Agentic Specification Generator for Move Programs

Yu-Fu Fu   Meng Xu[†]   Taesoo Kim
Georgia Institute of Technology   [†]University of Waterloo

***Abstract*—While LLM-based specification generation is gaining traction, existing tools primarily focus on mainstream programming languages like C, Java, and even Solidity, leaving emerging and yet verification-oriented languages like Move underexplored. In this paper, we introduce MSG, an automated specification generation tool designed for Move smart contracts. MSG aims to highlight key insights that uniquely present when applying LLM-based specification generation to a new ecosystem. Specifically, MSG demonstrates that LLMs exhibit robust code comprehension and generation capabilities even for non-mainstream languages. MSG successfully generates verifiable specifications for 84% of tested Move functions and even identifies clauses previously overlooked by experts. Additionally, MSG shows that explicitly leveraging specification language features through an agentic, modular design improves specification quality substantially (generating 57% more verifiable clauses than conventional designs). Incorporating feedback from the verification toolchain further enhances the effectiveness of MSG, leading to a 30% increase in generated verifiable specifications.**

***Index Terms*—LLM, Specification, Verification, Move**

I. Introduction

The need for high-assurance software has been increasingly recognized as software becomes more complex and critical. This is especially true for blockchains and smart contracts that now manage crypto assets worth billions of USD as any bug or vulnerability in the code can lead to significant financial losses for its stakeholders [1], [2]. One of the key techniques to deliver high-assurance software is formal methods such as model checking [3] and theorem proving [4]. Among these techniques, writing *formal program specification* that captures the intended behavior of a program is a crucial step, as it serves as the foundation for subsequent verification tasks.

However, writing specification is challenging, because it requires not only a deep understanding of the program's intended behavior but also expressing the intention in formal languages, which are often declarative in nature and require a different mindset than implementing an algorithm imperatively [5], [6], [7], [8]. Therefore, a more common practice in the smart contract community is that developers often implement based on loosely documented requirements first and then formally specify and verify the code later (if timing and budget permit) [9], [10]. While less optimal than a waterfall model (specifications before code), this practice is often necessary due to limited resources and time-to-market constraints.

And yet, late specification is better than no specification at all. To ease the burden of developers, a practical tool that automatically generates specifications from existing smart contract code, with acceptable quality, would be beneficial.

Extended version of the same paper published on ASE'25. Extra appendices are added at the end of the paper.

In this work, we join forces with the recent advances in large language models (LLMs) for formal methods (LLM4FM) [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27] and provide our own findings and insights based on our experience in developing an LLM-based tool, MSG, that automatically generates specifications (more precisely, functional pre/post conditions) for smart contracts written in an emerging language—Move [28].

MSG is based on the same rationale why LLMs should be used for specification generation—manually defined specification synthesis templates [21], [22] cannot match with the diversity and complexity of programs while LLMs can—as piloted in the literature [11], [12], [13]. However, existing works fail to provide more insights on the generalizability of their designs nor shed light on how to adapt to new programming/specification languages. So in this paper, we first seek to answer the following research questions (RQs) through the lens of Move:

RQ1 Will LLMs show degraded code comprehension and generation performance when the programming/specification language is not a mainstream one (hence ruling out fine-tuning opportunities as well)?

RQ2 As specifications need to be expressed in a formal language, what features of the specification language LLMs can leverage to generate better specifications?

RQ3 As the generated specifications will be eventually verified by a verification tool to prove that the code satisfies the specifications, will the feedback of such a verification tool help improve the quality of generated specifications?

Move is a suitable candidate for these RQs because it is a relatively new language (debuted in 2020 for the discontinued Diem blockchain [29] and later evolved by Aptos [30], Sui [31], and Movement [32] blockchains). Compared with mainstream ones like C, Java, or even Solidity which are preferred in prior works [11], [12], [19], [14], [13], Move has very limited codebases or documentation for LLMs to train or fine-tune on. And yet, formal verification is a first-class citizen in Move. In fact, Move comes with a built-in specification language called Move Specification Language (MSL), which allows developers to write expressive specifications for their code. The specifications are checked by an automatic verification tool—Move Prover [33]—which discharges proof obligations to SMT [34] solvers like Z3 [35] or CVC5 [36].

However, while addressing RQ1–3 help establish the feasibility of MSG, in order for MSG to be practical and useful in real-world Move codebases, we still need to resolve two technical concerns that are unaddressed in prior works:

RQ4 How to properly scope each LLM conversation context if the ultimate goal is to autonomously generate a useful set of specifications for the entire codebase without interrupting developers (i.e., human-in-the-loop)?

RQ5 While the accuracy (or quality) of generated specifications can be evaluated by comparing them against manually written specifications, how can we evaluate the comprehensiveness of the generated specifications?

Key Findings. The design of MSG is heavily influenced by the answers to these RQs, which we summarize as follows:

