On Effectiveness of Formal Model Repair by Large Language Models

Sebastião Carvalho
Instituto Superior Técnico, Portugal
sebastiaovscarvalho@tecnico.ulisboa.pt

Tsutomu Kobayashi
Japan Aerospace Exploration Agency, Japan
kobayashi.tsutomu@jaxa.jp

Fuyuki Ishikawa
National Institute of Informatics, Japan
f-ishikawa@nii.ac.jp

***Abstract*—The use of formal methods is a significant contribution to developing trustworthy software; however, it can be a complex task. For this, automation with generative artificial intelligence models, such as Large Language Models (LLMs), is considered a promising approach. We studied the use of LLMs to generate repairs for faulty formal models of the Event-B formalism. To repair faulty Event-B models, we propose a System Prompt that contains constraints on how to suggest repairs that respect the syntax and rules of the Event-B language. We also propose Retry Prompts, a type of prompt that aims to refine a repair suggested by an LLM by highlighting errors in previous responses. To evaluate our method, we developed a tool that generates faulty models (mutants) from existing correct models by removing a single action or guard predicate. The tool then interacts with an LLM to obtain a suggested repair for the mutant model. After modifying the model according to the suggestions from the LLM, we evaluate the correctness of the modified model. The results demonstrate that using Retry Prompts significantly increases the success rate of the suggested repairs, with over 80% of the faulty models in our dataset being successfully repaired. The results also indicated directions of possible future improvements, such as combining Generative AI with formal approaches to repair failing cases.**

***Index Terms*—Theorem Proving, Large Language Models, Event-B**

I. Introduction

Due to the rise of Large Language Models (LLMs), the cost of generating software artifacts has dramatically dropped. In this context, specifications are gaining much importance because they play a crucial role in generating and verifying artifacts in later phases, such as program code. Although LLMs also generate specifications effectively, due to their nature, the generated specifications often lack consistency.

To address this problem, the combination of LLMs and formal methods (FMs) is being studied [1]. Such approaches let LLMs generate specifications in formal language, then use FM tools to verify them rigorously.

We focus on *iterative small repairs*, rather than the "specify-all-verify-all" approach used in other existing techniques. We believe that this approach is promising because the process of AI-assisted construction often involves generation by LLM and then repeated small corrections by human developers.

From the perspective of LLM research, our approach can be seen as a way to reduce the hallucination and make the safety of generated artifacts more explainable. On the other hand, for FM research, we can expect a significant reduction in the cost of modeling and verification by using LLM.

This paper aims to investigate the extent to which LLMs can aid in repairing formal specifications. To this end, we combine ChatGPT with the Event-B [2] method, which is designed for *incremental* construction and verification of formal models. The typical feedback from Event-B's prover tool indicates which event is violating a particular safety predicate, thereby providing fault localization. We leverage this feature in our following approach: (1) We artificially construct a faulty Event-B model by mutating a correct one, (2) we run the Event-B prover to get the feedback, (3) we ask the LLM to repair the model using special prompts that includes the faulty model and the prover's feedback, and (4) we rerun the Event-B tool to check whether the LLM's repair is correct.

As a result, we found that our special prompts are effective in allowing LLMs to repair faulty Event-B models. We also identified future research directions for further improving the repair ability of LLMs.

Our contributions are summarized as follows:

- Prompts[1] for specifications repair,
- Tool that intermediates LLM and the Event-B prover,
- Experiments to measure the effectiveness of LLM in repairing specifications.

The rest of the paper is organized as follows: Section II introduces related work and Event-B. In Sections III and IV, we elaborate on our approach and experiments, respectively. We finally discuss and conclude this study in Section V.

II. Background and Related Work

A. Preliminary: Event-B

Event-B [2] is a formal method for system-level modeling and analysis based on first-order logic and set theory[2]. The primary components of a model include the *INVARIANTS* clause and the *EVENTS* clause (Fig. 1). The INVARIANTS clause declares predicates that should be inductive invariants, often expressing requirements such as safety. The EVENTS clause reflects the behavior of the system in a parametric guarded-command style; the conjunction of predicates in the *WHERE* clause is the necessary condition for the event to occur (*guards*), and the *THEN* clause declares the state changes of each variable (*actions*). An action can be specified as

[1]Prompts available at https://github.com/trarse-nii/ASYDE--2025

[2]See https://stups.hhu-hosting.de/handbook/rodin/current/html/ for detailed information on the formalism.

```
VARIABLES
○     n
INVARIANTS
○     type:  n ∈ N
○     capacity_limit:  n ≤ n_capacity
VARIANTS
○ n_capacity - n
EVENTS
○ INITIALISATION:
   THEN
   ○  n_init:      n ≔ 0
    END
○ enter: [ convergent ]
   WHERE
   ○  vacant:     n < n_capacity
   THEN
   ○  inc_n:      n ≔ n + 1
    END
○ leave: ...
END
```

Fig. 1. Event-B model of cars example

a nondeterministic assignment using a *before-after predicate* $BAP(v, v')$, that describes the relation between the before-value (v) and the after-value (v'). There is also a syntax sugar for a deterministic assignment, e.g., $v := \text{new\_value}$, which corresponds to a before-after predicate $v' = \text{new\_value}$. There are rules of variable occurrences in predicates. For example, after-state variables (v') cannot occur in guard predicates because guards are conditions on the before-state.

