comparisons, we explore five additional pairings for the planning and code generation stages, resulting in the variants shown in Table IV. Our chosen configuration — o3 for planning and Claude for code generation — serves as the default setting for PAT-Agent.

Results. From Table III, we observe that direct LLM-based generations are only able to obtain fully verified implementations for 37.5% to 52.5% of the systems, despite structured prompting

conditions. Furthermore, the strongest individual model still fails to compile in 25% of the cases. These failures are often due to incomplete condition handling, misinterpretation of requirements, or improper modeling abstractions for more complex systems.

In contrast, our pipeline achieves a 100% full-pass rate across all 40 systems. This result does not indicate flawless first-attempt generations, rather, it highlights the effectiveness of the integrated repair loop, which iteratively incorporates counterexamples and verification feedback to correct model errors until all specified requirements are satisfied, as shown in Table VI. A detailed breakdown of first-attempt and per-round performance is available in our GitHub repository [1], showing that multiple repair iterations were often necessary.

Table IV further demonstrates that our <o3, Claude> combination consistently outperforms all alternative model pairings across datasets. Across all datasets, the <o3, Claude> configuration achieves 1.0000 in compilation success rate, full-pass rate, and average pass rate, indicating that the generated systems not only compile successfully but also satisfy all user-specified requirements after repair. Other combinations show varying degrees of degradation. While <R1, Claude> performs relatively well overall (i.e., 0.9000 compilation and 0.8500 full-pass rate), it still fails to match the consistency of our pipeline. R1, when used as the planner, sometimes produces outputs that are ambiguous or insufficiently differentiated. For instance, it may fail to clearly separate constant identifiers from process names. This leads to conflicting definitions that confuse the Code Generation LLM, resulting in verification failures despite Claude's strong generative ability.

Beyond performance, we also note practical advantages in instruction-following: o3 reliably adheres to structured output formats (e.g., valid JSON), which simplifies integration into our user-facing interface. In contrast, R1 sometimes ignores prompt constraints or produces malformed outputs, making it less suitable for interactive settings where predictable formatting is crucial for usability.

D. Results of RQ2: Impact of Components

Settings. We perform ablations by disabling two key components of the PAT-Agent pipeline: (a) the Planning LLM module and (b) the Verification-Guided Repair Loop. Our goal is to assess how much each component contributes to the final verification performance. We evaluate these variants using the same set of metrics as in RQ1 — compilation success rate, full-pass rate, and average pass rate — across all datasets. We have the following variants:

- **Full Pipeline**: The complete PAT-Agent configuration, with both Planning and Repair components enabled.
- **Without Planning Model**: The detailed code generation plan produced by the Planning LLM is removed. Instead, the Code Generation LLM receives only the original natural language description as input. Other components, such as RAG-based example retrieval and syntax documentation, are still included to ensure fair comparison.

TABLE V: Ablation Study.

Dataset	Method	CSR	FPR	APR
PAT	**Full Pipeline**	**1.0000**	**1.0000**	**1.0000**
	Without Repair Loop	1.0000	0.8077	0.8378
	Without Planning Model	0.7308	0.5769	0.7162
	Without Both Components	0.7308	0.5385	0.6892
UCS	**Full Pipeline**	**1.0000**	**1.0000**	**1.0000**
	Without Repair Loop	1.0000	0.8333	0.9474
	Without Planning Model	0.6667	0.5000	0.6316
	Without Both Components	0.6667	0.1667	0.4211
A4F	**Full Pipeline**	**1.0000**	**1.0000**	**1.0000**
	Without Repair Loop	1.0000	0.5000	0.6750
	Without Planning Model	0.8750	0.7500	0.9000
	Without Both Components	0.8750	0.5000	0.6250
Overall	**Full Pipeline**	**1.0000**	**1.0000**	**1.0000**
	Without Repair Loop	1.0000	0.7500	0.8045
	Without Planning Model	0.7500	0.6000	0.7594
	Without Both Components	0.7500	0.4750	0.6316

TABLE VI: Breakdown of Repair Loop Impact.