- LLMs show remarkable performance in comprehending Move code and generating Move specification (in MSL syntax) despite the language's recency (**RQ1**). Overall, MSG successfully generates specifications for 84% (300/357) Move functions in Aptos core (analogous to `libc` in C) with the OpenAI o3-mini model. Compared to expert-written specifications, MSG generates 82% of matching verification conditions and produces an additional 57% that differ; aggregately 39% more than the expert-written ones. This is on-par with results in prior works on different languages: *SpecGen* [12] shows overall 60% correctness on Java, while AUTOSPEC [11] achieves 79% correctness on C.
- Specification language matters and leveraging its features explicitly at tool design time (instead of implicitly by LLMs) can improve the quality of generated specifications (**RQ2**). MSL allows function pre/post conditions to be expressed in four different classes of specifications, each encodes developers intentions from a different angle. MSG actively leverages this by generating sub-specifications of different classes in MSL, with one specialized agent for each class, and subsequently *ensemble* sub-specifications into an idiomatic one. Our evaluation shows that the agentic design generates 57% more verifiable specifications than a conventional all-in-one design, which implicitly relies on LLMs to learn the differences between the classes of specifications.
- Feedback from the verification tool can significantly improve the quality of generated specifications (**RQ3**). While prior works have shown that leveraging compiler feedback help generate syntactically correct specifications [19], MSG goes a step further by leveraging the feedback from Move Prover (especially the counterexample) as an oracle to fix the wrongly generated specifications (similar to [12], [26]), which improves the accuracy of generation as shown in the evaluation results. We observe 30% more verifiable specifications when using the Move Prover feedback for all-in-one design in the ablation study of our evaluation. When prover feedback is removed from the agentic design, the accuracy does not dramatically drop because of the fail-safe design: 13.1% less function are verified. However, a clear decrease is observed in the quality of generated specifications: it generates 33% less of clauses than the agentic design with prover feedback.
- Scaffolding is necessary to keep LLMs focused on the context of the function being specified while also catering to the compositional nature of functional specification (**RQ4**). Sending the entire codebase to LLMs is obviously infeasible [27], and sending only the function being specified is often not enough, MSG implements a series of scaffolding utilities including static dependency analysis, selective function inliner, and a pretty-printer to lift Move abstract syntax tree (AST) back to source code after inlining. These utilities help MSG to scope the conversational context with LLMs autonomously without human-in-the-loop. For example, MSG can automatically decide whether inline the callee functions based on the best scoping strategy. During the evaluation, selective inlining unblocks 7 new functions for agentic design and 20 new functions for all-in-one design, respectively.
- Specification coverage, a simple metric to measure how much code in the function body is covered by specifications, can be used to evaluate comprehensiveness (**RQ5**). Our experiment shows that MSG generated unique clauses that were missed in expert-written specifications: 130 `ensures`, 86 `aborts_if`, and 75 `modifies` clauses. These account for 33.2% (291) of the total 876 clauses generated by MSG. Furthermore, the specification coverage component validates its outstanding performance, as MSG produces comprehensive specifications covering provided code.

II. BACKGROUND

In this section, we briefly describe the Move programming language, Move specification language (MSL), and Move Prover.

A. Example for Move Smart Contract and Move Specification

To get a taste of programming and specifying in Move, Figure 1 (page 3) shows a simple Move smart contract and its specification. In the implementation (Figure 1a), two data types, `Coin` and `Balance`, and three functions, `transfer`, `withdraw`, and `deposit`, are defined. `transfer` withdraws a coin from the sender's account and deposits it into the receiver's account by internally calling `withdraw` and `deposit` respectively. In `withdraw` and `deposit`, `borrow_global_mut` is used to borrow a mutable reference to the global state of the blockchain, which is a map from addresses to the data structure `Balance`.

The corresponding specification (Figure 1b) defines the intended behavior of these functions, which follows the standard style of Hoare triple [37] with precondition (`true`) omitted (although more complicated pre-conditions can be specified with `require` clauses). `aborts_if` clauses capture exhaustively the conditions under which the function should abort. The `modifies` clauses marks modifications to blockchain global state, and the effects of such modifications are often captured by `ensures` clauses in an axiomatic style. Collectively, `modifies`, `aborts_if`, and `ensures` clauses capture the complete set of post-conditions for the specified function.

As a concrete example, the specification of `transfer` consists of two parts: bindings (by `let` and `let post`) and properties. For bindings (Line 3-9), `global` is used to access the global state of the blockchain, which is an immutable (read-only) reference. `let` is used to bind the state of variables *before* the function is executed, and `let post` is used to bind the state of

```
1  module NamedAddr::BasicCoin {
2  /// This module defines
3  /// a minimal and generic Coin and Balance.
4    use std::signer;
5    struct Coin has store {
6      value: u64
7    }
8
9    struct Balance has key {
10     coin: Coin
11   }
12
13   /// Transfers `amount` of tokens from `from` to `to`.
14   public fun transfer(from: &signer, to: address,
15                       amount: u64) acquires Balance {
16       let from_addr = signer::address_of(from);
17       assert!(from_addr != to, EEQUAL_ADDR);
18       let check = withdraw(from_addr, amount);
19       deposit(to, check);
20     }
21
22   /// withdraw `amount` to get a Coin
23   fun withdraw(addr: address, amount: u64): Coin
24               acquires Balance {
25       let balance = balance_of(addr);
26       assert!(balance >= amount, EINSUFFICIENT_BALANCE);
27       let balance_ref = &mut
28         borrow_global_mut<Balance>(addr).coin.value;
29       *balance_ref = balance - amount;
30       Coin { value: amount }
31     }
32
33   /// despoit a Coin to addr
34   fun deposit(addr: address, check: Coin)
35              acquires Balance{
36     let balance = balance_of(addr);
37     let balance_ref = &mut
38       borrow_global_mut<Balance>(addr).coin.value;
39     let Coin { value } = check;
40     *balance_ref = balance + value;
41   }
42 }
```

(a) Move Smart contract for a Coin

```
1  spec NameAddr::BasicCoin {
2    spec transfer {
3      let addr_from = signer::address_of(from);
4      let balance_from =
5        global<Balance>(addr_from).coin.value;
6      let balance_to = global<Balance>(to).coin.value;
7      let post balance_from_post =
8        global<Balance>(addr_from).coin.value;
9      let post balance_to_post = global<Balance>(to).coin.value;
10     modifies global<Balance>(addr);
11
12     aborts_if !exists<Balance>(addr_from);
13     aborts_if !exists<Balance>(to);
14     aborts_if balance_from < amount;
15     aborts_if balance_to + amount > MAX_U64;
16     aborts_if addr_from == to;
17
18     ensures balance_from_post == balance_from - amount;
19     ensures balance_to_post == balance_to + amount;}
20
21   spec withdraw {
22     let balance = global<Balance>(addr).coin.value;
23     modifies global<Balance>(addr);
24
25     aborts_if !exists<Balance>(addr);
26     aborts_if balance < amount;
27
28     let post balance_post = global<Balance>(addr).coin.value;
29     ensures result == Coin { value: amount };
30     ensures balance_post == balance - amount;}
31
32   spec deposit {
33     let balance = global<Balance>(addr).coin.value;
34     let check_value = check.value;
35     modifies global<Balance>(addr);
36
37     aborts_if !exists<Balance>(addr);
38     aborts_if balance + check_value > MAX_U64;
39
40     let post balance_post = global<Balance>(addr).coin.value;
41     ensures balance_post == balance + check_value;}
42 }
```