The modeler can also declare the *termination* of an event by annotating it with the keyword 'convergent'. The *VARIANT* clause states that its value should be decreased with every occurrence of convergent events. For instance, after an occurrence of event enter, the value of the variant clause is $\mathsf{n\_capacity} - (\mathsf{n} + 1)$, which is smaller than its value before the occurrence ($\mathsf{n\_capacity} - \mathsf{n}$).

The primary way of verification in Event-B is theorem proving. The consistency of the model *(Proof Obligations (POs))*, i.e., the properties to prove are generated from the model contents using patterns. For example, the *invariant preservation (INV)* PO is one of the primary POs generated for each pair of an invariant predicate and an event. INV PO about invariant i and event e (labeled as e/i/INV) corresponds to the following formula:

$$I(v) \wedge G_e(v, p) \wedge B_e(v, p, v') \wedge \ldots \implies i(v'),$$

where I, G_e, and B_e denote the conjunction of all invariant predicates of the model, the guards of e, and the before-after predicates of e, respectively. By proving the validity of this formula, the user can be confident that event e preserves the property of i. For instance, the formula corresponding to the PO enter/capacity_limit/INV (i.e., Does the number of cars never exceed the capacity when a car enters the building that is not full yet?) is:

$$\mathsf{n} \in \mathbb{N} \wedge \mathsf{n} \leq \mathsf{n\_capacity} \wedge \mathsf{n} < \mathsf{n\_capacity} \wedge \mathsf{n}' = \mathsf{n} + 1 \implies \mathsf{n}' \leq \mathsf{n\_capacity}.$$

This formula is valid.

The Rodin Platform [3] is an extensible development environment for seamlessly integrating modeling and verification in Event-B. Once the user saves a model, it checks types and grammar, generates POs, applies automatic provers, and shows which POs it failed to prove (discharge) automatically. Since POs not discharged yet indicate a possibility of problems in the model, for each PO not discharged, the user attempts to prove it manually and modifies the model if issues are found. This process is repeated until all POs are discharged.

Typical problems in Event-B models include specifying invariants or guards that are too weak or too strong. If an invariant is too weak (e.g., if invariant capacity_limit is $\mathsf{n} \leq 2 * \mathsf{n\_capacity}$), then the hypothesis of INV PO becomes too weak to be discharged. If an invariant is too strong (e.g., if invariant capacity_limit is $\mathsf{n} < \mathsf{n\_capacity}$), then the goal of INV PO becomes too strong to be discharged. If a guard is too weak (e.g., if guard vacant of event enter is $\top$), then the hypothesis of INV PO becomes too weak to be discharged. If a guard is too strong (e.g., if guard vacant of event enter is $\perp$), then it is no problem from the PO viewpoint, but it may be problematic from the validation viewpoint; for example, specifying $\perp$ in a guard means the event should never occur, which does not make sense in most cases.

B. Related Work

1) Active Effort on Generation: Code generation has been one of the main tasks supported by LLM [4]. This trend suggests that similar support will emerge for the construction of formal models using LLM. The work of Wu et al. demonstrated the ability of LLM to translate mathematical problems in a natural language into specifications in Isabelle/HOL [5]. However, the correct outcome is limited to around 35% at most, although the potential of automation itself is surprising.

The LeanReasoner [6] is an LLM-based tool for reasoning problems written in natural language, such as those from the FOLIO dataset [7]. From a natural language document of a problem, LeanReasoner formalizes it in the Lean theorem prover and tries to construct a proof of the theorem. LeanReasoner's fine-tuning and prompts achieve a near-perfect accuracy for problems in datasets. However, the target here is reasoning problems, rather than system models with complex behavior. Moreover, it does not deal with model repair.

We envision that it will become easier to obtain "mostly correct" formal models of software systems that reflect the necessary information in the natural language source document. We therefore focus on the problem of model repair to address small faults, for example, missing constraints in the source document or generated formal models.

2) Model Repair: The model repair problem or addressing failed proofs has been a significant issue in the practice of formal methods. We may start by considering possible repair directions and then focus on the concrete repair task for each direction. For the former, Hoang [8] classified causes of failed proofs in Event-B and provided guidelines for interpreting failed proofs. Our interest in this paper lies in the latter part of constructing a repair for the given proof failure.

Schmidt et al. [9] proposed an interactive workflow for repairing B models by combining templates and traces from a model checker. Cai et al. [10] proposed a tool that generates candidate fixes from traces and uses a learning-based candidate evaluator. We had another approach based on the deductive derivation and simplification of missing conditions [11].

We similarly focus on the concrete repair problem, but we evaluate the potential of the LLM-based approach.

III. Approach

The process of repairing Event-B models receives an incorrect or incomplete model as input, and constructs a correct model as output. We propose an LLM-based repair process.