Dataset	Repair Round	CSR	FPR	APR
PAT	Round 0	1.0000	0.8077	0.8378
	Round 1	1.0000	0.8077	0.8784
	Round 2	1.0000	0.8462	0.9324
	Round 5	**1.0000**	**1.0000**	**1.0000**
UCS	Round 0	1.0000	0.8333	0.9474
	Round 1	1.0000	0.8333	0.9474
	Round 2	1.0000	1.0000	1.0000
	Round 5	**1.0000**	**1.0000**	**1.0000**
A4F	Round 0	1.0000	0.5000	0.6750
	Round 1	1.0000	0.6250	0.8250
	Round 2	1.0000	0.8750	0.9750
	Round 5	**1.0000**	**1.0000**	**1.0000**
Overall	Round 0	1.0000	0.7500	0.8045
	Round 1	1.0000	0.7750	0.8722
	Round 2	1.0000	0.8750	0.9549
	Round 5	**1.0000**	**1.0000**	**1.0000**

- **Without Repair Loop**: The verification-guided repair process is disabled. The system only evaluates the initial model generated based on the planning output, without iterative correction based on verification feedback.
- **Without Both Components**: Both the Planning LLM and the Repair Loop are removed. The Code Generation LLM directly operates on the raw natural language input in a single-pass manner, but still benefits from RAG examples and syntax documentation.

Results. Table V shows that removing either the Planning LLM or the Repair Loop causes a significant drop in verification performance, with the most severe degradation occurring when both are disabled.

Removing the Repair Loop alone still maintains 100% compilation but causes the full-pass rate to drop to 0.7500 and average pass rate to 0.8045, highlighting that many models require verification-guided adjustments to satisfy all user-defined requirements. To further illustrate the impact of the repair loop, we provide a breakdown of performance across individual repair rounds in Table VI. This includes the initial
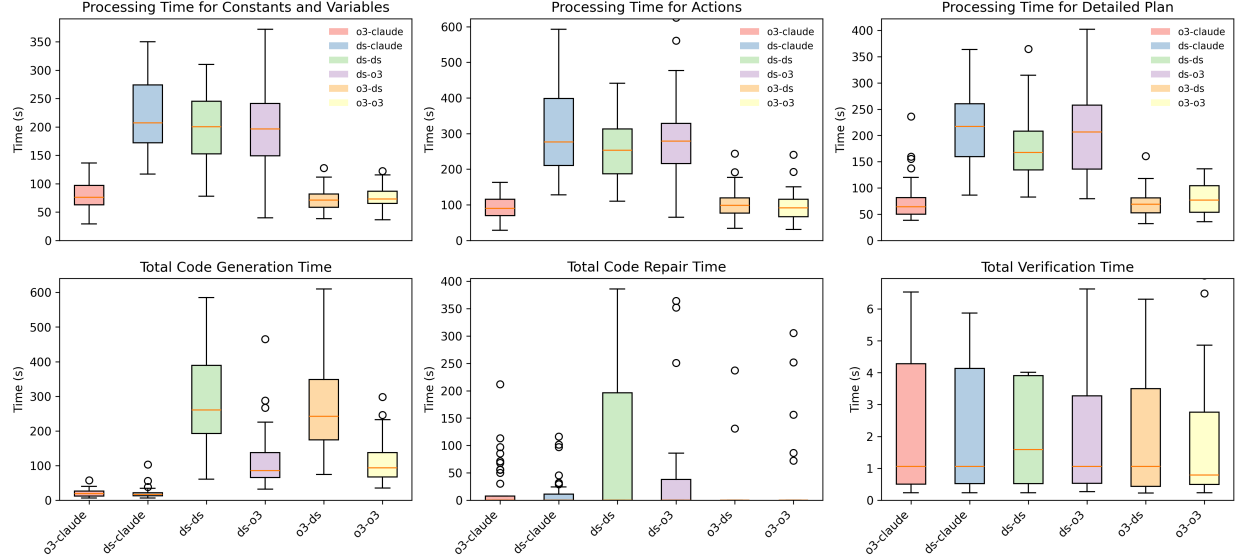
Fig. 3: Time Cost for Each Step across Pipelines.

generation (Round 0), intermediate repair rounds (1–2), and the final round (Round 5), demonstrating how iterative feedback progressively improves model correctness. The results show that the performance increases and finally goes to 100% during the repair loop.