(b) Corresponding specification

Fig. 1: An example of Move smart contract and its corresponding specification for a simple Coin used in Move-based blockchains.

variables *after* the function is executed. The `transfer` function modifies the global states of the struct `Balance` because of its calls to `withdraw` and `deposit`, which both also modify the global state of `Balance` (Line 10). The `transfer` function should abort when the `Balance` struct does not exist for either sender or receiver, on overflow, or when the sender has insufficient balance (Line 12-16). The `transfer` function, upon successful execution, should ensure that the sender's balance is reduced by the amount transferred and the receiver's balance is increased by the same amount (Line 18-19).

B. Move Prover

Move Prover [33] is an *automatic* formal verification tool specifically designed for Move, which as of now is primarily used within the Aptos blockchain ecosystem to ensure the correctness and security of smart contracts. Move Prover first translates specification and implementation into Boogie intermediate verification language [38], based on which the actual verification conditions (VCs) are produced in the form of SMT formulas, which are subsequently checked by SMT solvers such as Z3 [35] or CVC5 [36]. One notable features of Move Prover is its ability to provide feedback in the form of a call stack in Move source code when encountering verification failures by lifting the counterexample provided by Boogie back to the Move source code level. The original intention of this feature is to help developers debugging issues in code or specifications but such feedback can also be analyzed by LLM to auto-fix generated specifications, as later shown in §V.

III. MSG DESIGN

Figure 2 (page 4) illustrates the overall agentic design of MSG. To kickstart MSG, users specify the target Move function they want to generate specification for. MSG then performs static analysis to extract the function and its dependencies, such as definitions to (direct and indirect) callees, data structures, and constants involved. This enables MSG to produce two versions of conversation context for subsequent LLM interactions: (**V1**) target function with all callees inlined at best-effort (§III-D) which often leads to a single function with a complicated function body; (**V2**) target function with all callees' definition listed alongside, leveraging LLM's inherent capability to recognize function calls. Both versions are dispatched to separate specification generation (ClauseGen) agents (§III-A) as seed conversation context.

MSG employs four different ClauseGen agents, each specialized in generating a specific class of functional specification clauses expressible in MSL: loop invariants (if the code contains loops), abort conditions (i.e., `aborts_if` clauses), global state modification markings (i.e., `modifies` clauses), and axiomatic semantics for state changes (i.e., `ensures` clauses). Generated

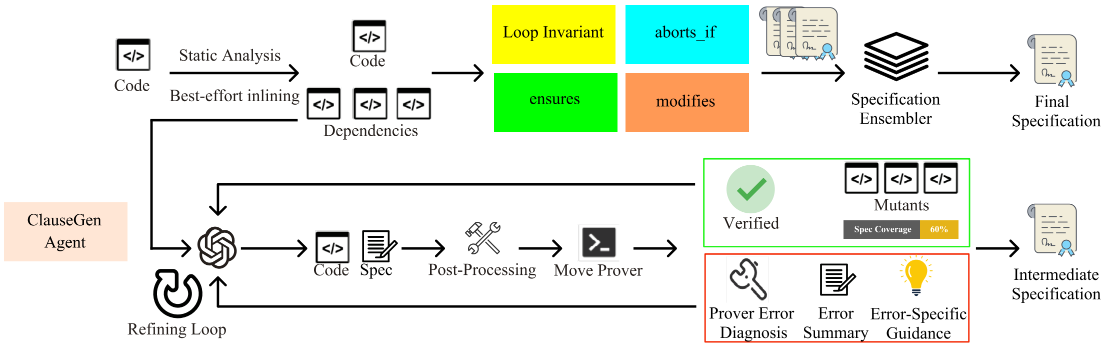

Fig. 2: Workflow of MSG. The top row shows the main agentic workflow of MSG, which contains 4 clauses generation (ClauseGen) agents for each class of specifications and the specification ensembler for idiomatic Move specifications. The bottom row shows a workflow in a single ClauseGen agent, which runs a generation loop to refine or fix the generated specification.

specification snippets from all agents are subsequently merged and optimized by a specification ensembler to produce a coherent and idiomatic set of specifications. While an alternative design could generate all classes of clauses in a single pass, MSG adopts an agentic design to avoid overwhelming LLM with too many instructions in a single system prompt, which is later proven effective in our evaluation (§V-B).

Inside each ClauseGen agent, MSG runs a generation loop to refine or fix the generated specification. After each round, the generated clauses are first vetted by the post-processing steps to fix common issues (e.g., misplaced semicolon or spaces), if any, followed by Move Prover to check whether the vetted specification, after ensembling, pass verification or fail with a diagnostic message (e.g. syntax error or a counterexample) (§III-B). If the ensembled specification fails, MSG will first instruct the LLM to summarize the diagnostic message and generate guidance for possible fixes. Besides the LLM-generated guidance, MSG also includes pre-defined guidance for common error patterns. All this information will be included in the user prompt for the next round of generation loop to help corresponding ClauseGen agent(s) fix wrong clauses. On the other hand, if the generated specification passes Move Prover, MSG will generate a series of mutated code snippets by randomly deleting parts of original Move code (i.e., nodes in the AST) to get the specification coverage (§III-C), which will be included in the user prompt for the next round to refine the specification. The whole process is repeated until a predefined bound (e.g. 5 rounds).