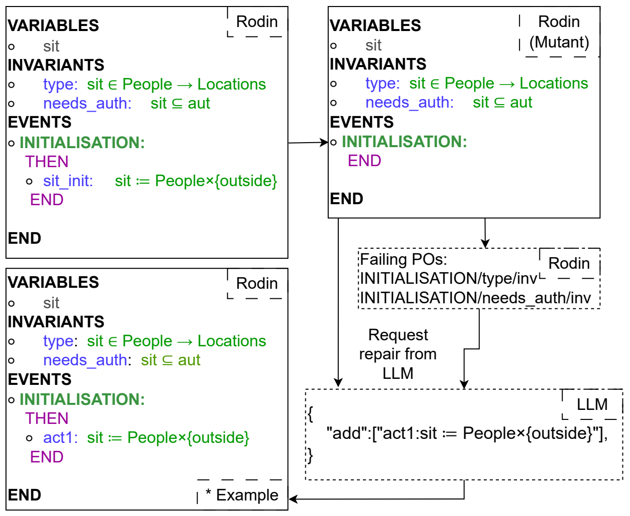

Fig. 2. Example of LLM-based repair

To evaluate the effectiveness of LLM in generating repairs for faulty models, we take a process that creates faulty models (*mutants*) from correct ones and then attempts to repair them using LLM suggestions (Figure 2). This process is implemented as a plugin for the Rodin platform. The evaluation process is as shown in Figure 4. We also implemented what we call the GPT server, which acts as a middleman between Rodin and the LLM (i.e, OpenAI API).

The process of communication between Rodin and the Large Language Model is as described in Figure 5.

The response obtained from the LLM should be in JSON format with the following schema:

```
{"add" : ["<String>"], "remove":
["<String>"]}
```

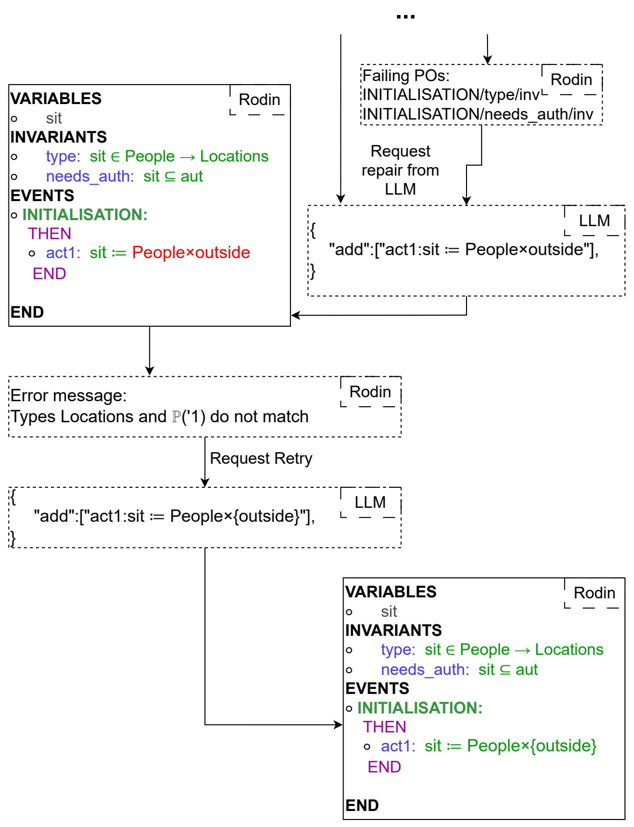

Fig. 3. Example of application of retry

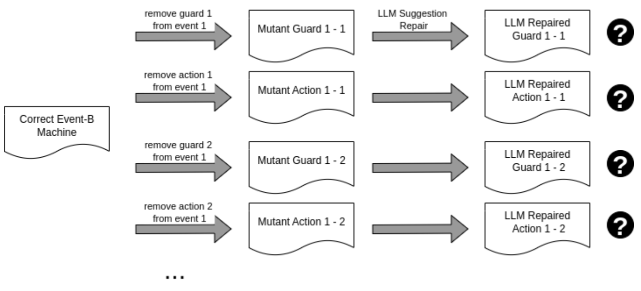

Fig. 4. Evaluation process using mutants

After obtaining a response from the GPT Server, our Rodin plugin attempts to interpret it and apply the suggested repairs. To simplify this process, we guide the LLM to suggest only additions or removals of guards, invariants, or actions. Other operations, such as adding or removing an event, are currently outside the scope of the plugin, because this type of operation is not the repair the user is usually looking for, and allowing them would give too much freedom to the LLM to easily solve the problem by removing the whole event instead of repairing.