On the other hand, excluding the Planning LLM has a substantial negative impact on both syntactic and semantic quality. The compilation success rate drops to 0.7500, indicating that without structured, semantics-aware planning, the same Code Generation LLM struggles to produce even syntactically valid PAT code in a quarter of the cases. Moreover, the full-pass rate drops to 0.6000, and the average pass rate falls to 0.7594, highlighting that even when the code compiles, it might fail to satisfy the requirements after repair. These results demonstrate the importance of the Planning LLM in structuring the input for code generation, ensuring not only syntax correctness but also alignment with user intent.

Overall, these results affirm the importance of both the Planning LLM and the Repair Loop: planning provides high-quality structural guidance, while repair ensures final verification success through iterative corrections.

E. Results of RQ3: Time Efficiency

Settings. We evaluate the time efficiency of different pipeline configurations by recording the wall-clock time for each processing step, including constant and variable analysis, action extraction, plan generation (all part of planning), code generation, repair, and verification. These measurements exclude user interaction time, which is assessed separately in the user study (Section V). Timing is computed only for systems that are eventually verified, so the number of samples varies across configurations. The detailed statistics are provided in our GitHub repository [1]. Figure 3 presents box plots of the

time taken by each processing step, with y-axes capped to improve visual clarity.

Results. Our pipeline configuration <o3, Claude> achieves not only the highest verification performance (as shown in RQ1) but also competitive efficiency across all stages. o3 consistently outperforms R1 in planning steps, reflecting both faster response time and more effective structure extraction. Claude exhibits the lowest latency in code generation among all models, confirming its fluency in producing valid PAT code.

For repair, the <o3, Claude> pipeline maintains low and stable runtimes, indicating that many systems are verified with only minor corrections. <R1, R1> and <R1, o3> exhibit generally longer repair times across their verified systems. In contrast, <o3, R1> and <o3, o3> show near-zero repair time for most of their successfully verified systems, suggesting that repair was often unnecessary. However, this efficiency must be considered alongside their lower overall verification coverage (as shown in RQ1), which means fewer systems benefited from repair in the first place. A small number of long-tail cases in these pipelines show elevated repair time, indicating that correction, when required, can occasionally be time-consuming. Verification time remains consistently low across all configurations, confirming that model checking contributes minimally to overall latency.

When aggregating the total time required to generate a verifiable system, <o3, Claude> achieves the lowest median runtime at 4.34 minutes per system. These results confirm that PAT-Agent achieves strong end-to-end performance with high time efficiency.

V. USER STUDY

To evaluate the usability and effectiveness of PAT-Agent's interactive mode in practical modeling scenarios, particularly

for those without formal methods expertise, we conducted a user study centered around natural language-based system modeling and formal verification tasks.

Settings. We recruited 20 participants with 2–8 years of computer science background. Among them, 30% had prior experience with PAT or CSP#, ranging from 1 to 5 years, ensuring the study included both novices and semi-experienced users. Participants were randomly assigned into two groups — Control Group (CG) and Experimental Group (EG) — with 70% novices in each group, and were numbered P1–P10 (CG) and P11–P20 (EG), respectively. All participants received a 10-minute tutorial on PAT and CSP#.

Both groups were also provided with (1) Clear natural language descriptions of the target systems and corresponding requirements. (2) The official PAT documentation. (3) A list of common modeling errors. (4) A well-formed PAT model example for reference.

Control Group (CG): Participants in the control group were instructed to use any resource (including LLM) except for PAT-Agent to generate PAT models from the given natural language system descriptions. They were encouraged to leverage the provided grammar guides and example errors to construct effective prompts and repair the generated code.

Experimental Group (EG): Participants in the experimental group used our PAT-Agent interface. Following the shared tutorial, they were given an additional walkthrough session that demonstrated how to use PAT-Agent. During the task, EG participants used PAT-Agent exclusively.

Modeling Tasks: To account for the expected difficulty faced by participants without prior PAT experience, we selected one relatively simple system from each of our three datasets, along with one more complex task.

- System 1 models a file system, where each file can be created, trashed, and protected.
- System 2 represents a trading scenario in which a buyer can purchase an item from a merchant if the offered price meets or exceeds the merchant's expectation.
- System 3 captures the classic missionaries and cannibals river-crossing puzzle.
- System 4 is a complex car system alluded in Section III.

Metrics. We evaluated the effectiveness and usability of PAT-Agent using both objective performance indicators and user feedback, as explained below.