"Agentic" Terminology. We treat self-contained LLM components as distinct sub-agents—such as ClauseGen (which includes clause generation and error summary/guidance) and the specification ensembler—each handling specific sub-tasks. While the definition of "agent" is debated in communities, our approach uses these sub-agents in a controlled, orchestrated manner rather than through a dedicated autonomous planning agent. Given MSG's modular architecture, integrating a planning agent would be straightforward. Notably, ClauseGen adopts an agent pattern—the Reflection pattern [39]—which enables it to fix errors autonomously with the help of Move Prover diagnostics and LLM-powered error summaries that suggest possible fixes.

A. Agentic Design for Different Classes of Specifications

As previously explained, MSG uses specialized agents to generate four classes of clauses expressible in MSL—`aborts_if`, `modifies`, `ensures`, and loop invariants—independently. This design decision is motivated by the observation that generating all types of clauses in one-pass could be too burdensome for LLM. To be specific, in an all-in-one design, we need to pack comprehensive MSL guidelines in a *single* system prompt, in which not all instructions are closely followed by the LLM. To make things worse, failures to generate one class of clauses might lead to overall failures in the all-in-one design, which is not ideal for a specification generation tool—ideally, even if one class of clauses fails, the rest of the clauses might still be correct and useful for users.

Backed by the fact that different classes of clauses encode completely different aspects of the function semantics (i.e., they are inherently compositional) in MSL, MSG takes the stance that there is no need to generate them in a single pass. Instead, MSG devises an agentic design: specific ClauseGen agents with specialized prompts for each class of clauses:

`aborts_if` Clauses. This agent is to find possible scenarios of aborts, such as integer overflows (e.g., line 15 in Figure 1b), missing resources (e.g., line 12 in Figure 1b), or manually annotated abort conditions in the form of assertions.

We instruct LLM to carefully looks for various types of abort conditions to generate `aborts_if` clauses. Additionally, we independently run Move Prover to check whether `aborts_if false` is true. `aborts_if false` signals the intention that the function will never abort in any condition, which is the strongest predicate for this class of clause. Therefore, if the Move Prover passes the `aborts_if false` check, we skip abort condition generation and simply output `aborts_if false`.

`modifies` Clauses. When `borrow_global_mut` expressions are identified in the target function (and its callees), MSG launches this agent to find all such expressions, check whether the mutable references are actually written, and generate corresponding `modifies` clauses. In Move, if we need to verify that the caller function modifies the global state, we need to have specifications for the callee functions, too. For example, in Figure 1b, the function `transfer` modifies the `Balance` global resources because of the callees `withdraw` and `deposit`. To pass the Move Prover, in addition to the `modifies` (Line 10) in `transfer`, the ones in `withdraw` (Line 23) and `deposit` (Line 35) are also needed. Therefore, in the specialized system prompt for `modifies` clauses, we instruct LLM to also generate the `modifies` clauses for the callee functions.

`ensures` Clauses. This agent is used to generate the post-conditions of the function: for example, a function summary which captures the correspondence between the function inputs and outputs like the `transfer` example in Figure 1b. We instruct the LLM to identify operations linking input (function arguments), output (return values), and global state modifications. When a function modifies the global state, it should use `let` and `let post` to capture the state before and after execution. We also provide a prompt addressing common syntactic errors specifically in `ensures` clauses. For example, if a function contains branches (if-else), the LLM might generate disallowed constructs such as `ensures if (c) ... else ...`. Instead, we suggest encoding branches with the `==>` (implies) operator: `ensures c ==> ...; ensures !c ==> ...`, which significantly reduces errors when dealing with branches.

Loop Invariants. When the code context contains loops, we enable this agent to find loop invariants. However, unlike other agents that simply return the requested clauses, this agent will return an *annotated* function with loop invariants embedded in the function body (as required by the MSL syntax). In addition, loop invariant inference by itself is a complex task and an independent research topic even in LLM4FM [15], [16], [17], [18]. To generate loop invariants, we instruct the LLM to carefully inspect the loop body, identifying modified variables and preserved properties as candidates for invariants. We observed many syntactic errors in the generated invariants, largely due to limited training data for Move—especially for loop invariants. Therefore, in addition to an annotated function example with loop invariants, we include a set of invariant examples to guide the LLM.

Ensembler for Idiomatic Specification. Although MSL permits embedding all pre/post conditions and loop invariants in the code like other deductive verification tools with inline specification blocks, it is not the idiomatic way in Move. *Idiomatic* Move specification instead puts all clauses in a separate `spec` block (or even file) except for loop invariants. Additionally, in the `spec` block, MSL supports declaring variable bindings (`let`/`let post`) for convenience and clarity. Therefore, when we have multiple specification from the ClauseGen agents discussed above, although they specify different properties of the function, they might declare overlapping variable bindings, which is not allowed. Naively concatenating bindings and clause snippets from different ClauseGen agents leads to not only compilation errors but also an over-verbose set of specification. As a result, MSG uses an ensembler (another LLM agent) to merge the output from ClauseGen agents. On top of that, the ensembler also enforces coherent coding styles, such as ordering variable bindings, `modifies`, `aborts_if`, and finally `ensures` clauses in the `spec` block.

B. Verifier-in-the-Loop: Incorporating Move Prover Feedback

While powerful, LLMs can rarely generate an acceptable set of specification in one attempt. Failures may arise due to compilation errors (e.g., wrong syntax) or semantic errors (incorrect specifications). As LLMs cannot self-validate the generated specifications, we use Move Prover as an oracle for verification. Move Prover checks for compilation errors and, if none exists, attempts to prove that the code matches the generated specification. This process will yield one of the following outcomes: ❶ the generated specification passes the prover, ❷ the generated specification fails with a counterexample generated, ❸ the verification times out, or ❹ a compilation error is reported. As long as verification fails (❷,❸,❹), we include the Move Prover diagnostics, error summary, pre-defined guidance on common fixes for known error types, and the counterexample (if any) in the prompt for one or more responsible ClauseGen agents to re-generate a new set of clauses in the next round.

