Event-B Agent: Towards LLM Agent for Formal Model Synthesis and Repair

HONGSHU WANG*, National University of Singapore, Singapore
XINYUE ZUO, National University of Singapore, Singapore
YUHAN SUN, East China Normal University, China
QIN LI*, East China Normal University, China
YAMINE AIT AMEUR, IRIT - National Polytechnic Institute of Toulouse, France
JIN SONG DONG, National University of Singapore, Singapore

Building software that is correct by construction is a long-standing goal in software engineering, as it ensures that reliability is achieved during design rather than after deployment. Formal methods realize this vision by allowing system behavior and requirements to be expressed in mathematics, enabling correctness to be guaranteed through proofs and verification. However, the steep learning curve and demand for mathematical expertise hinder its widespread adoption. Large language models (LLMs) have recently shown promise in bridging this gap through autoformalization, but existing approaches often address isolated tasks, such as theorem proving or model synthesis verified by model checking. While valuable, these single-perspective efforts do not fully exploit the potential of a more comprehensive framework where models and proofs evolve together. To address this gap, we propose **Event-B Agent**, a novel framework inspired by the interleaved nature of software design. From natural language requirements, Event-B Agent constructs an initial model and iteratively adjusts it using feedback from model checking and theorem proving. Adjusting and refining the model, in turn, simplifies proof discharge that ensures correctness with respect to requirements. Evaluation across systems of varying complexity shows that Event-B Agent substantially outperforms baselines in end-to-end formal model synthesis and repair. These results show a promising direction for achieving correctness-by-construction through probabilistic approaches like LLM agents.

CCS Concepts: • **Software and its engineering → Formal software verification**; *Software design engineering*.

Additional Key Words and Phrases: Autoformalization, Formal Verification, LLM Agent

ACM Reference Format:
Hongshu Wang, Xinyue Zuo, Yuhan Sun, Qin Li, Yamine Ait Ameur, and Jin Song Dong. 2026. Event-B Agent: Towards LLM Agent for Formal Model Synthesis and Repair. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE211 (July 2026), 23 pages. https://doi.org/10.1145/3808218

*Co-corresponding authors.

Authors' Contact Information: Hongshu Wang, National University of Singapore, Singapore, hongshu.wang@u.nus.edu; Xinyue Zuo, National University of Singapore, Singapore, zuoxy@nus.edu.sg; Yuhan Sun, East China Normal University, China, 51285902023@stu.ecnu.edu.cn; Qin Li, East China Normal University, China, qli@sei.ecnu.edu.cn; Yamine Ait Ameur, IRIT - National Polytechnic Institute of Toulouse, France, yamine.aitameur@toulouse-inp.fr; Jin Song Dong, National University of Singapore, Singapore, dcsdjs@nus.edu.sg.

ACM 2994-970X/2026/7-ARTFSE211
https://doi.org/10.1145/3808218

1 Introduction

An important branch of formal methods is *correct-by-construction* model development for software systems and real-world applications. In contrast to the generate-and-test approaches common in software engineering, these methods aim to guarantee correctness through formal verification approaches such as model checking and theorem proving. Typically, requirements are formalized into a model incrementally, and each intermediate model must be proven correct before refinement proceeds. However, the steep learning curve and reliance on mathematical expertise have hindered their widespread adoption. As system complexity increases, the number of requirements grows rapidly, leading to an explosion in proof obligations. Representative frameworks such as the B-Method [4] and Event-B [5] often generate hundreds of proof obligations even for moderately complex models. Traditionally, iteratively constructing the formal model, checking correctness through proof obligations, and repairing accordingly is performed manually, making the process time-consuming and dependent on human expertise.

Meanwhile, the rise of large language models (LLMs) [6] has attracted growing attention across research communities [26, 27, 49]. Their impressive natural language understanding capabilities open new opportunities to automate formalization while preserving correctness via formal verification. Recent research in formal methods has explored the potential of LLMs in a variety of autoformalization tasks, including linear temporal logic synthesis [16, 20] and program specification generation [45, 46]. However, these works do not tackle system-level modeling. By contrast, recent attempts at LLM-based formal model construction remain largely empirical and lack a systematic framework for the iterative procedure described above [12, 25]. Notably, the recently proposed PAT-Agent [50] is designed specifically for formal model construction, leveraging model checking.

However, model checking alone does not suffice to prove the correctness of generated models. While model checking is effective at reporting counterexamples within a given bound, the absence of counterexamples only guarantees correctness up to that bound, and the cost of exploration grows rapidly as the bounds increase. In contrast, theorem proving provides unbounded reasoning. Proof obligations can be generated to ensure well-definedness of expressions, preservation of invariants, feasibility of events under their guards, and termination of the model [4]. Consequently, correct-by-construction methods typically rely on a combination of model checking and theorem proving. Similar to autoformalization, there has been growing interest in LLM-powered automated theorem provers [13, 19, 31]. Yet most existing works focus on isolated tasks, such as theorem proving alone or model synthesis verified solely by model checking. In practice, formal development requires models and proofs to evolve together. When a theorem prover fails to discharge a proof obligation, the failure may stem either from inadequate modeling, or from the inherent difficulty of automated reasoning. In such cases, the model often needs to be revised before development can proceed. Since neither model correctness nor prover completeness can be assumed, practical development must allow revisions to both the formal model and its proof artifacts. To our knowledge, no existing automated approach formulates formal development as a joint state space over models and proof artifacts, enabling iterative coordination between model synthesis and proof-guided repair.

To address the above challenges, we propose **Event-B Agent**, a novel framework for end-to-end formal model construction powered by LLM. Given a requirement document in natural language, Event-B Agent constructs a formal model through three stages. (1) **Refinement Stratey Planning.** To allow incremental model design and reduce the complexity of both model synthesis and proof-guided repair, we adopt the LLM-proposed refinement strategy that distributes the requirements across multiple refinement steps. With proof obligations, refinement guarantees that properties proven in earlier steps are preserved in later ones, thereby reducing the difficulty of synthesis and theorem proving. (2) **Model Synthesis.** For each refinement step determined by the refinement

strategy, a formal model is synthesized according to the assigned requirements and iteratively repaired to eliminate compilation errors. This step is crucial in ensuring requirements coverage, thereby increasing the number of requirements that can be fulfilled at the current refinement step. (3) **Model & Proof Repair.** After synthesizing the model at a refinement step, it is verified and repaired as needed. The Model & Proof Repair component discharges the refinement proof obligations automatically generated by Rodin [2], an IDE that supports Event-B specification, model checking, and theorem proving. This ensures that properties established in earlier refinement levels are preserved. It also proves the correctness of the new properties introduced at the current level. The process is repeated until all proof obligations at the current refinement step are discharged, after which the synthesis-and-repair procedure continues for the next step. The whole procedure stops when all the requirements have been addressed.

We evaluate Event-B Agent on the consistency and correctness of the constructed models on a benchmark of 27 formal systems with an average of 182.41 proof obligations each. Since the existing works cannot address the model-proof co-evolution problem, we compare against the closest existing attempts with the same input and output setting, that is, the LLM-based autoformalization methods. Across all evaluated metrics, Event-B Agent consistently outperforms the baselines, achieving a proof obligation discharge rate (PDR) of 97.86%, and surpasses the baselines in requirement coverage (RC) and requirement fulfillment (RF) by 4.63% and 18.01%, respectively. The ablation studies further demonstrate the effectiveness of the two key components of our framework: Refinement Strategy Planning and Model & Proof Repair. Moreover, these components enhance the performance of each other. Refinement reduces the complexity of individual proofs, while the repair component establishes the correctness of refinements, ensuring that fulfilled requirements are preserved across refinement steps. Additionally, Event-B Agent completes synthesis and repair for each system in an average of 74.45 minutes, while discharging proof obligations takes an average of 0.24 minutes. In summary, this paper makes the following contributions:

(1) **Framework.** We introduce **Event-B Agent**, the first end-to-end framework for formal model synthesis and repair that supports refinement-based development and co-evolution of models and proof artifacts, embodying the principle of correctness by construction.
(2) **Tool.** We implement a proof-of-concept tool and integrate it into the Rodin IDE [2], demonstrating the practicality of our approach.
(3) **Evaluation.** We evaluate Event-B Agent on 27 formal systems against baselines, with ablations to quantify component contributions, along with efficiency and qualitative analyses. All code, datasets, and the interface are publicly available[1].

2 Motivation

2.1 Motivating Example

Table 1 lists the requirements of a formal system specifying an algorithm that searches for the minimum value of $f(j)$ over all valid j in the domain of function f. In this example, the requirements are divided into two categories: EQP (equipment) and FUN (function). The equipment requirements define the constants, sets, and axioms available to the system, while the function requirements specify its intended behavior. In this example, $EQP{-}1$ and $EQP{-}2$ introduce the constants n and f. $FUN{-}1$ states that upon termination, the algorithm must return an index $j \in dom(f)$ such that $f(j)$ is the minimum value in the range of f. Requirements $FUN{-}2$ through $FUN{-}6$ describe the iterative searching process used to achieve this goal.

As discussed in Section 1, we select a set of the existing approaches that could potentially address the problem of formal model construction, namely GPT-5 with medium reasoning, Cursor, and

[1] https://anonymous.4open.science/r/EventB_Agent-939D

ID	Description
EQP-1	Provides a finite natural number function f defined on the domain $\{0..n-1\}$.
EQP-2	Provides n as the size of the domain of f.
FUN-1	The final condition requires $j \in \text{dom}(f)$ and $f(j) = \min(\text{ran}(f))$.
FUN-2	The system has a boolean variable `searching` to indicate whether the minimum has been fully identified. Two indices are maintained: i as the scanning index and j as the current candidate for the minimum.
FUN-3	Initialization sets i = 1 and j = 0, with searching = true.
FUN-4	While `searching` is true, if $i \neq n$ and $f(i) < f(j)$, update $j := i$ and increment i.
FUN-5	While `searching` is true, if $i \neq n$ and $f(i) \geq f(j)$, keep j unchanged and increment i.
FUN-6	When $i = n$, `searching` becomes false, marking the completion of searching.