Task Completion Time: The average time taken to complete modeling, measured in minutes and averaged across all systems.

Assertion Accuracy: The percentage of assertions written by each participant that were both syntactically correct and semantically aligned with the intended properties.

System Accuracy: The proportion of reference (ground-truth) assertions for each task that were correctly verified — i.e., the participant-generated model produced the expected verification outcome (`VALID` or `INVALID`) for each property.

Statistical Testing: We used the Mann–Whitney U test [24] to compare CG and EG. This nonparametric test requires no assumption of normally distributed data and is suited to small

TABLE VII: Comparison of CG and EG Performance. Statistical significance: * ($p < 0.05$), ** ($p < 0.01$), *** ($p < 0.001$).

Metric	EG Mean	CG Mean	p-value
Time (min)	12.85*	17.11	1.16E-02
Assertion Accuracy	0.9958***	0.7500	1.85E-07
System Accuracy	0.9688***	0.6633	2.66E-05

sample sizes and ordinal measures. All comparisons were two-tailed, with a significance threshold of $\alpha = 0.05$.

Results. Full user study outcomes are available in our GitHub repository [1]. We present the summarized user-study outcomes in Table VII, reporting the mean values for each metric in both control and experimental groups over the four system tasks, together with the corresponding significance results. Our principal findings are as follows:

- **Task Efficiency**: EG users, using PAT-Agent, completed tasks faster than CG users, showing improved efficiency and reduced cognitive overhead. RQ3 shows that our pipeline averages 4.34 minutes to generate a verifiable system. The difference suggests users spend time validating intermediate outputs and interpreting verification results via the interface.
- **Assertion Accuracy:** EG users achieved near-perfect assertion accuracy (0.9958), significantly outperforming CG. This reflects the framework's effectiveness in helping users formalize system properties correctly, which is an essential step for ensuring overall model correctness.
- **System Accuracy:** Models generated by EG users achieved significantly higher system accuracy compared to CG. For simpler tasks, EG consistently reached 1.0000 accuracy across all users. In CG, users with a formal methods background tended to perform more consistently across systems, while those without such experience struggled to produce verifiable models, particularly on system 4. In contrast, PAT-Agent enabled consistent success among EG users regardless of prior experience.
- **User Feedback:** To evaluate the interface design, we gathered user feedback on its clarity and ease of use. Several CG participants reported challenges in prompting LLMs and writing correct PAT code. In contrast, EG participants found the modular layout and structured tables intuitive, helping them understand how formal models are constructed. All EG users highlighted that the interface made it easy to inspect and modify specific model components. For example, P19 noted that it allowed convenient review of variable declarations and controlled adjustments. These responses suggest the interface effectively balances automation with user control.

VI. DISCUSSION

Auto-formalization systems that translate raw natural language directly into a target formal language have been proposed for mainstream notations such as TLA$^+$ [19], Z [31], or Isabelle theories [25]. In contrast, CSP# is comparatively niche and lacks public corpora or prior end-to-end translators on which a strong auto-formalization baseline could be built. To the best of our knowledge, no existing work attempts an

automatic NL → CSP# pipeline that covers concurrent constructs, shared variables, and refinement assertions. Establishing such a baseline would therefore require replicating (and fairly re-engineering) the very contribution that PAT-Agent itself introduces, defeating the purpose of comparative evaluation.

Recent verification studies often contrast their approach with program-repair frameworks that iteratively patch counterexamples. In our context, however, the combination of prompt engineering and large language models already achieves high requirement-satisfaction rates. Empirically, incorporating a separate repair pipeline yielded negligible improvements while adding substantial overhead. We therefore focus our analysis on synthesis quality rather than post-hoc repair, and leave generalized repair strategies as future work.

Finally, an important consideration is the generalization ability of the approach. While our implementation is built on CSP# and PAT, the workflow of planning, code generation, and verification-guided repair is not tied to this particular formalism, and has the potential to generalize to other model checkers [3], [12], [20], and modeling languages such as PEPA and Petri nets [15], [27].

For CSP-style variants (e.g., tock-CSP [29]) and for model checkers such as FDR [12], most prompts and guidance are reusable since the analysis tasks are fundamentally similar, with some effort needed to provide corresponding syntax documentation and example corpora, and process verification feedback. The same workflow can also extend to other languages, with prompt templates and feedback integration adapted to their primitives.