More specifically, if the verification failure is caused by an incorrect `ensures` clause, MSG will instruct the `ensures` agent (and the loop invariant agent if loops are involved) to re-generate a new set of clauses. MSG also attempts to provide some pre-defined guidances for common errors by pattern matching the prover error message. The most common errors include: the usage of undefined functions or impure functions (e.g. containing early returns in CFG) that cannot be used in the specification because they cannot be translated to Boogie. We use regular expressions to match the prover error message to find those bad function usages and guide LLM to avoid them in the next round of generation loops.

C. Specification Coverage to Pinpoint Missing Parts

Generating a complete (strong) specification for a function is notoriously hard. Therefore, Move Prover accepts partial specifications and expressions in MSL have partial semantics by design. This implies that there might be multiple specifications that can pass the verification for a given function. To differentiate and rank them, we introduce a metric called *specification coverage*, analogous to traditional line code coverage. Intuitively, given two specifications, s_a is considered more complete than s_b if $cov_{s_a} > cov_{s_b}$.

Unlike code coverage which can be readily collected during execution, we cannot instrument specifications and track how they are used in the proof obligations. Instead, we approximate specification coverage via a method called *random deletion* (inspired by FAST [40]). The idea is simple: if the implementation is incorrect after deleting parts of code, then a correct specification s should fail the Move Prover, unless s is incomplete and captures only part of the behavior.

Consider the function f in Figure 3a (page 6) with its complete specification s^+ in Figure 3c (verifying both elements of a pair), versus a modified function f^- in Figure 3b and its incomplete specification s^- in Figure 3d (verifying only

```
public fun add_pair(a: (u64, u64), b: (u64, u64)): (u64, u64) {
    let (a0, a1) = a; let (b0, b1) = b;
    (a0 + b0, a1 + b1)
}
```

(a) Original Move Code f

```
public fun add_pair(a: (u64, u64), b: (u64, u64)): (u64, u64) {
    let (a0, a1) = a; let (b0, b1) = b;
    (a0 + b0, 0) // changed to 0 for deletion
}
```

(b) Modified Move Code f^-

```
spec add_pair(a: (u64, u64), b: (u64, u64)): (u64, u64) {
    let (a0, a1) = a; let (b0, b1) = b;
    ensures result.0 == a0 + b0;
    ensures result.1 == a1 + b1;
}
```

(c) Complete Specification s^+

```
spec add_pair(a: (u64, u64), b: (u64, u64)): (u64, u64) {
    let (a0, a1) = a; let (b0, b1) = b;
    ensures result.0 == a0 + b0;
    // ensures result.1 == a1 + b1; // miss
}
```

(d) Incomplete Specification s^-

Fig. 3: Example to show the random deletion of Move code. (a) Original Move code, (b) Modified Move code, where the deleted parts are marked with a comment, (c) Complete specification, and (d) Incomplete specification.

one element). Here, s^- can pass the prover for both f and f^-, while s^+ passes only for f, indicating that s^+ is more complete.

We implement random deletion at the AST level, and the specification coverage measurement is as follows:

① Randomly delete parts of the AST (e.g., code blocks, statements, expressions) of function to create mutants.
② Lift modified AST to source code and check the code mutant against the generated specification by Move Prover.
③ If verification passes, the deletion is not covered by the specification; otherwise, the part is covered.

We then include line diffs for the deleted codes to pinpoint the non-specified parts in the prompt for the next round.

D. Construct Context for Specification Generation

The full dependency information for the target function is indispensable for verifiable specification generation. While including the entire Move library is tempting, this is unrealistic given LLM context length limits and instruction-following capabilities. To address this, MSG uses static analysis to slice the target and its dependencies from the library and packs them as the normal conversion context (**V2**). However, even after slicing, the isolated dependencies might still be overwhelming. Thus, we perform best-effort function inlining to selectively fold as many dependencies as possible into the target function's body, reducing LLM burden in reasoning across multiple function boundaries. The inlined target function and remaining non-selected dependencies are then packed as the inlined conversion context (**V1**). Both contexts are dispatched to ClauseGen agents for specification generation.

Static Analysis for Cross-module Function Dependencies. We perform interprocedural def-use analysis to identify target functions and their dependencies (structs, constants, and both direct and indirect callees). Using this analysis, MSG tracks *cross-module usage* and uncovers dependencies often missed by similar tools limited to single code snippets.

Best-effort Function Inlining to Reduce Context Complexity. MSG intercepts compilation to eagerly inline callees. Naively inlining all callees is infeasible due to compiler safety checks (e.g., resource safety). Instead of exhaustively enumerating safe combinations, MSG uses a *best-effort* monotonic approach: it attempts to inline the first callee, inlining it if compilation passes; if not, it skips it. This continues for each callee, with later ones skipped if they conflict with earlier inlines.

The inlined representation is then generated as an AST, converted back to Move code via an AST pretty-printer, and used to build the inlined conversion context (**V1**).

E. Abstract Specification as Last Resort

When all ClauseGen agents fail to generate a specification, e.g., due to excessive complexity or unsupported operations (e.g., non-linear arithmetic), MSL allows writing abstract specifications [41] using *uninterpreted functions* [42] as stubs. These stubs may later be replaced with concrete specifications or serve as implementation contracts. Thus, instead of abandoning the effort when the LLM fails across all specification classes, MSG instructs the LLM to generate abstract specifications as proof templates to assist experts. However, we do not consider abstract specifications as verifiable specifications in evaluating MSG as they are only placeholders for developers essentially. §A shows a concrete example of an abstract specification generated by MSG.