Table 1. Equipment (EQP) and Function (FUN) requirements of minimum-searching model.

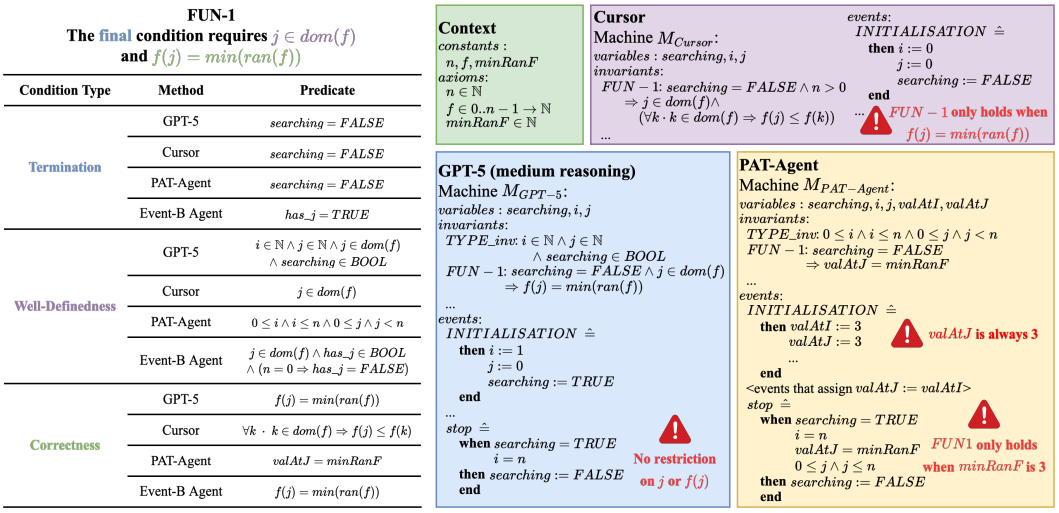

Fig. 1. Simplified Event-B models for the motivating example generated by GPT-5, Cursor, and PAT-Agent.

PAT-Agent, and their performances are illustrated in Figure 1, while the models generated by Event-B Agent are presented in Figure 2. The "Context" block in Figure 1 represents the shared context across all four models, corresponding to the EQP requirements. The table on the left of the figure shows how each method formalizes requirement $FUN{-}1$ to achieve algorithm termination, well-definedness, and correctness. In model M_{Cursor} generated by Cursor, the $INITIALISATION$ event sets $j := 0$ and $searching := FALSE$, violating $FUN{-}1$ unless $f(0) = min(ran(f))$. In contrast, the rest of the methods do not impose restrictions on termination during initialization. GPT-5 produces model M_{GPT-5} with $FUN{-}1$: $searching = FALSE \ \land \ j \in dom(f) \Rightarrow f(j) = min(ran(f))$, where the termination condition is not required to hold in the initial state. However, in the $stop$ event, the guards impose no restriction on j or $f(j)$, allowing the algorithm to terminate with an arbitrary j, violating $FUN{-}1$. PAT-Agent, a framework specialized for formal model synthesis, produces a stronger model $M_{PAT-Agent}$ in which the guards of $stop$ enforce $valAtJ = minRanF$, thereby satisfying $FUN{-}1$. Nonetheless, this model contains subtle flaws: in $INITIALISATION$, both $valAtI$ and $valAtJ$ are set to 3, and subsequent events only update $valAtJ := valAtI$ without modifying $valAtI$. As a result, the guard of $stop$ implies that the constant $minRanF$ must always be

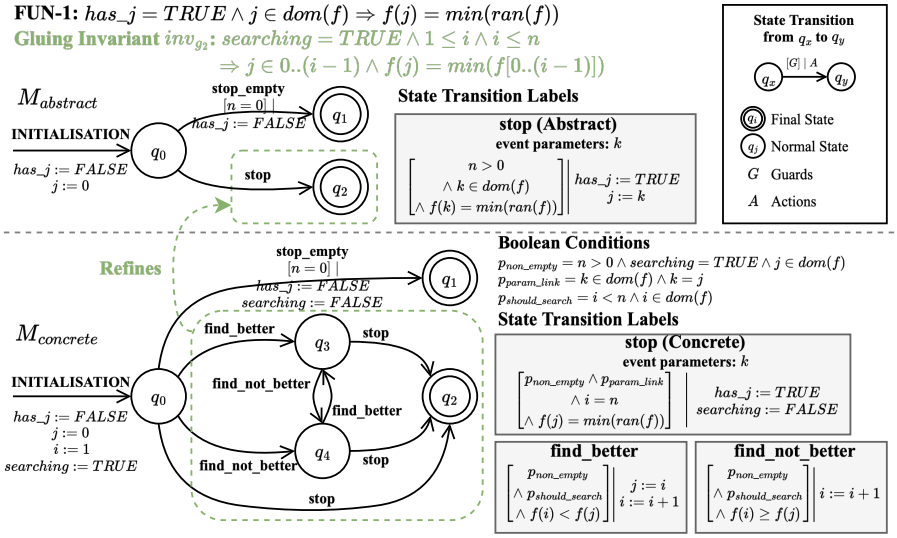

Fig. 2. The abstract and concrete formal models constructed by Event-B Agent for the motivating example.

3. From a model checking perspective, this system is deemed correct, but the correctness does not generalize beyond the single case $minRanF = 3$.

The models generated by Event-B Agent, $M_{abstract}$ and $M_{concrete}$, resolve this issue through refinement. The models are shown in Figure 2, where events are represented by state transitions, and $[G]|A$ is an event with guards G and actions A. Since $FUN{-}1$ depends only on j and f, the abstract model omits auxiliary variables such as i and $searching$, and focuses solely on proving $FUN{-}1$. $FUN{-}1$ is vacuously true for $INITIALISATION$ and $stop\_empty$ events. For the $stop$ event, $FUN{-}1$ is preserved because the event's assignments ensure $has\_j = TRUE \wedge j \in dom(f)$ in the post-state, and the guard enforces $f(j) = min(ran(f))$. By construction of the refinement, the refined events $INITIALISATION$, $stop$, and $stop\_empty$, strengthen the guards of their abstract counterparts and simulate the abstract assignments, thereby preserving the invariant in M_{concrete}.

This example highlights three key insights motivating the design of Event-B Agent. First, refinement reduces modeling and proving complexity by abstracting away irrelevant details. Second, when models are inaccurate, a systematic repair framework is required, e.g., updating the actions of $INITIALISATION$ in M_{Cursor} and the guards of $stop$ in M_{GPT-5}. Finally, model checking alone is insufficient to guarantee correctness, and proof information must be exploited in model synthesis and repair. Accordingly, we propose Event-B Agent, which addresses the formal model synthesis and repair problem by incorporating refinement, a novel repair component, and proof information.

2.2 Problem Statement

Let M denote the set of formulas extracted from a formal model, π denote a proof artifact witnessing that a proof obligation (PO) φ is derivable from M under a sound proof system $\mathcal{R}$. Then M is said to discharge φ if and only if there exists π that witnesses $M \vdash_{\mathcal{R}} \varphi$. We abstract formal development as a co-evolution process over models and proof artifacts. Formally,

$$(M^0, \pi_0) \rightsquigarrow (M^1, \pi_1) \rightsquigarrow \ldots \rightsquigarrow (M^n, \pi_n) \text{ s.t. } \forall\, t \in \{0, \ldots, n\}, \pi_t \text{ is a valid proof for } M^t \vdash_{\mathcal{R}} \varphi_t,$$

where $M^0 = \varnothing$, $\pi_0 = \epsilon$, n is the number of POs, and the step $(M^0, \pi_0) \rightsquigarrow (M^1, \pi_1)$ synthesizes the initial model M^1 given natural language requirements and discharges π_1. Each pair (M^t, π_t)

i ::= *literal*;	Identifier	
il ::= $\{i\}$;	List of identifiers	
$pred_{il}$ ::= *predicate*;	Predicate such that $FV(pred_{il}) \subseteq il$	
M ::= *Context*, *Machine*;	Formal model	
Context ::= i,	Context identifier	
["extends", i],	Extended context	
"sets", s,	$s ::= il$	
"constants", c,	$c ::= il$	
"axioms", A,	$A ::= \{pred_{s \cup c}\}$	
"theorems", T;	$T ::= \{pred_{s \cup c}\}$	

Machine ::= i,	Machine identifier	
["refines", i],	Refined machine	
["sees", i],	Seen context	
"variables", v,	$v ::= il$	
"invariants", I,	$I ::= \{pred_{s \cup c \cup v}\}$	
"variants", N,	$N ::= \{pred_{s \cup c \cup v}\}$	
"theorems", T,	$T ::= \{pred_{s \cup c \cup v}\}$	
"events", $\mathcal{E}$;	$\mathcal{E} ::= \{Event\}$	
Event ::= i,	Event identifier	
"any", x,	Event parameters, $x ::= il$	
"where", G,	Event guards, $G ::= \{pred_{s \cup c \cup v \cup x}\}$	
"then", v : | BA;	Actions, $BA ::= \{pred_{s \cup c \cup v \cup x \cup v'}\}$	

Fig. 3. The grammar of Event-B, where $FV(pred)$ is the set of free variables in $pred$.

represents an intermediate model M^t with a proof π_t discharged at step t, allowing updates to both M^t and π_t, while ensuring soundness at each step through π_t under the sound proof system $\mathcal{R}$.

We position our work relative to two lines of research. LLM proof assistants assume a fixed model M and search for valid π, but cannot construct $M^0 \rightsquigarrow M^1$ from natural language requirements or revise M^t during development. In contrast, LLM-based autoformalization methods iteratively construct models $M^0 \rightsquigarrow \cdots \rightsquigarrow M^k$ without generating proof artifacts or using proof-guided repair, validating correctness only through bounded model checking. Neither line of work formulates formal development as joint model–proof co-evolution.