We then classify the model obtained by applying the suggestion from LLM into one of the following categories, according to whether the prover of Rodin succeeded in discharging POs:

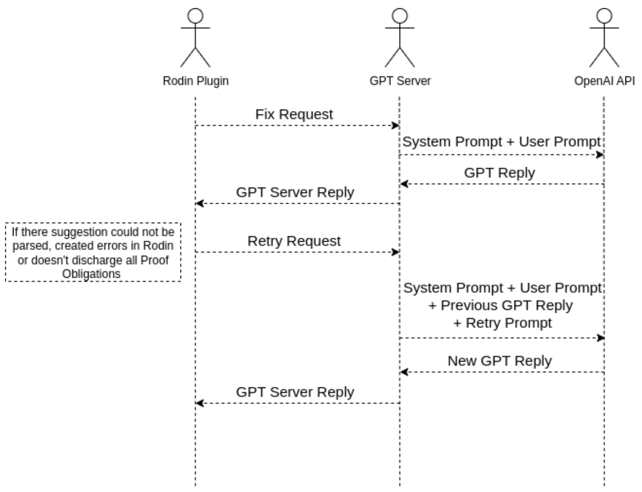
Rodin Plugin
GPT Server
OpenAI API
Fix Request
System Prompt + User Prompt
GPT Reply
GPT Server Reply
If there suggestion could not be parsed, created errors in Rodin or doesn't discharge all Proof Obligations
Retry Request
System Prompt + User Prompt + Previous GPT Reply + Retry Prompt
New GPT Reply
GPT Server Reply

Fig. 5. LLM Suggestion Process

- *Solved* - All POs are discharged, no errors are reported
- *Error* - Rodin reported syntax errors, type errors, or violations of variables' occurrence rules
- *Partial* - Discharged some POs but not all of them
- *Indifferent* - Either didn't solve any POs, or discharged some POs but created more not discharged ones
- *Invalid* - The operation proposed by GPT is not supported by our plugin, or is not in the correct response format to be parsed by the plugin

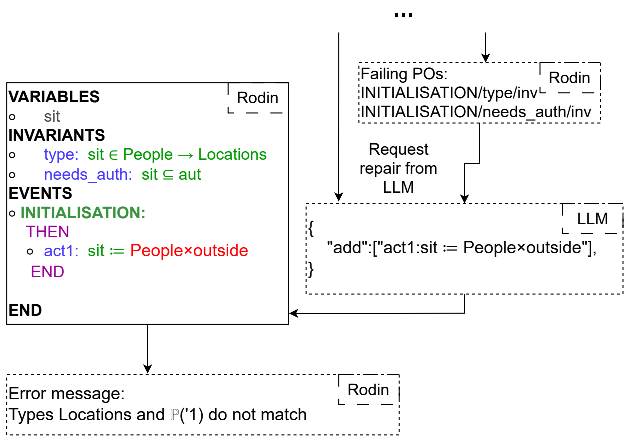

Fig. 6. Example of Error classification

The System Prompt given to LLM contains a brief description of the task, a specification of the response format, and constraints on the responses. The constraints in the system prompt can be classified into the following categories:

- *Formatting And Supported Operations Constraints (C1)* - These constraints give extra guidance on the response format and which operations can be suggested by the LLM. An example of a constraint in this set is "Each entry in the JSON schema should only contain an array of Event-B predicates, and not plain text inside them."

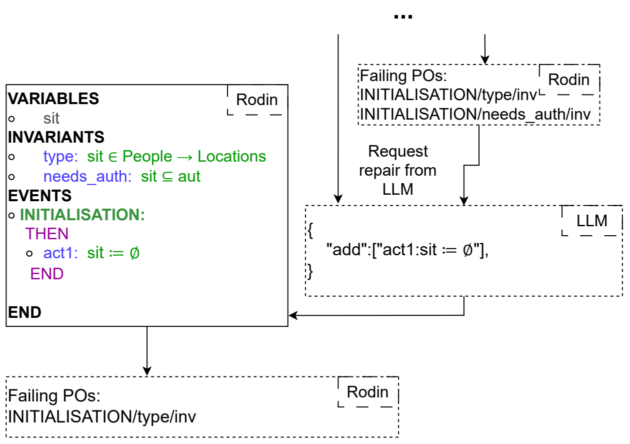

Fig. 7. Example of Partial classification

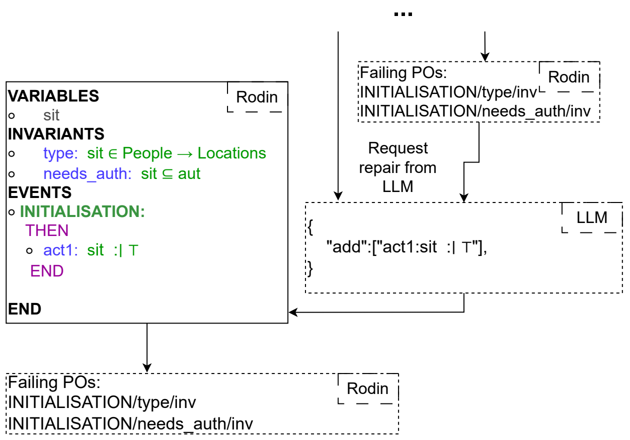

Fig. 8. Example of Indifferent classification

- *Syntax Constraints (C2)* - These constraints provide information on Event-B syntax rules and aim to mitigate the presence of syntax errors. An example of a constraint in this constraint set is "The syntax for functional relations is as follows: - ordered pairs: $(x \mapsto y)$ - relations and functions (as sets of ordered pairs): $\{x0 \mapsto y0, x1 \mapsto y1, ...\}$"
- *Variable Occurrence Rule Constraints (C3)* - These constraints provide information to prevent violations of variables' occurrence rules, including the usage of undefined variables in predicates. An example of a constraint in this constraint set is "You are only allowed to use the parameters and variables provided."