IV. Implementation

MSG is implemented within the Aptos Move toolchain monorepo for ease of integration and code reuse. MSG consists of the core agentic system that generates, fixes, and refines specifications (5k LoC) as well as scaffolding utilities (e.g., dependency analysis, pretty-printer, scripts) consisting of 2.3k LoC. The entire monorepo with MSG included can be found in the supplementary material associated with this paper. MSG is open source at https://github.com/sslab-gatech/MSG.

V. Evaluation

We evaluate MSG based on the research questions (RQs) that both motivate and influence its design (§I):

- **RQ1:** (*Effectiveness*) How effective is MSG in generating acceptable specifications for real-world codebases? (§V-A)
- **RQ2, RQ3, RQ4:** (*Ablation Study*): How does each component affect the performance of MSG (§V-B)? More specifically: is the agentic design of MSG beneficial? is the Move Prover feedback useful? is function inlining helpful?
- **RQ5** (*Comprehensiveness*): How complete are the specifications generated by MSG? Are there gaps in coverage (§V-C)?

Experiment Settings. MSG attempts to generate specifications in multiple rounds as shown in Figure 2. During this evaluation, we set the number of rounds to 5. We conduct the same experiments 3 times to reduce randomness from the LLM.

Target Move Codebase.

We have evaluated MSG on the following foundational Move libraries on Aptos:

- move-stdlib: standard libraries like vector, string.
- aptos-stdlib: extended standard libraries designed for Aptos.
- aptos-framework: Aptos Framework libraries including account, coin, etc, which are used to build the smart contracts.

Sui [31], another popular Move-based blockchain mentioned in §I, is not chosen as the Move Prover is not compatible with the extended UTXO model embedded in Sui Move.

Target Function Selection. Not all the functions in the target Move codebase are selected as the target for specification generation. We exclude 1) native functions which are implemented in Rust instead of Move. 2) functions that are marked as not verified (marked with `pragma verify = false`). Table I presents the breakdown details for each library: 357 functions in total, 17 of them containing loops. The small number of loop-related functions arises because: (1) smart contracts seldom use loops, and (2) loops mostly occur in low-level libraries, such as vector modules.

Codebase	LOC	Public	Private	Args	Loops
move-stdlib	82	10	0	15	8
aptos-stdlib	397	58	5	122	1
aptos-framework	2,780	205	79	492	8
TOTAL	3,259	273	84	629	17

TABLE I: Code metrics for three Move codebases. 357 functions in total, 17 of them containing loops. LOC counts only lines inside function bodies (type/constant declarations excluded). Public/Private are function counts; Args is the total number of formal parameters across all functions; Loops counts functions with loops.

Specification Quality Metrics. We use two metrics to gauge the quality of generated specifications: verifiability and comprehensiveness. Verifiability simply means that the generated specification, combined with the implementation of the target function, produces valid verification conditions that can be proved by the Move Prover.

Regarding specification comprehensiveness, Aptos smart contracts include expert-written Move specifications for parts of their codebase. To assess whether the specifications generated by MSG are comparable to these experts' ground-truth specifications, we define a metric called *specification comprehensiveness*, which is checked by subsumption [43].

Consider the complete specification s^+ in Figure 3c, which proves both the first and second members of the result pair, versus an incomplete specification s^- in Figure 3d, which only proves the first member. In this case, the complete specification is stronger, as it implies the incomplete one ($s^+ \implies s^-$).

We formalize this idea as follows. Given a generated specification s_g and an expert-written specification s_e, we first decompose s_e into verification conditions:

$$s_e \equiv s_{e_1} \wedge s_{e_2} \wedge \cdots \wedge s_{e_n},$$

where each s_{e_i} corresponds to clauses such as `ensures` and `aborts_if` in Move. We then check for each $i \in \{1, 2, \ldots, n\}$ whether s_g implies s_{e_i} and count the number of successful implications. The ratio of successful cases over n serves as the specification comprehensiveness.

Automatic splitting and implication checking is non-trivial. Fortunately, with a manageable number of functions (357), we manually extracted and split the expert-written specifications from the Aptos Move codebase, and verified each implication through human inspection or by running the Move Prover.

LLM Backends. During the evaluation, we use OpenAI's LLMs as the backend for MSG. We use the official API endpoints without web-browsing features. Both base models and reasoning models are used, as follows:

- OpenAI o3-mini: the compact reasoning model with default medium reasoning effort (o3-mini-medium).
- OpenAI GPT-4o: the general-purpose model.
- OpenAI GPT-4o-mini: the compact general-purpose model.

During our evaluation, the weaker models, OpenAI GPT-4o and GPT-4o-mini, are only used by the simpler design, MSG_{AIO}, for the ablation study (§V-B) to showcase that stronger models have better performance in generation as expected. We also evaluate weaker models (GPT-4o and GPT-4o-mini) in addition to o3-mini results (see Table VI and Table VII).

A. *Overall Effectiveness of* MSG

Specification Verifiability. As shown in Table II (page 8), MSG successfully generated valid specifications for *84%* of the target functions (300 out of 357), which is a clear indication that MSG is effective in generating specifications for real-world Move codebases. The specifications generated by MSG cover all 10 functions in move-stdlib, 48 out of 63 in aptos-stdlib, and 242 out of 284 in aptos-framework as shown in Table III (page 8). Additionally, MSG generates abstract specifications (§III-E) for the remaining 57 functions, which serve as placeholders for experts to further complete. Finally, within the 357 functions, 17 of them contain loops. MSG successfully generates verifiable loop invariants for 14 out of 17 functions, which is at the same level of accuracy as generation for loop-free functions.