Furthermore, refinement should be supported to incrementally develop the system and reduce proof effort. Let M_i^t denote the t-th model at refinement level i, where $i = \{1, \ldots, m\}$. For each $i < m$, the last model at level i is refined by the first model at level $i+1$, with refinement soundness ensured by POs. Eventually, M_m^n is produced as the final model. Existing LLM-based autoformalization does not support refinement in this sense, and refinement is not applicable to LLM proof assistants.

3 Preliminary

The Event-B Notation. In Figure 3, we summarize the grammar of Event-B [5]. An Event-B model M is a discrete transition system grounded in set theory and first-order logic. All types in Event-B are represented as sets. For example, set *STRING* may be declared as the set of all strings, while $A \rightarrow B$ denotes the set of total functions with domain A and range B.

A formal model M consists of *contexts* and *machines*. Let il denote a list of identifiers, then sets s, constants c, variables before and after an event v and v', and event parameters x can each be defined by il, and these sets are disjoined. A predicate $pred$ with $FV(pred) = il$ is denoted by $pred_{il}$, where $FV(pred)$ is the set of free variables in $pred$. A list of predicates using only variables in il is denoted by $\{pred_{il}\}$, which can be used to express components like axioms, $A ::= \{pred_{s \cup c}\}$.

Correctness by Construction. In this paper, a formal model is said to be *correct by construction* if its correctness is guaranteed during development rather than via post-hoc testing. Let REQ_M denote the requirements of formal model M. Each $req \in REQ_M$ is associated with a predicate $Covered(M, req) \in \{\text{true}, \text{false}\}$, which holds if and only if req is covered in M as an invariant, variant, axiom, or event. Similarly, each proof obligation $po \in PO_M$ has a predicate $Discharged(M, po) \in \{\text{true}, \text{false}\}$, which holds if and only if po is successfully proved. We define a predicate $Correct(M)$ to indicate that M is *correct by construction*, namely:

$$Correct(M) \triangleq \Big(\bigwedge_{req \in REQ_M} Covered(M, req) \Big) \wedge \Big(\bigwedge_{po \in PO_M} Discharged(M, po) \Big) \tag{1}$$

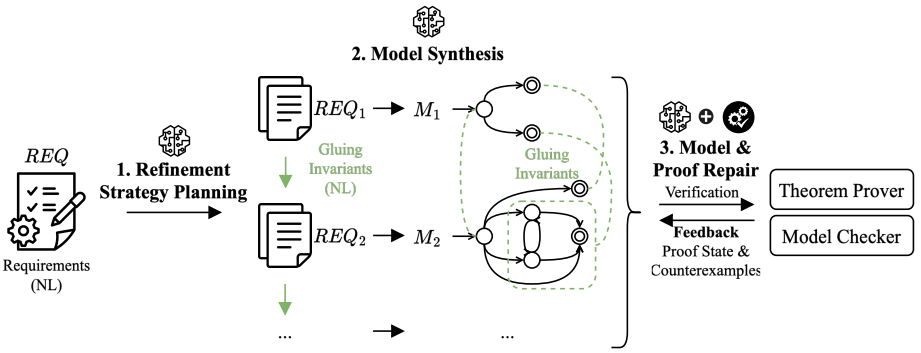

Fig. 4. Overview of Event-B Agent. From natural-language requirements (*REQ*), the system plans a refinement strategy to guide incremental model synthesis, with verification and repair at each refinement level.

4 Methodology

4.1 Overview

In this work, we propose Event-B Agent, an LLM-powered end-to-end framework for formal model synthesis and repair. Figure 4 shows the overall workflow of Event-B Agent, the Model & Proof Repair component is elaborated in Figure 5, and Figure 2 illustrates the minimum-searching model generated by Event-B Agent. System description in natural language is provided to Event-B Agent to generate a refinement strategy, which distributes requirements to various refinement steps. In this process, gluing invariants between two refinement steps are summarized in natural language. During model synthesis, the distributed requirements and gluing invariants are formalized into a model. The correctness of the model is then verified through model checking and theorem proving, and repairs are applied accordingly. This procedure is repeated for each refinement step until all proof obligations are discharged or the trial limit is reached.

Event-B Agent adopts a neurosymbolic design, where semantic tasks such as refinement planning, model synthesis, and repair are delegated to specialized LLMs, and deterministic components handle the rest. The model checker and theorem prover perform formal verification over synthesized models, pattern matching identifies candidate repair rules based on the proof state, and atomic fixing functions execute concrete repairs. This functional separation combines the flexibility of LLMs with the reliability of symbolic reasoning, achieving both semantic versatility and soundness.

4.2 Refinement Strategy Planning

Automated reasoning over realistic software systems is challenging due to the rapid growth of the search space as model complexity increases. Refinement offers a complementary approach by establishing properties in simpler abstract models, where proofs are easier to discharge due to abstraction, and preserving them in more detailed concrete models through refinement proof obligations. This ensures that proof effort scales with model complexity while preserving correctness.

On this basis, Event-B Agent utilizes LLMs to automatically plan refinement strategies from natural language requirements. Refinement introduces additional detail in a disciplined manner, guided by gluing invariants that relate abstract and concrete models to maintain consistency.

4.2.1 Refinement. Having introduced the role of refinement, we now formalize this notion. Let M_A denote an abstract model and M_C a concrete refinement of M_A. Let REQ_M denote the set of requirements encoded in model M. Refinement preserves requirements of the abstract model REQ_{M_A} while introducing additional ones REQ_{diff}, i.e., $REQ_{M_A} \subset REQ_{M_C} \wedge REQ_{diff} \triangleq REQ_{M_C} \setminus REQ_{M_A}$.

The preservation of proven properties is guaranteed through gluing invariants and the corresponding refinement POs. For instance, the refined events are required to have stronger guards than the abstract events, and their actions must simulate the abstract counterparts. With these guarding POs, Event-B Agent is able to ensure refinement correctness, as illustrated in Figure 2.

4.2.2 Gluing Invariant. A gluing invariant is a predicate over both abstract variables and concrete variables, specifying how states in the concrete model correspond to states in the abstract model. Figure 2 includes a gluing invariant between $M_{abstract}$ and $M_{concrete}$ generated by Event-B Agent. $M_{concrete}$ introduces a variable i to represent the current index during search, and a boolean variable $searching$ to represent whether the searching procedure is ongoing. The gluing invariant relates the abstract variable j to the two concrete variables i and $searching$, and specifies that during searching, $f(j)$ is always the minimum value among the explored domain $0..(i-1)$. The Model Synthesis step later formalizes it into $searching = TRUE \wedge 1 \leq i \wedge i \leq n \Rightarrow j \in 0..(i-1) \wedge f(j) = min(f[0..(i-1)])$.

Gluing invariants are added to the model M and play a critical role in discharging refinement POs. In Event-B Agent, candidate gluing invariants are first proposed by the refinement LLM and then formalized during model synthesis. Since LLM-based generation offers no correctness guarantee, gluing invariants are validated in two steps: (1) counterexample checking and contradiction detection with the model checker, and (2) attempting proofs relevant to the gluing invariants before other proofs. Only after no counterexamples are found and the proofs succeed, the invariants will be accepted into M for subsequent reasoning.

4.2.3 Refinement Strategy. Refinement proceeds inductively, distributes requirements across successive steps, and preserves established properties. The first stage of Event-B Agent is to plan a refinement strategy that decides how requirements are allocated across steps and how successive models are linked by gluing invariants. Once the strategy is set, the model synthesis and repair components construct the models, verify the requirements, and apply corrections if needed.

Concretely, the refinement strategy planning step of Event-B Agent partitions REQ into n disjoint subsets REQ_{M_i} such that $REQ = \bigcup_{i=1}^{n} REQ_{M_i}$. REQ_{M_1} includes the requirements for the initial abstract model, REQ_{M_2} contains the additional requirements for the second model, etc. Between steps $i-1$ and i, the refinement LLM also proposes a set of gluing invariants $\mathcal{I}_{g_i}$, which are formalized and added into M_i during model synthesis. Thus, the additional requirements for step i are $REQ_{M_i} \cup \mathcal{I}_{g_i}$, where REQ_{M_i} specifies system requirements and $\mathcal{I}_{g_i}$ aims to preserve all previously satisfied requirements $\bigcup_{j=1}^{i-1} REQ_{M_j} \cup \mathcal{I}_{g_j}$, where $\mathcal{I}_{g_1} = \varnothing$.

In the minimum-searching example, Event-B Agent creates a refinement strategy of 2 steps, $REQ_{M_1} = \{FUN\text{-}1\}$ and $REQ_{M_2} = \{FUN\text{-}2, FUN\text{-}3, FUN\text{-}4, FUN\text{-}5, FUN\text{-}6\}$. Additionally, as shown in Figure 2, a set of gluing invariants $\mathcal{I}_{g_2} = \{inv_{g_2}\}$ is generated to capture the relation between refinement steps 1 and 2. At this stage, the requirements in REQ_{M_1}, REQ_{M_2}, and $\mathcal{I}_{g_2}$ are in natural language form and will be formalized during the Model Synthesis step.

Formally, the correctness of a refined model M_i with respect to all requirements up to step i is defined inductively as:

$$Correct(M_i) \triangleq \Big(\bigwedge_{j=1}^{i-1} Correct(M_j)\Big) \wedge \Big(\bigwedge_{req \in REQ_{M_i} \cup \mathcal{I}_{g_i}} Covered(M_i, req)\Big) \wedge \Big(\bigwedge_{po \in PO_{M_i}} Discharged(M_i, po)\Big) \quad (2)$$

This planning stage provides the blueprint for synthesis and repair: each step has well-defined requirements and invariants, while correctness is ensured and preserved through proof obligations.