The User Prompt, used in conjunction with the System Prompt, contains a list of the not-discharged POs, the Event-B model we want to repair, and lists of variables and parameters that the LLM can utilize in its repair.

The retry prompt can be one of 3 different types:

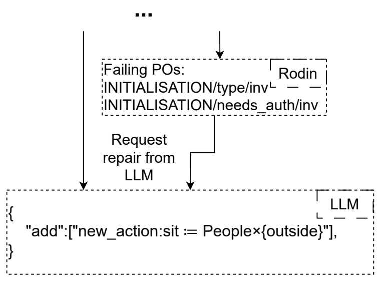
...
Failing POs:
INITIALISATION/type/inv
INITIALISATION/needs_auth/inv
Rodin
Request repair from LLM
LLM
{
"add":["new_action:sit := People×{outside}"],
}

Fig. 9. Example of Invalid classification

- *Error Retry Prompt* - If the previous repair made a model classified as *Error*
- *Failing PO Retry Prompt* - If the previous repair made a model classified as *Partial* or *Indifferent*
- *Wrong Format or Unsupported Operations Retry Prompt* - If the previous repair made a model classified as *Invalid*

For evaluation, we used the following correct models[3] as bases of mutants (Table I):

- *Cars Over Bridge* [2, Chapter 2]. This system models the traffic control of cars on a one-way bridge between two islands, utilizing traffic lights and sensors.
- *Bounded Re-Transmission Protocol* [2, Chapter 6]. This system models a file transfer protocol with fault tolerance in unreliable transfer channels.
- *Location Access Controller* [2, Chapter 16]. This system models people's movement between rooms in a building. The movement is restricted to comply with the security rules.

IV. Experiments

We carried out experiments using the "o3-mini" from the OpenAI GPT family, with the reasoning effort parameter set to "high". All other parameters were set to their default value.

A. RQ1. Effect of Completeness of Guidance

To assess the effectiveness of the System Prompt developed, we compared the usage of LLMs with the complete System Prompt against a subset of the System Prompt. In particular, the following subsets were employed:

- $\{C1, C2, C3\}$ (All Constraints)
- $\{C2, C3\}$ (Formatting And Supported Operations Constraints Removed)
- $\{C1, C3\}$ (Syntax Constraints Removed)
- $\{C2, C3\}$ (Variable Occurrence Rule Constraints Removed)

We compared the usage of the complete System Prompt against incomplete System Prompts *without* Retry Prompts in Table II, and *with* Retry Prompts in Table III, respectively (R denotes the usage of Retry Prompts).

[3]The original models are available at https://web-archive.southampton.ac.uk/deploy-eprints.ecs.soton.ac.uk/view/subjects/deploy=5Ftraining=5Feventb.html

To determine the distinctions between employing a complete System Prompt and an incomplete one, we examined the unsuccessful instances associated with each of the incomplete System Prompts and identified examples absent from the results generated by the complete System Prompt.

1) Formatting And Supported Operations Constraints Removed: In this subset, the number of *Invalid* cases is higher than the number of *Solved* cases. This is expected, since the *Formatting And Supported Operations Constraints* aim to force the replies to be in the correct response format and be on the list of supported operations (i.e., adding or removing guards, actions, or invariants). Examples include:

- Wrong format for reply - either in full text like
  ```
  "In EVENT RCV\_current\_data: add
  guard 'grd4 : l = FALSE'"
  ```
  or in an incorrect format such as
  ```
  {"add" : ["o grd1 : r_st = working"],
  "remove": []}
  ```
- Unsupported operations - e.g., removing a variant
  ```
  {"add" : [], "remove": ["VARIANTS\n
  o{success, failure,s_st}"]}
  ```

2) Syntax Constraints Removed: In this subset, examining the *Error* cases, we found a significant presence of syntax errors. For example:

- LLM suggests expressions not present in Event-B. For instance, although if-then-else expression is not supported in Event-B language, LLM suggests the following:
  ```
  {"add" : ["act4:B := (if il_out_10 =
  TRUE then b + 1 else b + 2)"],
  "remove": []}
  ```
- Wrong syntax for other expressions, such as:
  ```
  {"add" : [], "remove": ["inv0:{success,
  failure,s_st}"]}
  ```
  In this case, the invariant is expressed as a set, while it should be a predicate.

3) Variable Occurrence Rule Constraints Removed: The *Error* cases appeared using this subset include:

- Adding new variables, such as:
  ```
  {"add" : ["inv2:measure ∈ ℕ, act2:
  measure := 0"], "remove": []}
  ```
  In this case, the LLM attempts to use a variable 'measure', which is not declared in the original model.