Specification Comprehensiveness. Figure 4 (page 8) shows the distribution of specification comprehensiveness percentage for all generated specifications by MSG that are verifiable across the Aptos libraries. A "full match" indicates that all manually written clauses in the codebase (e.g., `ensures`, `aborts_if`, and `modifies`) are implied by the generated specification. MSG shows strong performance in the less complex libraries (move-stdlib and aptos-stdlib) with 86.2% (50/58) fully matched, but achieves only 63.8% (120/188, excluding functions without ground-truth conditions) in the more complex aptos-framework. We observe that the missing clauses from MSG are mostly `aborts_if` clauses, which LLMs tend to miss in functions with deeply-nested calls.

	MSG	MSG$^-$	MSG$_{\text{inline}}$
Fail	0	0	0
Success	300	253	299
Abstract	57	104	58
`ensures`	466	328	510
`aborts_if`	252	182	245
`modifies`	144	73	89
`Loop invariants`	14	7	14
All clauses	876	590	858
$\frac{\text{Success}}{Total}$	**84%**	70.9%	**83.8%**
% of Clauses	**100%**	67.6%	97.9%

	MSG$_{\text{AIO}}$	MSG$^-_{\text{AIO}}$	MSG$_{\text{AIO-inline}}$	MSG$_{\text{AIO-naive}}$
1st	132	140	132	106
2nd	33	6	35	18
3rd	14	0	15	7
4th	8	0	7	7
5th	4	1	4	3
Fail	54	77	47	220
Success	191	147	193	137
$>$ 1st	59	7	61	31
Abstract	112	133	117	0
$\frac{\text{Success}}{Total}$	53.5%	41.1%	54.1%	38.3%
$\frac{\text{1st}}{\text{Success}}$	69.1%	95.2%	68.4%	77.4%
$\frac{>\text{1st}}{\text{Success}}$	30.9%	4.8%	31.6%	22.6%
$\frac{\text{Success}}{\text{Success+Abstract}}$	63%	52.5%	62.3%	100%

TABLE II: Comparison of agentic design (MSG variants) and all-in-one design (MSG$_{\text{AIO}}$ variants) using o3-mini model.
Left: Results of MSG, MSG$^-$, and MSG$_{\text{inline}}$ with counts of failures, successes, and abstracts specification along with clause metrics (`ensures`, `aborts_if`, `modifies`), loop invariants, and success rates.
Right: Detailed results for all-in-one variants showing round-by-round performance where 1st to 5th represents the earliest successful round. "Fail" means the system cannot generate specification within five rounds. "Success" is the sum of 1st to 5th. "> 1st" means non-one-shot generation (sum of 2nd to 5th). "Abstract" is the number of generated abstract specifications. The success rate, one-shot rate, and concrete specification rate are shown in the bottom rows.

	move-stdlib	**aptos-stdlib**	**aptos-framework**
Verified Functions	10	48	242
Expert-written Clauses	29	149	442
Generated Clauses	29	143	690
Matched Clauses	29	133	347
Loop Invariants	8	1	5
Unique `ensures`	0	4	126
Unique `aborts_if`	1	3	82
Unique `modifies`	0	3	72
$\frac{\textbf{Match}}{\textbf{Total}}$	100%	89.3%	78.5%
$\frac{\textbf{Generated}}{\textbf{Total}}$	100%	96%	156.1%
$\frac{\textbf{Uniques}}{\textbf{Generated}}$	3.4%	7%	40.6%

TABLE III: Comparison of the number of verified functions, expert-written, generated, matched clauses, loop invariants, and unique clauses for different codebases. Please refer to Table I: loop invariants are from 17 selected loop-containing functions.

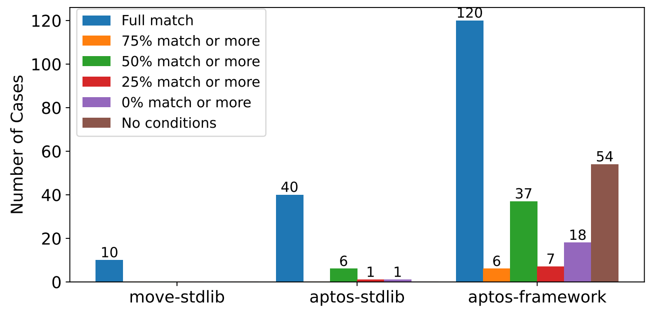

Fig. 4: Specification comprehensiveness distribution of verifiable specifications by MSG.

Variant	A	P	S	I	V
MSG	✓	✓	✓	✗	84.0%
MSG$^-$	✓	✗	✓	✗	70.9%
MSG$_{\text{inline}}$	✓	✓	✓	✓	83.8%
MSG$_{\text{AIO}}$	✗	✓	✓	✗	53.5%
MSG$^-_{\text{AIO}}$	✗	✗	✓	✗	41.1%
MSG$_{\text{AIO-inline}}$	✗	✓	✓	✓	54.1%
MSG$_{\text{AIO-naive}}$	✗	✗	✗	✗	38.3%

TABLE IV: Feature breakdown and verifiability for all variants used in the evaluation and ablation study. Abbreviations: A = Agentic design, P = Prover feedback, S = System Prompt & Static Analysis, I = Function inlining, V = Verifiability rate with o3-mini.

Table III summarizes the *aggregated* statistics: MSG generates 100% clauses for move-stdlib, 89.3% for aptos-stdlib, and 78.5% for aptos-framework, aggregating to 82.1% overall. Although some generated specifications are not fully matched, they remain useful—often yielding even *aggregately* more verification conditions than the expert-written versions (*aggregately* 139%: 82% matching ones plus an additional 57% that differ). Among them, MSG also generates *unique clauses* that manually-written specifications miss: 3.4% in move-stdlib, 7% in aptos-stdlib, 40.6% in aptos-framework, and overall 33.2% (291) of all 876 generated clauses. The excellent performance corresponds to less presence of specification coverage feedback discussed later in §V-C.