VII. THREATS TO VALIDITY

While **PAT-Agent** demonstrates promising results in autoformalization, several limitations remain.
Internal validity. The effectiveness of PAT-Agent depends on the quality and clarity of the natural language input: ambiguous or incomplete specifications can produce incorrect or underspecified formal models. Moreover, highly entangled control logic or intricate data-dependent behaviors may not decompose cleanly into the semantic-prompt structure we employ, limiting the ability of LLMs to synthesize faithful models. In addition, like all LLM-based methods, PAT-Agent inherits stochasticity in generation. We mitigate this by fixing LLM versions with timestamps and incorporating multiple repair iterations guided by PAT feedback, but stochasticity remains an inherent limitation.
External validity. Our rule-based repair heuristics are tailored to the violations observed in our datasets, primarily safety and liveness errors. While effective in these settings, they may not generalize to other types of failures, where alternative forms of guidance could be required.

VIII. RELATED WORK

Autoformalization and Repair. Being able to convert the ambiguous natural language descriptions into unambiguous, executable formal specifications is essential in ensuring the success of verifications. Traditionally, this conversion is done manually, with a high requirement for the expertise of the programmer. The textual understanding capability of LLMs inspires research in autoformalization through LLMs. For example, research efforts have been dedicated into translating natural language instructions into Linear Temporal Logic (LTL) using LLMs [8], [11]. While LTL is a basic building block of many specification languages, including PAT, it is only one type of supported assertions in PAT-Agent. The PAT model synthesis task involves constructing and improving models until they pass the given assertions, which is much more complicated. The other applications of LLMs in formal verification include the synthesis of program specifications [36], [37], and formal proof synthesis and repair [10], [21], [6].

On the other hand, existing research on synthesizing system-level models in various specification languages such as Alloy [16] and B-method [5] are generally limited in depth. Compared to PAT-Agent, these works target the synthesis of simpler system-level models or fine-grained formulae. In addition, they rely on directly mapping natural language to formal code, which has been demonstrated to be of limited effectiveness [5]. In contrast, PAT-Agent offers a more structured and effective solution.
Neuro-symbolic Methods. Neuro-symbolic AI combines the pattern-matching power of neural networks with the rigour and compositionality of symbolic reasoning [23]. Recent work attaches large encoders to automated theorem provers: retrieval-augmented LeanDojo boosts success rates on formal proofs [38], and ProofWriter induces natural-language rulebases while recovering proofs [34]. Closer to formal verification, FVEL converts real-world code into Isabelle specifications and calls an LLM-guided prover loop [40]. Unlike these efforts, we target model checking rather than interactive theorem proving and show how an LLM can populate PAT's process, after which PAT discharges temporal-logic obligations automatically.

Another line of work generates verified code from formalized requirements [4]. In contrast, PAT-Agent assumes both a system description and requirements: it first constructs a model from the description and then verifies it against the requirements.

IX. CONCLUSION AND FUTURE WORK

We presented **PAT-Agent**, a novel framework that integrates large language models with formal verification to automate the synthesis, validation, and repair of system models from natural language descriptions. By leveraging structured semantic prompts and verification-guided feedback, PAT-Agent reliably produces syntactically and semantically correct models, achieving a 100% verification success rate across 40 tasks. We plan to expand the semantic prompt library to support more advanced modeling paradigms, such as probabilistic, real-time, and hybrid systems. We also aim to generalize the PAT-Agent architecture to other formal methods, including Event-B and Alloy, thereby broadening its applicability.

REFERENCES

[1] Github, 2025. URL: https://github.com/ZuoXinyue/PAT-Agent.

[2] Ayodeji James Akande, Zhe Hou, Ernest Foo, and Qinyi Li. Ltl-based runtime verification framework for cyber-attack anomaly prediction in cyber–physical systems. *Computers & Security*, 155:104455, 2025.

[3] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 200–236. Springer, 2004.

[4] Yufan Cai, Zhe Hou, David Sanán, Xiaokun Luan, Yun Lin, Jun Sun, and Jin Song Dong. Automated program refinement: Guide and verify code large language model with refinement calculus. *Proceedings of the ACM on Programming Languages*, 9(POPL):2057–2089, 2025.