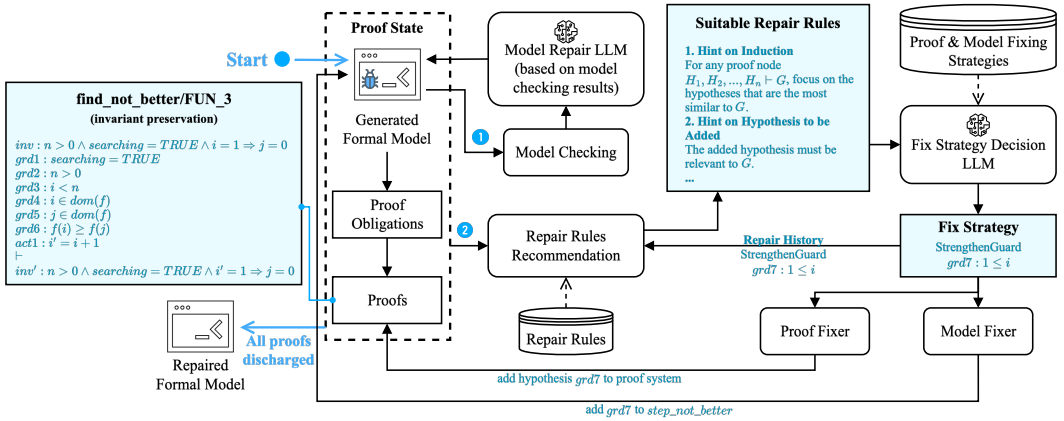

Fig. 5. Workflow of the Model & Proof Repair component. The model is ① checked by a model checker (bounded) and repaired, then ② repaired guided by unbounded theorem proving.

4.3 Model Synthesis

For each refinement step i with requirements $REQ_{M_i} \cup \mathcal{I}_{g_i}$, Event-B Agent synthesizes a formal model that captures the specified requirements. The model must be syntactically correct and free of compilation errors so that formal verification tools can be applied effectively.

Schema-Guided Formalization. Simple prompt-based generation frequently produces ill-formed code, including undeclared variables, malformed invariants, and syntactic errors that are not parsable, particularly in the synthesis of formal specifications where training data is sparse. To address this, we design a JSON schema that encodes the grammar of Event-B shown in Figure 3. The schema encodes structural constraints, for example, every machine must contain at least one event, and may include variables, invariants, and variants. Because the schema is language-agnostic, it can be readily adapted to other formal specification languages, making the approach generalizable.

Synthesis and Repair Loop for Well-Formedness. The model synthesis LLM first generates a candidate model according to the JSON schema. The model is parsed and compiled into Event-B code. The schema eliminates most of the syntax errors but cannot enforce typing constraints, so common issues including undeclared variables, type mismatches, and invariants referencing undefined sets still exist. If compilation errors arise, the error messages together with the current model are fed back to the LLM for iterative repair. Through this synthesis and repair loop, every model passed to verification tools is both syntactically valid and type-checked, providing a reliable foundation for subsequent refinement and model repair.

Ensuring Refinement Soundness. While synthesizing a refined model, the requirements at the current refinement level and the previous abstract model are provided to the LLM as context. The refinement relation is preserved through refinement POs after generating the refined model. In our experiments, the refinement PO discharge rate is collected to evaluate the refinement correctness.

4.4 Model & Proof Repair

One of the key contributions of our work lies in the Model & Proof Repair component, which iteratively verifies the models, reasons about the cause of inaccuracies, and repairs the models and proofs. An overview of this component is shown in Figure 5.

4.4.1 Formal Verification and Repair. Whenever a new model is synthesized, it is first verified by the model checker to detect violations of invariants, including both requirement and gluing

invariants (Step ① in Figure 5). The model repair LLM suggests suitable bounds, within which the checker searches for states that violate invariants $\mathcal{I}$. If a counterexample trace is found, it indicates an invariant violation. This trace is then passed to the model repair LLM as feedback for correcting the model to resolve the violation within the given bounds.

While model checking is effective for detecting invariant violations, it cannot guarantee full correctness. This is because bounded analysis may miss unreachable states or behaviors outside the explored bounds. To cover all executions, theorem proving is applied in addition. After proof obligations are generated, automated provers and constraint solvers attempt to discharge them under the relevant hypotheses extracted from the model. When proofs fail, the cause may range from modeling inaccuracies or limitations of the prover itself. To address this, the Model & Proof Repair component analyzes the proof state and applies targeted fix strategies to either the model or its proofs (Step ② in Figure 5).

In our method, repairs targeting discharging POs are generated through the repair rules recommendation component and the fix strategy decision LLM. Based on pattern matching, the former retrieves repair rules to construct the prompt for the latter. A sequence of repairs is accepted only when the combined effect of all applied repairs discharges the target PO under formal verification. Once a PO is discharged, the theorem provers guarantee its validity. After every model modification, all proofs are replayed so that any proof invalidated by later changes is not considered as successful. Therefore, soundness is inherited from the verification pipeline and does not rely on the repair rules. The repair rules offer heuristic guidance to LLM by suggesting common proof actions, and our ablation study shows clear empirical gains. Importantly, the rules do not alter the formal modeling or proof mechanisms and therefore do not affect soundness.

4.4.2 Proof Rules Recommendation. To improve the reasoning capability of the Agent, we design a novel repair rules recommendation component, which recommends suitable rules for repairing the given model M and a proof π about the model, and provides them as guidance to the fix strategy decision LLM. We observe that characteristic patterns emerge across model M, proof π, the proof obligation type, and their corresponding historical repairs $R(M, \pi)$.

To illustrate how repair rules are recommended, consider the case of universally quantified hypotheses. In interactive theorem proving, such hypotheses become useful only after instantiation with concrete terms. Although SMT solvers perform heuristic instantiation, this process is incomplete and often fails in Event-B proofs. Thus, after adding a universal predicate as a lemma in π, the next step is typically to instantiate it. If the repair history $R(M, \pi)$ indicates that a universal lemma has been added but no instantiations appear, and the child nodes in the proof tree remain open, the following rule is recommended: *"If a universal lemma has been added to a node in the proof tree, but its child nodes remain unclosed, suggest instantiation values for the lemma"*. In this example, the first action modifies M by adding the universal lemma as a reusable axiom in the context, while the second action instantiates it in π if SMT solvers do not already discharge the proof.

Beyond this example, our system supports a total of seven categories of rules, summarized in Table 2. The "Conditions" column specifies how each category depends on the current proof state, which includes information from the model M, the proof π, the repair history $R(M, \pi)$, and the type of proof obligation. Within each category, multiple concrete repairs may apply, and these sometimes cannot be determined by pattern matching. Instead, the fix strategy decision LLM analyzes the scenario, considers the recommended rules, and may attempt additional fixes if necessary. The "Rules" column lists a non-exhaustive set of simplified rules under various scenarios. These categories and rules were derived empirically based on our experience in manually discharging proofs. While not exhaustive, they capture recurring patterns observed across a wide range of models. If the current proof state does not fall into one of the known categories, the LLM

	Category	Condition	Rules
1	Contradictory Goal	Goal is $\bot$	1. PO is about an invariant → update the invariant to eliminate contradictions 2. PO is about an event → add or update actions or guards to eliminate contradictions
2	True by Definition	Goal matches definitional pattern, e.g. $x = y, x \in S$ etc.	1. Goal holds by definition → apply the corresponding tactics 2. (a) Rule 1 doesn't update $\mathcal{P}$, or (b) Goal is false → Similar rules as **Category 1**
3	Existential Goal	Goal is an $\exists$ predicate, e.g. $\exists x \cdot < predicate >$	1. x is a variable → add or update actions to define x 2. x is a constant → add hypothesis as an axiom in the context to establish its existence
4	Equality PO	PO type is Equality, e.g. goal G is $x = y$, where x, y are from machines at different refinement steps	1. If there is an indirect relation H (e.g. $f(x) = f(y)$), add axiom of the form $H \Rightarrow G$ if suitable.
5	Well-Definedness	PO type is Well-Definedness e.g. ensure $y \neq 0$ for x/y, ensure $x \in dom(f)$ for $f(x)$	1. PO is about guard → strengthen the guard based on the definition of well-definedness 2. PO is about invariant → strengthen the invariant based on the definition of well-definedness
6	Quantified Invariant Preservation	PO is Invariant Preservation about a quantified invariant	1. Universal invariant → Find quantified hypothesis $\forall x \cdot H(x) \Rightarrow G(x)$, where $G(x)$ is similar to proof goal $G(x')$. Instantiate the hypothesis with x'. 2. Existential invariant $\exists x \cdot P(x)$ → Similar rules as **Category 3**
7	Uninstantiated Hypothesis	Previous repairs add quantified hypotheses that are not yet instantiated	1. Quantified hypothesis added but not instantiated → suggest suitable instantiation values

Table 2. Categories of proof states and the corresponding repair rules supported by Event-B Agent.

falls back on a set of general default rules. Figure 5 shows an example of model-proof repair with the default rules. $FUN\text{–}3$ in Table 1 specifies a property about initial values of variables, formalized as $inv : \ n > 0 \wedge searching = TRUE \wedge i = 1 \Rightarrow j = 0$. The PO $find\_not\_better/FUN\_3$ fails because an event instance with pre-state $i = 0$ and post-state $i' = 1$ is admitted, yielding a post-state invariant inv' that requires $j = 0$, which cannot be derived from the proof context. However, such a state is unreachable in the actual execution of the algorithm, where $i \geq 1$ is maintained (cf. Figure 2).

As the proof state does not match any rule category in Table 2, the repair rules recommendation component falls back to default rules, whose simplified forms are shown as "Suitable Repair Rules" in Figure 5. These rules guide the later steps to focus on the inductive relationship between inv and inv' (Rule 1), and add relevant hypotheses to inv' when needed (Rule 2).

4.4.3 Fix Strategy Selection and Execution. The repair rules introduced in Section 4.4.2 guide the repair decisions of the fix strategy decision LLM. However, large language models may still hallucinate during repair. To mitigate this, Event-B Agent restricts modifications to a library of deterministic atomic functions (e.g., strengthen an invariant) that correspond to the fix strategies and update models and proofs. The functions are described as "atomic" because they represent the smallest modification units. After the recommended rules are obtained, the LLM selects an atomic function to execute, after which the proof state is updated. This iterative process continues until the proof is discharged or a fixed trial limit is reached.