B. RQ2. Effect of Retry

We assessed the efficacy of the Retry Prompts by comparing the use of LLMs employing Retry Prompts, with a maximum of two retry attempts, to LLMs without Retry Prompts.

We analyzed the utilization of Retry Prompts against the absence of Retry Prompts with *complete* System Prompts in Table IV and with *incomplete* System Prompts in Table V, respectively.

From both tables, we observed a significant increase in the number of Solved cases. In some cases, there is also an increase in the number of Error and Indifferent cases.

Name of Original Model	Number of Mutants
Cars Over Bridge	151
Bounded Re-Transmission Protocol	107
Location Access Controller	95

TABLE I
NUMBERS OF MUTANTS IN DATASET

Prompts used	Solved	Partial	Error	Indifferent	Invalid	Total
{C1, C2, C3}	252	5	41	48	7	353
{C2, C3}	123	3	25	29	173	353
{C1, C3}	231	7	54	56	5	353
{C1, C2}	241	5	39	52	16	353

TABLE II
COMPARISON OF INCOMPLETE SYSTEM PROMPTS WITH COMPLETE SYSTEM PROMPT (WITHOUT RETRY)

Prompts used	Solved	Partial	Error	Indifferent	Invalid	Total
{C1, C2, C3, R}	295	7	38	12	1	353
{C2, C3, R}	230	1	28	32	62	353
{C1, C3, R}	258	2	63	29	1	353
{C1, C2, R}	295	5	34	16	3	353

TABLE III
COMPARISON OF INCOMPLETE SYSTEM PROMPTS WITH COMPLETE SYSTEM PROMPT (WITH RETRY)

Prompts used	Solved	Partial	Error	Indifferent	Invalid	Total
{C1, C2, C3}	252	5	41	48	7	353
{C1, C2, C3, R}	295	7	38	12	1	353

TABLE IV
COMPARISON OF RETRY VS NO RETRY USING COMPLETE SYSTEM PROMPT

C. RQ3. Analysis of Failed Cases

To examine the effectiveness of our System Prompt, we analyzed the results obtained with all constraints without retry (i.e., $\{C1, C2, C3\}$) and identified patterns of non-solved cases (i.e., Partial, Error, Indifferent, or Invalid).

1) Wrong Response Format (result: Invalid):

- Updating Event-B code instead of using add/remove in JSON. An example is shown in Figure 2, where the code is labeled with "* Example".

2) Unsupported Operations (result: Invalid):

- Updating Variant (which is not supported by our Rodin plugin) :

```
{"add" : [], "remove": ["VARIANT
{success, failure,s_st}"]}
```

or

```
{"add" : [], "remove": ["VARIANT
{success, failure,r_st}"]}
```

3) Adding Invariant (result: Indifferent or Partial): While adding or removing invariants can help discharge POs, they can also create undischarged POs for other events. An example of such a case is shown in Table VI.

4) Adding Action Instead of Guard (result: Indifferent or Partial): In some cases, LLMs are not able to discern if a predicate should be added as a condition for the occurrence of the event or as an effect of it, which can cause some Proof Obligations not to be discharged. An example of such a case is shown in Table VI.

5) Adding Weak Guard Predicate (result: Indifferent or Partial): Adding a weak guard may not significantly constrain the conditions under which the event occurs, but at the same time, it may not guarantee the maintenance of the invariants. An example of such a case is shown in Table VI.

6) Specific Cases For Specific Event-B Problems (result: Error):

- Wrong capitalization of variables: This is a case that occurred in a model with two different variables, 'red' and 'RED', of different types. The LLM often returns suggestions that change occurrences of 'red' to 'RED', which causes errors in the model.

V. DISCUSSION AND CONCLUDING REMARKS

A. Remarkable results

1) RQ1. Effect of Completeness of Guidance: Table II shows that, without retry attempts, using the complete System Prompt yielded better results than using an incomplete System Prompt. However, Table III shows that, with Retry Prompts, the complete System Prompt did not outperform the System Prompt without the *Variable Occurrence Rule Constraints*.

We conclude that using all constraints yields better results than using only part of them. Additionally, although using Retry Prompts improves results, using only System Prompts takes less time on average, as it requires fewer interactions with the LLMs. Lastly, although the numbers indicate that having {C1, C2, C3} is effective, there were some exceptional cases (e.g., mutants solved with {C1, C3} but not solved when using {C1, C2, C3}). We discuss the details of such cases in Section V-B.