[5] Alfredo Capozucca, Daniil Yampolskyi, Alexander Goldberg, and Maximiliano Cristiá. Do ai assistants help students write formal specifications? a study with chatgpt and the b-method. *arXiv preprint arXiv:2502.07789*, 2025.

[6] Pedro Carrott, Nuno Saavedra, Kyle Thompson, Sorin Lerner, João F Ferreira, and Emily First. Coqpyt: proof navigation in python in the era of llms. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 637–641, 2024.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[8] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification*, pages 383–396. Springer, 2023.

[9] Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-act: Improving planning of agents for long-horizon tasks. *arXiv preprint arXiv:2503.09572*, 2025. Accepted at ICML 2025.

[10] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1229–1241, 2023.

[11] Francesco Fuggitti and Tathagata Chakraborti. Nl2ltl–a python package for converting natural language (nl) instructions to linear temporal logic (ltl) formulas. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 16428–16430, 2023.

[12] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and Andrew W Roscoe. Fdr3—a modern refinement checker for csp. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 187–201. Springer, 2014.

[13] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[14] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[15] Jane Hillston. *PEPA: Performance enhanced process algebra*. University of Edinburgh, Department of Computer Science, 1993.

[16] Yang Hong, Shan Jiang, Yulei Fu, and Sarfraz Khurshid. On the effectiveness of large language models in writing alloy formulas. *arXiv preprint arXiv:2502.15441*, 2025.

[17] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

[18] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024.

[19] Leslie Lamport. *Specifying systems*, volume 388. Addison-Wesley Boston, 2002.

[20] Michael Leuschel and Michael Butler. Prob: A model checker for b. In *International symposium of formal methods europe*, pages 855–874. Springer, 2003.

[21] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. Proof automation with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1509–1520, 2024.

[22] Nuno Macedo, Alcino Cunha, José Pereira, Renato Carvalho, Ricardo Silva, Ana CR Paiva, Miguel S Ramalho, and Daniel Silva. Sharing and learning alloy on the web. *arXiv preprint arXiv:1907.02275*, 2019.

[23] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems*, pages 3749–3759, 2018.

[24] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[25] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[26] OpenAI. Openai o3 and o4-mini system card. https://cdn.openai.com/pdf/2221c875-02dc-4789-800b-e7758f3722c1/o3-and-o4-mini-system-card.pdf, 2025. Accessed: 2025-05-28.

[27] Carl Adam Petri. Communication with automata. 1966.

[28] Andrew W Roscoe. *Understanding concurrent systems*. Springer Science & Business Media, 2010.

[29] Anthony Roscoe. The theory and practice of concurrency. 1998.

[30] Ling Shi, Yongxin Zhao, Yang Liu, Jun , Jin Song Dong, and Shengchao Qin. A utp semantics for communicating processes with shared variables. In Lindsay Groves and Jing , editors, *Formal Methods and Software Engineering*, pages 215–230, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[31] J Michael Spivey and Jean-Raymond Abrial. *The Z notation*, volume 29. Prentice Hall Hemel Hempstead, 1992.

[32] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using stateful timed csp. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):1–29, 2013.

[33] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 709–714, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[34] Øyvind Tafjord, Bhavana Dalvi, and Peter Clark. Proofwriter: Generating implications, proofs, and abductive statements over natural language. In *Findings of ACL-IJCNLP*, pages 3621–3634, 2021.

[35] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. *arXiv preprint arXiv:2203.05132*, 2022.

[36] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*, pages 302–328. Springer, 2024.

[37] Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. *arXiv preprint arXiv:2310.04870*, 2023.

[38] Kaiyu Yang, Jia Deng, and Danqi Chen. Theorem proving with retrieval-augmented language models. *arXiv preprint arXiv:2306.15626*, 2023.

[39] Jia-Yu Yao, Kun-Peng Ning, Zhen-Hui Liu, Mu-Nan Ning, and Li Yuan. Llm lies: Hallucinations are not bugs, but features as adversarial examples. *arXiv preprint arXiv:2310.01469*, 2023.

[40] Yuchen Zhang, Xiaoyu Wang, Qian Li, et al. Fvel: An interactive formal verification environment with large language models. In *Advances in Neural Information Processing Systems*, 2024. To appear.