The commonly used fix strategies are summarized in Figure 7, which are grouped into 4 categories:

- **Model modification strategies**: This group of strategies corresponds to atomic functions such as adding or updating invariants, strengthening guards, and modifying actions;
- **Proof modification strategies**: These strategies correspond to atomic functions, including instantiating quantified hypotheses and unfolding definitional equalities;
- **Joint model–proof modification strategies**: Operations that simultaneously modify the model and the selected proof, e.g., introducing a hypothesis as a context axiom while injecting it into the proof context, thereby repairing both in tandem;
- **Information Retrieval strategies**: operations such as invoking the model checker to obtain guidance for subsequent repair steps, without directly modifying the model or proof.

To summarize, the fix strategy decision LLM is responsible for two complementary tasks: (1) selecting the appropriate function based on the current proof state and recommended rules; and (2) generating the specific repair content when it cannot be inferred statically. For instance, in Figure 5, the selected function is $strengthenGuard$, and the repair content is adding $grd7 : 1 \leq i$, which is inferred from the inductive relationship between inv and inv'.

5 Evaluation

Event-B Agent is evaluated through the following research questions:

- **RQ1.** Is Event-B Agent effective in constructing consistent and correct models? How does it compare against existing approaches attempting this problem?
- **RQ2.** How does each component of Event-B Agent contribute to its overall performance?
- **RQ3.** How efficient is Event-B Agent in the task of formal model synthesis and repair?
- **RQ4**. How does Event-B Agent construct and repair models across different refinement steps, and what repair strategies does it employ?

5.1 Experimental Setting

Tool. We developed a proof-of-concept implementation of the Event-B Agent integrated into the Rodin IDE [2]. The structured JSON schema described in Section 4.3 enables the generated Event-B specification to be parsed and incorporated into the IDE, which natively supports structured Event-B code. In our experiments, model checking was performed by ProB [3], a model checker for Event-B that checks deadlock-freeness, liveliness, consistency of axioms, and invariants preservation.
Backbone LLM. We use GPT-5 (medium reasoning configuration, 2025-08-07 version) [37] as the backbone LLM in our experiments to demonstrate that formal model synthesis and repair remain challenging even for one of the most advanced LLMs.
Baselines. To the best of our knowledge, no prior work formulates formal development as a joint co-evolution of models and proof artifacts to achieve correctness-by-construction. For empirical evaluation, we therefore consider the closest existing agentic approaches that share the same input–output setting, even though they do not natively support proof-guided repair. For comparison, the outputs must support proof obligation generation and theorem proving. A common specification language with such support is therefore required. We chose Event-B because it provides unbounded proving and model checking, enabling the comparison and allowing the model checking based approaches to operate. Specifically, we evaluate against three representative baselines below:

(1) **LLM Model Synthesis with Automated Provers.** A straightforward baseline is to generate specifications with an LLM and discharge the resulting proof obligations using automated provers. Within Rodin, this includes built-in engines such as PP (Predicator Prover) and integrated SMT solvers (CVC4 [8], Z3 [17], etc.).
(2) **Adapted General Purpose Coding Agent (Cursor).** We evaluate Cursor [1], a commercial coding LLM agent that integrates features such as file editing, codebase search, and terminal

execution. In our setup, web search is disabled, while other features remain enabled. Cursor is guided by high-level task instructions and explicit commands to parse Event-B specifications (JSON) into a formal model and run the model checker.

(3) **Adapted PAT-Agent.** To evaluate existing formal agents in our problem setting, we adapt PAT-Agent [50], originally designed for autoformalization in PAT [43]. PAT and Event-B are both event/state-based specification languages, differing mainly in syntax and prover support. Our adaptation generates Event-B models by mapping context and machine constructs to PAT's constants, variables, guarded actions, and processes; rewriting syntax documentation and examples in Event-B; replacing the PAT model checker with ProB for verification feedback; and omitting proof obligations, as PAT-Agent focuses solely on model checking. This adapted version serves as a baseline to contrast with Event-B Agent's dual approach, integrating both model checking and theorem proving.

After model construction by Cursor and PAT-Agent, we apply the automated provers from baseline (1) to their outputs, ensuring all baselines are evaluated under comparable conditions.

Dataset. We collected a dataset with 27 formal systems, including classic examples and algorithms collected by Jean-Raymond Abrial (the designer of the B Method and Event-B) and widely regarded as representative Event-B developments [5], together with real-world systems [40, 41]. For the dataset on classic algorithms and real-world systems, we manually construct the requirement document based on the descriptions of the systems. We partition the dataset into three subsets, "Simple", "Medium", and "Complex", based on the number of requirements (3–8, 9–13, and 14–24, respectively), with nine systems in each partition. Although requirement count is not the only possible criterion, it is available prior to model construction and correlates with modeling complexity, as reflected by the increasing average number of POs in Event-B models generated by our method (89.22, 173.7, and 284.3). Other potential metrics (e.g., proof size or structural properties) are either unavailable before construction or do not reliably reflect modeling difficulty. Therefore, we adopt the requirement count as a deterministic and practical partition criterion.

Metrics. In this work, we design three metrics to evaluate the consistency and correctness of the constructed formal models.

Consistency. A formal model is consistent if it is not contradictory to itself. We evaluate the consistency level of a model through the Proof Obligation Discharge Rate (PDR). For a model M with a set of proof obligations PO_M, the PDR is defined as follows:

$$PDR = \frac{\sum_{po \in PO_M} \mathbb{I}[Discharged(M, po)]}{|PO_M|}$$

The PDR metric corresponds to the second conjunct in Equation 1, measuring the proportion of proof obligations that are successfully discharged. In our experiment, we report the PDR of the final model after refinement and repair, since the final model's consistency reflects the consistency of the entire refinement chain.

Correctness. A model M is correct if it covers and fulfills all requirements from the requirement document REQ_M. A requirement $req \in REQ_M$ is considered covered if it is presented in M without errors, and req is fulfilled if:

(1) It is covered in M, i.e. $Covered(M, req)$
(2) POs associated with req are discharged, i.e. $\bigwedge_{po \in PO_{req}} Discharged(M, po)$

During model construction, we instruct the corresponding LLM to generate labels for elements corresponding to the requirements they represent. In this way, we can compute both requirement coverage (RC) and requirement fulfillment (RF):

$$RC = \frac{\sum_{req \in REQ_M} \mathbb{I}[Covered(M, req)]}{|REQ_M|}, \quad RF = \frac{\sum_{req \in REQ_M} \mathbb{I}[Covered(M, req) \wedge (\bigwedge_{po \in PO_{req}} Discharged(M, po))]}{|REQ_M|}$$

RC and *RF* are computed across multiple refinement layers, since the refine layer hides the preserved requirements and only includes new requirements or updated requirements.

Validity of Assumption. When computing the metrics, we assume that once a requirement is covered and fulfilled in an abstract model M_i, it remains so in the subsequent refinement models $M_{i+1}, \ldots, M_n$. This assumption holds provided the refinement is correct, i.e., all refinement POs are discharged. In practice, a failed refinement PO does not necessarily imply that all requirements from M_i are violated, but it is also unclear which subset of requirements may be affected. To make *RC* and *RF* computable across refinement layers, we therefore adopt this assumption, and validate it empirically by reporting the *PDR* for refinement-related POs for all refinement layers.

5.2 RQ1. Overall Performance

We assess the effectiveness of Event-B Agent for end-to-end formal model synthesis and repair by comparing it with the three baseline methods introduced in Section 5.1. Table 3 summarizes the performance of all four methods across datasets partitioned by system complexity, as defined in Section 5.1, and reports their overall results. The methods under comparison are:

- **LLM + auto provers**: a naive baseline that synthesizes models using an LLM and attempts repair with existing automated theorem provers.
- **Cursor**: a general-purpose coding agent baseline implemented with Cursor.
- **PAT-Agent**: a baseline for formal model synthesis without exploiting proof information.
- **Event-B Agent**: our proposed framework.

The results demonstrate that our method consistently outperforms the three baselines on the metrics *PDR*, *RC*, and *RF*, both overall and across different levels of system complexity.

Consistency. Event-B Agent attains an overall *PDR* of 97.86%, indicating that the synthesized models are largely self-consistent, with only 2.14% of proof obligations remaining undischarged. In contrast, the LLM + auto provers, Cursor, and PAT-Agent baselines achieves *PDR*s of 89.20%, 90.07%, and 95.56% respectively, falling short of our method and exhibiting greater variance. Notably, Cursor's *PDR* fluctuates by 12.5% across different complexity levels, which makes it the most unstable method among all. Without a structured framework, its performance solely relies on the capability of the underlying LLM. While PAT-Agent attains stronger performance, its *PDR* still varies more than ours and declines as system complexity increases. In contrast, Event-B Agent consistently maintains *PDR* above 97.0% across all dataset partitions.

Correctness w.r.t. Requirements. Metrics *RC* and *RF* measure how well a formal model covers and satisfies system requirements. As discussed in Section 5.1, this requires validating the assumption of refinement correctness. The *PDR* for refinement-specific proof obligations (Refinement *PDR*) achieved by Event-B Agent is reported in Table 5. The full version of Event-B Agent reaches a Refinement *PDR* of 92.56%, indicating that the assumption holds for most refined models. On this basis, *RC* and *RF* can be used as approximations of requirement-level correctness.

The last two columns of Table 3 show that Event-B Agent achieves overall *RC* and *RF* values of 97.13% and 93.79%, respectively, which are the highest among all evaluated methods. In particular, *RC* exceeds the second-best method by 4.63% and *RF* by 18.01%. Across all dataset partitions, our method consistently delivers the best performance with lower variance on both metrics, demonstrating that models produced by Event-B Agent cover and satisfy the largest proportion of system requirements. It is also worth noting that lower *RC* and *RF* values imply that *PDR* could have been superficially inflated, as the generated formal system may bypass more challenging

requirements. Moreover, the ratio RF/RC for our method is 0.97, substantially higher than those of the three baselines (0.75, 0.77, and 0.82). This indicates that once a requirement is captured in the formal model, Event-B Agent is able to discharge nearly all corresponding proof obligations. We therefore conclude that Event-B Agent performs the best in terms of correctness with respect to system requirements.