Prompts used	Solved	Partial	Error	Indifferent	Invalid	Total
{C2, C3}	123	3	25	29	173	353
{C2, C3, R}	230	1	28	32	62	353
{C1, C3}	231	7	54	56	5	353
{C1, C3, R}	258	2	63	29	1	353
{C1, C2}	241	5	39	52	16	353
{C1, C2, R}	295	5	34	16	3	353

TABLE V
COMPARISON OF RETRY VS NO RETRY USING INCOMPLETE SYSTEM PROMPT

Pattern Detected	Clause Removed	Suggestion
Adding Invariant	$grd3 : a = 0$	$inv27 : ml\_pass = 1 \iff a = 0$
Adding Action Instead of Guard	$grd5 : ml\_out\_10 = FALSE$	$act4 : ml\_out\_10 := FALSE$
Adding Weak Guard Predicate	$grd1 : IL\_OUT\_SR = on$	$grd2 : B > 0$

TABLE VI
PATTERNS DETECTED FOR CASES CLASSIFIED AS INDIFFERENT OR PARTIAL

2) RQ2. Effect of Retry: Tables IV and V show that when using Retry Prompts, the number of Solved cases increased for all types of System Prompts. However, in some cases, the number of Error and Indifferent cases increased. Additionally, we observed that the improvement was greater for cases where an incomplete System Prompt was used.

We can conclude that using Retry Prompts is effective, even though they require more interactions with the LLMs, which may lead to an increase in costs and time spent to obtain a response classified as Solved. Lastly, exceptional cases (e.g mutants solved with {C1, C2, C3} but not solved with {C1, C2, C3, R})) may exist, but only demonstrate the nondeterministic nature of LLMs, since the constraints present in the System Prompt are the same, and the reply is different.

3) RQ3. Analysis of Failed Cases: From examining both the *Wrong Response Format* and *Unsupported Operations* cases, we observed that even though the system prompt contains constraints that specify guidelines for following the response format and avoiding unsupported operations, suggestions that violate these constraints still occurred. This may occur due to the LLM's nature, which sometimes causes it to overlook instructions provided in the prompts.

The failed cases of *Adding Invariant*, *Adding Action Instead of Guard*, and *Adding Weak Guard Predicate* patterns (Section IV-C) indicate the room for improvement to our LLM-based repair approach.

While allowing the addition and removal of invariants can be valuable, as in some cases invariants can be too strict or a new invariant requires additional assumptions, the LLM tends to over-suggest repairs with invariants, which leads to problems such as failing POs in other events.

Another problem is the suggestion of weak guards. While we don't want guards that are too strong and make the event impossible to occur (e.g., grd1: $\perp$), guards that are too weak do not help discharge the failing proof obligations.

Lastly, the *Specific Cases for Specific Event-B Problems* case highlights the limitations of the current version of the System Prompt. In our future work, we will attempt to address these issues by introducing new constraints to the System Prompt, such as those related to capitalization.

In conclusion, while the use of the System Prompt is effective, LLMs have a nondeterministic nature and sometimes fail to follow some of the instructions provided in the System Prompt. Additionally, there is still room for improvement in the System Prompt, particularly for unusual cases, such as issues related to capitalization.

B. Exceptional Cases

Tables II and III show that using a complete System Prompt (i.e., {C1, C2, C3} or {C1, C2, C3, R}), we get the same or an increased number of Solved cases compared to when using any incomplete System Prompt. However, some mutants are not classified as Solved using a *complete* System Prompt and are classified as Solved using one of the *incomplete* System Prompts.

For example, when we carried out the experiment *without* using Retry Prompt (Table II), we observed the following case:

- Guard removed to create the mutant:
  `"grd5:ml_out_10 = FALSE"`
- Reply to *complete* System Prompt ({C1, C2, C3})
  ```
  {"add":["act4:ml_out_10 := FALSE"],
  "remove":[]}
  ```
- Reply to *incomplete* System Prompt ({C1, C3})
  ```
  {"add":["grd6:ml_out_10 = FALSE"],
  "remove":[]}
  ```

Similar cases were observed in the experiment *using* Retry Prompt. For instance:

- Guard removed to create the mutant:
  `"grd3:s+1 = n"`
- Reply to *complete* System Prompt ({C1, C2, C3, R}):
  ```
  {"add":["inv4_4:(s_st = working
  ∧ w = FALSE) ⇒ r_st = success"],
  "remove":[]}
  ```
- Reply to *incomplete* System Prompt {C1, C3, R}
  ```
  {"add":["grd4:r_st = success"],
  "remove":[]}
  ```

In these cases, for mutants created by removing a guard, LLM suggested adding an invariant or action to repair when we used *complete* System Prompt. In contrast, it suggested adding a guard when we used *incomplete* System Prompt. This is counterintuitive because, as we discussed in Section V-A1, complete System Prompts obtained a better result in general. This is likely due to the uncertain nature of LLM, indicating that we should be aware of such cases.

C. Threats to Validity

Limitation of error scope. Our repair approach focused only on the lack of guards or actions in faulty models. For a thorough evaluation, other types of faults should also be taken into consideration.

Internal threats. We reduced our reliance on human judgment by automating all steps. However, the nondeterministic behavior of LLMs remains an internal threat.

External threats. The models used to create mutants (Table I) are from the textbook of Event-B [2], and thus carefully designed by the creator of the Event-B method. While we used multiple models to examine the generality, our method would benefit from testing on more realistic faulty models.

D. Future Perspective

As the threats to validity indicate, our future work should expand the scope of faulty models to be repaired by supporting other types of faults and evaluating the method with more models.