System Complexity	Method	PDR	RC	RF
Simple	LLM + auto provers	0.8615	0.8889	0.7546
	Cursor	0.8418	0.9278	0.6370
	PAT Agent	0.9783	0.9167	0.7806
	Event-B Agent	**0.9954**	**0.9639**	**0.9417**
Medium	LLM + auto provers	0.9492	0.8974	0.7090
	Cursor	0.9668	0.9057	0.8114
	PAT Agent	0.9653	0.8623	0.7572
	Event-B Agent	**0.9700**	**0.9667**	**0.9347**
Complex	LLM + auto provers	0.8653	**0.9886**	0.6051
	Cursor	0.8936	0.8450	0.6175
	PAT Agent	0.9232	0.9825	0.7357
	Event-B Agent	**0.9706**	0.9834	**0.9373**
Overall	LLM + auto provers	0.8920	0.9250	0.6896
	Cursor	0.9007	0.8928	0.6886
	PAT Agent	0.9556	0.9205	0.7578
	Event-B Agent	**0.9786**	**0.9713**	**0.9379**

Table 3. Overall performance of Event-B Agent compared to baselines with different dataset complexities. GPT-5 with medium reasoning level is used as the backbond LLM for all methods.

5.3 RQ2. Ablation Study

The two key components of Event-B Agent are the refinement strategy planning step and the model & proof repair step. The latter integrates formal verification with a novel repair guidance system, namely, repair rules recommendation, fix strategy decision LLM, and atomic function execution. As shown in Section 5.2, Event-B Agent outperforms Cursor and PAT-Agent, both of which do not exploit proof information. To address RQ2, we perform a finer-grained evaluation regarding the model & proof repair step. Specifically, we examine the performance of Event-B Agent when proof information is available, but repair guidance is disabled, leaving the LLM to directly repair the model based on the proof state.

Ablation Baselines. The ablation study evaluates three variants:

(1) **None Enabled**: Both refinement and repair guidance are removed. This baseline isolates the capability of the LLM in model construction and repair when provided only with the model and verification results from the model checker and theorem provers.
(2) **Refinement only**: Refinement is retained, but repair guidance is removed. Since refinement introduces additional proof obligations and helps to reduce proving efforts at the same time, this baseline assesses how refinement affects the evaluation metrics.
(3) **Repair guidance only**: Refinement is omitted while repair guidance is retained. This baseline isolates the effectiveness of the repair guidance component.

Results and Discussion. Table 4 summarizes the ablation results. Across all metrics, the full version of Event-B Agent consistently delivers the strongest overall performance, outperforming

the ablation baselines. On the "Simple" and "Medium" partitions, some ablations obtain marginally higher scores, but these gains are limited and do not persist across the dataset as a whole. For *RC* and *RF*, the overall performance of the full version of Event-B Agent again surpasses all baselines by a significant margin, confirming the pattern observed in Section 5.2. Since these metrics reflect requirement-level correctness, lower coverage may unexpectedly inflate the observed *PDR* values. Taken together, these results highlight the complementary contributions of refinement and repair guidance to the effectiveness of Event-B Agent.

Dataset	Method	PDR	RC	RF
Simple	(1) None enabled	0.9279	0.7611	0.6833
	(2) Refinement only	0.9425	0.8944	0.8296
	(3) Repair guidance only	0.9890	**0.9861**	**0.9583**
	(4) Event-B Agent	**0.9954**	0.9639	0.9417
Medium	(1) None enabled	0.9799	0.8316	0.8316
	(2) Refinement only	**0.9861**	0.8970	0.8543
	(3) Repair guidance only	0.9659	0.9632	0.8863
	(4) Event-B Agent	0.9700	**0.9667**	**0.9347**
Complex	(1) None enabled	0.9600	0.9162	0.7954
	(2) Refinement only	0.9664	0.8951	0.8210
	(3) Repair guidance only	0.9529	0.8990	0.7548
	(4) Event-B Agent	**0.9706**	**0.9834**	**0.9373**
Overall	(1) None enabled	0.9559	0.8363	0.7701
	(2) Refinement only	0.9650	0.8955	0.8350
	(3) Repair guidance only	0.9693	0.9494	0.8665
	(4) Event-B Agent	**0.9786**	**0.9713**	**0.9379**

Table 4. Ablation studies on the Refinement Strategy Planning and Model & Proof Repair components of Event-B Agent.

Dataset	Method	Refinement PDR
Simple	Refinement only	0.7778
	Event-B Agent	**1.000**
Medium	Refinement only	0.7878
	Event-B Agent	**0.9556**
Complex	Refinement only	0.4653
	Event-B Agent	**0.8213**
Overall	Refinement only	0.6769
	Event-B Agent	**0.9256**

Table 5. Refinement PO Discharge Rate (Refinement *PDR*) of Event-B Agent and the refinement-related ablation baseline. Refinement *PDR* estimates the confidence that the assumption of refinement correctness holds.

While comparing baseline 2 ("Refinement only") and Event-B Agent with their counterparts without refinement (baseline 1, "None enabled", and baseline 3, "Repair guidance only"), we generally observe a positive trend across all metrics when refinement is present. There are, however, two notable exceptions. First, in the "Simple" partition, although Event-B Agent achieves a higher *PDR* than baseline 3, its *RC* and *RF* values are slightly lower. As system complexity increases, the *RC* and *RF* values for baseline 3 decline, and Event-B Agent surpasses it. This suggests that refinement becomes increasingly important for ensuring correctness in more complex systems. Second, in the "Complex" partition, baseline 2 achieves only slightly better *RF* than baseline 1, and a lower *RC*. From Table 5, we observe that the "Refinement only" baseline attains a Refinement *PDR* of just 46.53% in this partition, compared to 77.78% and 78.78% in the other two. This correlation between low Refinement *PDR* and reduced performance is consistent with the explanation in Section 5.1: without sufficiently validated refinement steps, the observed requirement-level metrics become inflated and unreliable. Overall, these results confirm that high Refinement *PDR* is essential for validating refinement links and preserving the inductive property in Event-B.

Regarding the repair guidance component, we observe that the "Repair guidance only" baseline performs significantly better than baseline 1 in the "Simple" partition, demonstrating its effectiveness in less complex systems. However, its *PDR* decreases as system complexity increases, likely due to implementation limitations. The repair module currently supports only a fixed set of atomic repair functions, which suffices for simple cases but is inadequate for more complex obligations. When combined with refinement, this limitation is mitigated. Refinement decomposes complex systems

into smaller ones, in which proofs fall within the reach of the repair guidance system, underscoring the complementary nature of the two components. This synergy is reflected in the Event-B Agent achieving the highest *PDR* in two partitions and in the overall results.

In summary, RQ2 shows that refinement and repair guidance are mutually reinforcing. Refinement reduces the complexity of individual proofs by decomposing model construction into smaller steps, making them more amenable to repair. Conversely, effective model and proof repairs improve the discharge of refinement obligations, enabling the construction of higher-quality models that provably preserve the requirements established in simpler abstract models.

5.4 RQ3. Efficiency

Dataset	Refinement Strategy Planning			Model Synthesis			Model & Proof Repair			Overall		
	Time	#Calls	#Tokens	Time	#Calls	#Tokens	Time	#Calls	#Tokens	Time	#Calls	#Tokens
Simple	1.22	1.00	4973.00	19.75	13.55	79359.56	15.76	15.55	127694.89	37.00	30.11	212027.44
Medium	1.20	1.00	4940.33	25.48	13.33	97841.11	52.89	53.22	2393013.44	83.21	67.56	2495794.89
Complex	1.18	1.00	6131.22	29.97	13.89	121982.78	62.46	59.45	2137659.11	103.14	74.33	2265773.11
Overall	1.20	1.00	5348.19	25.07	13.59	99727.81	43.71	42.74	1552789.15	74.45	57.33	1657865.15

Table 6. The efficiency of each Event-B Agent component and the overall framework. Time is the execution time in minutes, #Calls is the number of LLM calls, and #Tokens is the number of tokens consumed.

We evaluate the efficiency of Event-B Agent using three metrics: Time, the total execution time (in minutes) per component and overall, #Calls, the number of LLM invocations, and #Tokens, the number of tokens consumed. Table 6 reports their distributions across components and the overall model construction cost. Time and #Calls for refinement strategy planning remain stable across system complexities, as it runs once per system. Variation in #Tokens likely reflects differences in requirement counts. For model synthesis, #Calls are similar across partitions, while Time and #Tokens increase with system complexity, indicating higher per-call cost for complex systems. For model & proof repair, both Time and #Calls increase with complexity. Notably, #Tokens for "Complex" is lower than "Medium", consistent with our observation that refinement reduces proving effort. Although "Complex" has more POs, they are not necessarily harder.

The model & proof repair step takes on average 43.71 minutes, 42.74 LLM calls, and 1552789.15 tokens, largely due to high number of proof obligations (182.41 on average). #Calls is lower than PO count because many POs are discharged automatically, while the remaining require more complex reasoning or additional premises, where LLM-based model & proof repair becomes necessary. In practice, Event-B experts typically spend substantially more time on interactive proofs. In contrast, the average time to attempt discharging a single PO in Event-B Agent remains below 0.30 minutes across all partitions (0.18, 0.30, and 0.22 minutes), with an overall average of 0.24 minutes. Overall, this suggests that the repair overhead scales primarily with the number of POs, rather than exhibiting uncontrolled growth with system complexity.

5.5 RQ4. How does Event-B Agent construct and repair models?

Refinement. In Figure 6, we present how the three metrics evolve as refinement proceeds. Across all refinement steps, the average *PDR* is maintained at a consistently high level (97.86–98.50%). The average *RC* gradually increases from 31.19% in the abstract models to 97.13% in the final model,

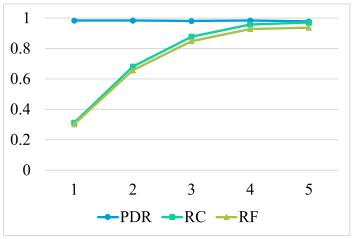

Fig. 6. Evolution of the three metrics across refinement steps.

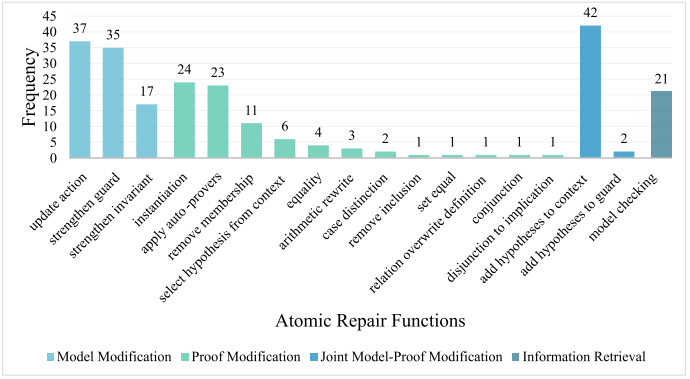

Fig. 7. Distribution of model and proof repair functions contributing to successful PO discharge.

while average *RF* rises from 30.24% to 93.79%. This trend shows that refinement effectively expands requirement coverage and fulfillment without compromising proof obligation discharge rate.

Atomic Repair Functions. In Figure 7, we show the frequency of atomic repair functions invoked by Event-B Agent that successfully contributed to discharging proof obligations. Overall, functions from all four categories are involved, demonstrating their effectiveness. Among them, model modifications account for 38.36% of successful invocations, followed by proof modifications at 33.62%. Joint model–proof modifications contribute 18.97%, showing the impact of operations that simultaneously affect both the model and the proof. Finally, retrieve information functions, such as model checking, make up 9.10%, indicating that even non-modifying actions play a role in guiding future repairs. This distribution shows that while proof- and model-level operations dominate individually, the hybrid category provides a substantial contribution.

Qualitative Evaluation of Repair. In this section, we present three representative repairs from the motivating example (Figure 2) to illustrate how Event-B Agent uses POs to guide model repair.

(1) Ensuring Well-Definedness. Initially, the invariant $FUN - 1$ generated by Event-B Agent is $has\_j = TRUE \Rightarrow f(j) = min(ran(f))$, which is not well defined because j could be out of the domain of f. The agent then enters the repair step and calls *StrengthenInvariant* on this invariant, and updates it to be $has\_j = TRUE \wedge j \in dom(f) \Rightarrow f(j) = min(ran(f))$. After this repair to the model, the well-definedness PO for this invariant is successfully discharged.

(2) Ensuring Refinement Correctness. In the abstract machine $M_{abstract}$, the event $stop\_empty$ doesn't assign a value to j, while the corresponding event in the concrete machine $M_{concrete}$ assigns $j := 0$. As a result, a PO has been generated to ensure that under guard $n = 0$ of this event, j must be 0. This PO cannot be discharged due to missing hypotheses. Event-B Agent repairs this by calling the *StrengthenInvariant* function, and adds invariant $n = 0 \Rightarrow j = 0$ to $M_{concrete}$, which helps to discharge the generated PO. Furthermore, invariant preservation POs are generated for this new invariant, and are all discharged in $M_{concrete}$, guaranteeing that the repair is consistent and correct.

(3) Invariant Preservation. As discussed in Section 4.4, Event-B Agent is capable of ensuring that every event preserves the invariants. In the motivating example, Event-B Agent calls the *StrengthenGuard* function and adds $grd7 : 1 \leq i$ to event $find\_not\_better$, since i can never be 0 due to the design of the algorithm. As a result, the post-state invariant inv' becomes vacuously true and the invariant preservation PO is discharged.

6 Related Work

6.1 Autoformalization and Repair

LLMs have been explored to reduce the effort of translating informal natural language requirements into formal specifications [49]. One direction is property synthesis, where natural language is mapped to LTL [16, 20], though this covers only a fragment of specification languages and cannot capture full system models. Other work addresses program specification generation [45, 46] or proof synthesis and repair [13, 19, 31], but these remain task-specific rather than providing an integrated modeling and verification workflow.

More recent studies target system-level models, such as Alloy [25] and Event-B [12], but these approaches are limited in scope and rely on direct text-to-code mappings. The closest work is PAT-Agent [50], which synthesizes PAT models and validates them via model checking. While effective for complex systems and requirements, its verification is confined to full-state-space model checking. By contrast, Event-B Agent combines bounded model checking with theorem proving, enabling counterexample-guided repair and proof-based correctness.

Beyond LLMs, the B and Event-B community has advanced construction and refinement methods [7, 18, 32], model repair [10, 11, 28], and code generation [21, 33, 39]. These efforts remain human-guided, repair-specific, or downstream, whereas Event-B Agent unifies synthesis, repair, and verification in an automated framework.

6.2 Neurosymbolic Methods

Neurosymbolic methods [14] combine the expressiveness of neural learning with the verifiability of symbolic reasoning. Recent directions include minimal neural activation patterns for verifiable robustness [23], automata embeddings for reinforcement learning with semantic guarantees [47], and specification learning from combined membership and preference queries [42]. Other efforts integrate logic into differentiable frameworks, such as tensorized LTL_f constraints verified in Isabelle/HOL and utilized for neural training [15], or world models that follow principles for physical interpretability, such as structuring latent spaces by physical intent and partitioning outputs for verifiability [38].

Event-B Agent shares this neurosymbolic philosophy but targets formal verification, using LLMs for semantic tasks while ensuring correctness through model checking and theorem proving in a unified synthesis-and-repair loop.

6.3 Proof Tactic Recommendation

Tactic recommendation in interactive theorem proving has long aimed to reduce user effort. Early approaches learned from existing proofs and replayed strategies, including PaMpeR for Isabelle/HOL [35], TacticToe for HOL4 [22], Tactician for Coq [9], and PSL, a strategy language that expands into concrete tactic sequences [36].

Recent work leverages transformers and LLMs. MagnusHammer [34] enhances premise selection with contrastive learning, while LeanDojo [48] combines LLMs with large proof libraries in a retrieval-augmented prover. Other directions explore proof-step prediction [24], recursive proof generation [44], lifelong agents such as LeanAgent [29], and approaches such as Lean-STaR [30], which interleave chain-of-thought reasoning with tactic prediction to improve proof success.

These efforts center on tactic prediction and retrieval within provers, whereas Event-B Agent embeds tactic guidance in a broader neurosymbolic framework, where model construction, verification, and repair progress together rather than as isolated tasks.

7 Threats to Validity

7.1 Internal Validity

Potential Biases and Assumptions of Requirements. Although our evaluation annotates requirements with lightweight labels (e.g., "EQP"), Event-B Agent can operate on unstructured natural language. The labels are used solely to compute RC and RF, require minimal annotation effort, and do not encode modeling structure. Since all methods use the same requirements and specification language, any bias from requirement phrasing affects them uniformly.

We also assume internally consistent requirements, whereas real-world specifications may be inconsistent. Handling such cases (e.g., supporting human-in-the-loop) is left for future work.

Requirement Capturing. To compute RC and RF, formal model elements are labeled with requirement identifiers during synthesis. In our experiments, this labeling is performed by LLMs but occasionally violates the expected format, affecting automatic matching and underestimating RC and RF. We manually corrected such cases for all methods to ensure fair and consistent evaluation.

Prompt Sensitivity and Reproducibility. LLM-based systems are sensitive to prompt phrasing. However, in Event-B Agent, the prompt structure is fixed for each task, with only task-specific information programmatically injected. More importantly, system behavior is governed by PO-guarded verification, and every LLM-proposed change is checked by deterministic tools (the Event-B compiler, model checker, and theorem provers), limiting the impact of minor prompt variations.

Although GPT-5 introduces nondeterminism through stochastic decoding, we mitigate this via schema-based constraints and verifier-mediated acceptance. Moreover, reproducibility here refers to the stability of the improvement process rather than identical modeling outputs, as multiple correct models may exist. The PO-guarded repair ensures that accepted changes do not regress the model with respect to the target PO, and our ablation results empirically demonstrate the robustness of the pipeline to LLM nondeterminism.

7.2 External Validity

Scalability. Since theorem proving is unbounded in principle, there is no theoretical limit on the system size supported. Although LLMs impose context-window constraints, these were not encountered in our experiments. For larger systems, refinement and decomposition can limit prompt scope by operating on partial models while preserving correctness via POs.

Implementation Limitations. The current libraries of repair rules and fix strategies are not exhaustive, as they were derived empirically from proof patterns across diverse systems. Extending these libraries to cover additional rules, proof states, and repair functions is left for future work.

8 Conclusion

We presented Event-B Agent, a novel framework for end-to-end formal model construction integrating refinement-based development with model–proof co-evolution. Evaluation on 27 formal systems demonstrates consistent improvements over baselines across multiple metrics, with ablation studies highlighting the complementary contributions of refinement and repair, while preserving reasonable efficiency. Future work includes expanding the library of repair rules and fix strategies, and supporting human-in-the-loop mechanisms to address requirement inconsistencies. Overall, this work takes a step toward practical and sound LLM-assisted formal development guided by the principle of correctness by construction.

9 Data Availability

The implementation of Event-B Agent and the datasets used are publicly available on GitHub[2].

[2] https://anonymous.4open.science/r/EventB_Agent-939D

References

[1] 2025. Cursor: The AI Code Editor. https://cursor.com/ Accessed: 2025-09-11.

[2] 2025. Event-B and the Rodin Platform. https://www.event-b.org/ Accessed: 2025-09-11.

[3] 2025. ProB Animator and Model Checker. https://prob.hhu.de/ Accessed: 2025-09-11.