From the patterns observed from RQ3 (Section IV-C), we can see that one of the main issues of the current model is not always being able to identify when there is a need for an action, a guard, or an invariant. This type of problem can be fixed by using three different User Prompts for the LLM:

- "try to discharge these POs by suggesting addition/removal of *guards*",
- "try to discharge these POs by suggesting addition/removal of *actions*",
- "try to discharge these POs by suggesting addition/removal of *invariants*".

This method would provide us with three potential repairs for the problem, from which we could select the best one using a heuristic method.

Not always knowing which variable should be handled in a guard, action, or invariant is another problem observed in RQ3. LLMs may find it challenging to determine which variable to use in the proposed repair because the information presented only includes the names of the not-discharged POs and not the detailed state of the proof. Adding more feedback in the User Prompt, either from data collected from the prover of Rodin (such as the proof's current state) or from another formal methods tool (such as a counter-example discovered by a model checker), would be one way to address this issue.

Another approach that may be explored is combining the use of Generative AI with formal repair methods (e.g., [11]). Formal repair methods, which guarantee the correctness of the repair but can be computationally expensive, can be used complementarily to our LLM-based repair method. This approach can be effective in dealing with the nondeterminism of LLMs. Another possible benefit is reducing redundant suggestions. Repairs provided by the LLM can sometimes contain redundant clauses, and the use of formal methods can help reduce the set of clauses to add or remove to the smallest necessary subset.

Another issue that could be improved is the generation of repairs that discharge POs, but are not desirable from a requirements viewpoint. For instance, to repair a mutant created by removing the action inc_n of enter event of the cars example (Fig. 1), LLM may suggest using the following before-after predicate: $\mathsf{n}' \in \mathbb{N} \land \mathsf{n}' \leq \mathsf{n\_capacity}$. Intuitively, this predicate means that the value of variable n can change arbitrarily, as long as the after state (n') satisfies the invariant. Therefore, the predicate does not convey much information, and thus is not desirable to be included in a formal model. A possible solution for this type of issue is to incorporate the developer's intention into the User Prompt given to the LLM.

References

[1] Y. Zhang, Y. Cai, X. Zuo, X. Luan, K. Wang, Z. Hou, Y. Zhang, Z. Wei, M. Sun, J. Sun, J. Sun, and J. S. Dong, "The fusion of large language models and formal methods for trustworthy AI agents: A roadmap," 2024. [Online]. Available: https://arxiv.org/abs/2412.06512

[2] J.-R. Abrial, *Modeling in Event-B: System and software engineering*. Cambridge University Press, 2010.

[3] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 6, pp. 447–466, Nov 2010.

[4] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Trans. Softw. Eng. Methodol.*, Jul. 2025. [Online]. Available: https://doi.org/10.1145/3747588

[5] Y. Wu, A. Q. Jiang, W. Li, M. Rabe, C. Staats, M. Jamnik, and C. Szegedy, "Autoformalization with large language models," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 32 353–32 368. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/d0c6bc641a56bebee9d985b937307367-Paper-Conference.pdf

[6] D. Jiang, M. Fonseca, and S. B. Cohen, "LeanReasoner: Boosting complex logical reasoning with Lean," 2024. [Online]. Available: https://arxiv.org/abs/2403.13312

[7] S. Han, H. Schoelkopf, Y. Zhao, Z. Qi, M. Riddell, W. Zhou, J. Coady, D. Peng, Y. Qiao, L. Benson, L. Sun, A. Wardle-Solano, H. Szabo, E. Zubova, M. Burtell, J. Fan, Y. Liu, B. Wong, M. Sailor, A. Ni, L. Nan, J. Kasai, T. Yu, R. Zhang, A. R. Fabbri, W. Kryscinski, S. Yavuz, Y. Liu, X. V. Lin, S. Joty, Y. Zhou, C. Xiong, R. Ying, A. Cohan, and D. Radev, "Folio: Natural language reasoning with first-order logic," 2024. [Online]. Available: https://arxiv.org/abs/2209.00840

[8] Thai Son Hoang, "How to interpret failed proofs in Event-B," ETH Zürich, Tech. Rep., 2010, https://doi.org/10.3929/ethz-a-006857374.

[9] J. Schmidt, S. Krings, and M. Leuschel, "Repair and generation of formal models using synthesis," in *Integrated Formal Methods (iFM 2018)*, C. A. Furia and K. Winter, Eds. Cham: Springer International Publishing, 2018, pp. 346–366.

[10] C.-H. Cai, J. Sun, G. Dobbie, Z. Hóu, H. Bride, J. S. Dong, and S. U.-J. Lee, "Fast automated abstract machine repair using simultaneous modifications and refactoring," *Form. Asp. Comput.*, vol. 34, no. 2, Sep 2022.

[11] T. Kobayashi and F. Ishikawa, "Repairing Event-B models through quantifier elimination," in *Formal Methods and Software Engineering: 25th International Conference on Formal Engineering Methods, ICFEM 2024, Hiroshima, Japan, December 2–6, 2024, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 18–36. [Online]. Available: https://doi.org/10.1007/978-981-96-0617-7_2