[4] Jean-Raymond Abrial. 1996. The B-book: Assigning Programs to Meaning. *Cambridge University Press* (1996). doi:10.1017/CBO9780511624162

[5] Jean-Raymond Abrial. 2010. *Modeling in Event-B: system and software engineering*. Cambridge University Press. doi:10.1017/CBO9781139195881

[6] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774* (2023). doi:10.48550/arXiv.2303.08774

[7] Eman Alkhammash, Michael Butler, Asieh Salehi Fathabadi, and Corina Cîrstea. 2015. Building traceable Event-B models from requirements. *Science of Computer Programming* 111 (2015), 318–338. doi:10.1016/j.scico.2015.06.002

[8] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. cvc4. In *International Conference on Computer Aided Verification*. Springer, 171–177. doi:10.1007/978-3-642-22110-1_14

[9] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. The tactician: A seamless, interactive tactic learner and prover for coq. In *International Conference on Intelligent Computer Mathematics*. Springer, 271–277. doi:10.1007/978-3-030-53518-6_17

[10] Cheng-Hao Cai, Jing Sun, and Gillian Dobbie. 2019. Automatic B-model repair using model checking and machine learning. *Automated Software Engineering* 26, 3 (2019), 653–704. doi:10.1007/s10515-019-00264-4

[11] Cheng-Hao Cai, Jing Sun, Gillian Dobbie, Zhé Hóu, Hadrien Bride, Jin Song Dong, and Scott Uk-Jin Lee. 2022. Fast automated abstract machine repair using simultaneous modifications and refactoring. *Formal Aspects of Computing* 34, 2 (2022), 1–31. doi:10.1145/3536430

[12] Alfredo Capozucca, Daniil Yampolskyi, Alexander Goldberg, and Maximiliano Cristiá. 2025. Do ai assistants help students write formal specifications? a study with chatgpt and the b-method. In *2025 IEEE/ACM 37th International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 19–29. doi:10.1109/CSEET66350.2025.00009

[13] Pedro Carrott, Nuno Saavedra, Kyle Thompson, Sorin Lerner, João F Ferreira, and Emily First. 2024. 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*. 637–641. doi:10.1145/3663529.3663814

[14] Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, and Yisong Yue. 2021. Neurosymbolic programming. *Foundations and Trends in Programming Languages* 7, 3 (2021), 158–243. doi:10.1561/2500000049

[15] Mark Chevallier, Filip Smola, Richard Schmoetten, and Jacques D Fleuriot. 2025. Formally Verified Neurosymbolic Trajectory Learning via Tensor-based Linear Temporal Logic on Finite Traces. *arXiv preprint arXiv:2501.13712* (2025). doi:10.48550/arXiv.2501.13712

[16] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. 2023. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification*. Springer, 383–396. doi:10.1007/978-3-031-37703-7_18

[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[18] Guillaume Dupont, Yamine Ait-Ameur, Neeraj Kumar Singh, and Marc Pantel. 2021. Event-B hybridation: A proof and refinement-based framework for modelling hybrid systems. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 4 (2021), 1–37. doi:10.1145/3448270

[19] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. 2023. 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*. 1229–1241. doi:10.1145/3611643.3616243

[20] Francesco Fuggitti and Tathagata Chakraborti. 2023. 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*, Vol. 37. 16428–16430. doi:10.1609/aaai.v37i13.27068

[21] Andreas Fürst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. 2014. Code generation for Event-B. In *International Conference on Integrated Formal Methods*. Springer, 323–338. doi:10.1007/978-3-319-10181-1_20

[22] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. 2021. TacticToe: learning to prove with tactics. *Journal of Automated Reasoning* 65, 2 (2021), 257–286. doi:10.1007/s10817-020-09580-x

[23] Chuqin Geng, Zhaoyue Wang, Haolin Ye, and Xujie Si. 2024. Learning Minimal Neural Specifications. *arXiv preprint arXiv:2404.04662* (2024). doi:10.48550/arXiv.2404.04662

[24] Fabian Gloeckle, Baptiste Roziere, Amaury Hayat, and Gabriel Synnaeve. 2023. Temperature-scaled large language models for Lean proofstep prediction. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS'23*. https://openreview.net/forum?id=sSgdyY0YJR

[25] Yang Hong, Shan Jiang, Yulei Fu, and Sarfraz Khurshid. 2025. On the Effectiveness of Large Language Models in Writing Alloy Formulas. *arXiv preprint arXiv:2502.15441* (2025). doi:10.48550/arXiv.2502.15441

[26] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79. doi:10.1145/3695988

[27] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *arXiv preprint arXiv:2406.00515* (2024). doi:10.1145/3747588

[28] Tsutomu Kobayashi and Fuyuki Ishikawa. 2024. Repairing Event-B Models Through Quantifier Elimination. In *International Conference on Formal Engineering Methods*. Springer, 18–36. doi:10.1007/978-981-96-0617-7_2

[29] Adarsh Kumarappan, Mo Tiwari, Peiyang Song, Robert Joseph George, Chaowei Xiao, and Anima Anandkumar. 2024. LeanAgent: Lifelong Learning for Formal Theorem Proving. *arXiv preprint arXiv:2410.06209* (2024). doi:10.48550/arXiv.2410.06209

[30] Haohan Lin, Zhiqing Sun, Sean Welleck, and Yiming Yang. 2024. Lean-star: Learning to interleave thinking and proving. *arXiv preprint arXiv:2407.10040* (2024). doi:10.48550/arXiv.2407.10040

[31] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof automation with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1509–1520. doi:10.1145/3691620.3695521

[32] Atif Mashkoor, Faqing Yang, and Jean-Pierre Jacquot. 2017. Refinement-based validation of Event-B specifications. *Software & Systems Modeling* 16, 3 (2017), 789–808. doi:10.1007/s10270-016-0514-4

[33] Dominique Méry and Neeraj Kumar Singh. 2011. Automatic code generation from Event-B models. In *Proceedings of the 2nd Symposium on Information and Communication Technology*. 179–188. doi:10.1145/2069216.2069252

[34] Maciej Mikuła, Szymon Tworkowski, Szymon Antoniak, Bartosz Piotrowski, Albert Qiaochu Jiang, Jin Peng Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. 2023. Magnushammer: A transformer-based approach to premise selection. *arXiv preprint arXiv:2303.04488* (2023). doi:10.48550/arXiv.2303.04488

[35] Yutaka Nagashima and Yilun He. 2018. PaMpeR: proof method recommendation system for Isabelle/HOL. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 362–372. doi:10.1145/3238147.3238210

[36] Yutaka Nagashima and Ramana Kumar. 2017. A proof strategy language and proof script generation for Isabelle/HOL. In *International Conference on Automated Deduction*. Springer, 528–545. doi:10.1007/978-3-319-63046-5_32

[37] OpenAI. 2025. GPT-5. https://openai.com/gpt-5/ Released: 2025-08-07.

[38] Jordan Peper, Zhenjiang Mao, Yuang Geng, Siyuan Pan, and Ivan Ruchkin. 2025. Four Principles for Physically Interpretable World Models. *arXiv preprint arXiv:2503.02143* (2025). doi:10.48550/arXiv.2503.02143

[39] Victor Rivera, Néstor Catano, Tim Wahls, and Camilo Rueda. 2017. Code generation for Event-B. *International Journal on Software Tools for Technology Transfer* 19, 1 (2017), 31–52. doi:10.1007/s10009-015-0381-2

[40] Peter Riviere, Neeraj Kumar Singh, Yamine Aït-Ameur, and Guillaume Dupont. 2023. Formalising liveness properties in Event-B with the reflexive EB4EB framework. In *NASA Formal Methods Symposium*. Springer, 312–331. doi:10.1007/978-3-031-33170-1_19

[41] Peter Riviere, Neeraj Kumar Singh, Yamine Ait-Ameur, and Guillaume Dupont. 2025. Extending the EB4EB framework with parameterised events. *Science of Computer Programming* 243 (2025), 103279. doi:10.1016/j.scico.2025.103279

[42] Ameesh Shah, Marcell Vazquez-Chanlatte, Sebastian Junges, and Sanjit A Seshia. 2023. Learning formal specifications from membership and preference queries. *arXiv preprint arXiv:2307.10434* (2023). doi:10.48550/arXiv.2307.10434

[43] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. 2009. PAT: Towards flexible verification under fairness. In *International conference on computer aided verification*. Springer, 709–714. doi:10.1007/978-3-642-02658-4_59

[44] Haiming Wang, Huajian Xin, Zhengying Liu, Wenda Li, Yinya Huang, Jianqiao Lu, Zhicheng Yang, Jing Tang, Jian Yin, Zhenguo Li, and Xiaodan Liang. 2024. Proving Theorems Recursively. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 86720–86748. doi:10.52202/079017-2753

[45] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*. Springer, 302–328. doi:10.1007/978-3-031-65630-9_16

[46] Haoze Wu, Clark Barrett, and Nina Narodytska. 2023. Lemur: Integrating large language models in automated program verification. *arXiv preprint arXiv:2310.04870* (2023). doi:10.48550/arXiv.2310.04870

[47] Beyazit Yalcinkaya, Niklas Lauffer, Marcell Vazquez-Chanlatte, and Sanjit A Seshia. 2025. Provably Correct Automata Embeddings for Optimal Automata-Conditioned Reinforcement Learning. *arXiv preprint arXiv:2503.05042* (2025).

doi:10.48550/arXiv.2503.05042

[48] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 21573–21612. https://proceedings.neurips.cc/paper_files/paper/2023/file/4441469427094f8873d0fecb0c4e1cee-Paper-Datasets_and_Benchmarks.pdf

[49] Yedi Zhang, Yufan Cai, Xinyue Zuo, Xiaokun Luan, Kailong Wang, Zhe Hou, Yifan Zhang, Zhiyuan Wei, Meng Sun, Jun Sun, Jing Sun, and Jin Song Dong. 2025. Position: Trustworthy AI Agents Require the Integration of Large Language Models and Formal Methods. In *Forty-second International Conference on Machine Learning Position Paper Track*. https://openreview.net/forum?id=wkisIZbntD

[50] Xinyue Zuo, Yifan Zhang, Hongshu Wang, Yufan Cai, Zhe Hou, Jing Sun, and Jin Song Dong. 2025. PAT-Agent: Autoformalization for Model Checking. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2122–2133. doi:10.48550/arXiv.2509.23675

Received 2025-09-12; accepted 2026-03-24