Overall, these statistics answer **RQ1**: LLMs can reason about Move programs well even though Move is an emerging language, which is validated by the strong performance of MSG in generating not only more verifiable specifications but unique ones compared to expert-written specifications.

Reasons for Not Generating Specifications. MSG fails to generate specifications for 57 functions (or rather it generates an abstract specification as a last resort). The main reasons include:

- *Calling impure functions.* Impure functions are Move functions that have side effects, such as modifying global storage and those functions should not be called as an expression in the specification. Instead, the idiomatic way is to write a pure version of the function as a helper function and call it in the specification. While we explicitly instruct the LLM to do this, it still sometimes calls them, causing failures.
- *Calling non-existent functions.* We observe that LLMs might still hallucinate and generate specifications that invoke non-existent functions. This could be due to the fact that LLMs are trained with limited data on Move and improvised with knowledge from other programming languages.
- *Other compiler errors.* LLMs sometimes generate code that is not valid Move syntax, including type errors or calling a function with the wrong number of arguments. This might be another sign of hallucination.
- *Complexity of the target function.* Some functions are too complex for the LLM to understand, especially those with deeply-nested calls or branches that appear in the aptos-framework (as shown in Figure 4 as well).

B. *Ablation Study for* MSG *and* MSG$_{AIO}$

In this section, we evaluate the agentic design and its underlying components. This ablation study highlights the contributions of each component in our design.

MSG$_{\text{AIO}}$: All-in-one Version of MSG. In addition to the agentic version of MSG, we implement an all-in-one variant, MSG$_{\text{AIO}}$, which was our initial approach to explore the capability of LLMs in generating Move specifications. The key difference is that MSG$_{\text{AIO}}$ generates all classes of specifications in a single generation loop, whereas MSG generates them separately and then merges the results using the specification ensembler. Notably, MSG$_{\text{AIO}}$ lacks the prover error summary agent, the guidance for addressing prover errors, and the post-processing steps used to resolve common prover issues.

To illustrate the benefits of our agentic design, we compare MSG with MSG$_{\text{AIO}}$. Furthermore, to assess the individual contributions of the underlying components (i.e., curated system prompts, static analysis, and prover feedback), we evaluate three variants of MSG$_{\text{AIO}}$:

- **MSG$_{\text{AIO-naive}}$:** A baseline version without any features.
- **MSG$^{-}_{\text{AIO}}$:** MSG$_{\text{AIO-naive}}$ enhanced with a system prompt and static analysis, but without prover feedback.
- **MSG$_{\text{AIO}}$:** The full MSG$_{\text{AIO}}$ incorporating the Move Prover.

Based on our evaluation, prover feedback significantly contributes to the overall accuracy of MSG$_{\text{AIO}}$. Consequently, we introduce a variant of MSG, called MSG^{-}, which employs the agentic design without incorporating Move Prover feedback. In later paragraphs, we evaluate the effect of function inlining with MSG$_{\text{inline}}$ and MSG$_{\text{AIO-inline}}$. Table IV shows feature breakdown and verifiability for all variants that will be used in the evaluation and ablation study.

Table II presents the breakdown and aggregated results from three runs, comprising 357 functions with three trials per function. In our evaluation framework, a function is considered successful if any one of the three trials produces a verifiable specification. If multiple trials succeed, the trial with the earliest successful round is recorded.

Effectiveness of the Agentic Design. Comparing the results between MSG and MSG$_{\text{AIO}}$, the benefits of the agentic design become clear. With the advanced o3-mini model, MSG produced non-abstract specifications for 84% of the cases (300 out of 357), while MSG$_{\text{AIO}}$ achieved only 53.5%. In addition, MSG yielded abstract specifications for 57 functions, whereas MSG$_{\text{AIO}}$ generated 112 abstract specifications but outright failed on 54 functions. These results demonstrate that the agentic design significantly enhances verifiability, thereby addressing **RQ2**: leveraging MSL features (four compositional classes of specifications) truly makes a difference.

Round and Move Prover Feedback. To assess the impact of Move Prover feedback for **RQ3**, we compare MSG$^{-}_{\text{AIO}}$ with MSG$_{\text{AIO}}$, and MSG with MSG^{-}, respectively. With additional rounds in the generation loop, the LLMs have more opportunities to refine or fix the generated specifications with the aid of the Move Prover. In contrast, without prover feedback, the later rounds offer limited corrective value, as the LLM lacks guidance regarding specific errors.

The results show that Move Prover feedback is indeed beneficial for the generation process. For example, in the all-in-one design MSG$_{\text{AIO}}$, with o3-mini, 30.9% (59 out of 191) of verifiable specifications were generated in later rounds using prover feedback, compared to only 4.8% in MSG$^{-}_{\text{AIO}}$ (high one-shot rate 95.2%, but little self-correction). Similarly, when comparing the agentic design MSG with MSG^{-}, the overall accuracy for MSG^{-} drops by 13.1% from 84% to 70.9% after removing prover feedback. Moreover, MSG^{-} verifies only 13.1% fewer functions while generating just 67.6% of the clauses compared to MSG (a 32.4% reduction), indicating that Move Prover feedback improves both the accuracy and quality of generated specifications.

This substantial difference demonstrates that the improvement primarily stems from incorporating prover feedback rather than solely increasing the number of rounds, thereby answering **RQ3**: Move Prover feedback is indeed useful.

System Prompts and Static Analysis. The naive version of all-in-one design, MSG$_{\text{AIO-naive}}$, achieves only 38.3% verifiability with o3-mini and 18.4% with GPT-4o-mini (detailed results in Table VI), showing that without enriched prompts, the LLM generates incorrect Move syntax and ignores callee function contents. By using a curated system prompt with examples and applying static analysis to capture function dependencies, MSG$^{-}_{\text{AIO}}$ significantly improves performance: 41.4% verifiability with o3-mini and 29.1% with GPT-4o-mini—a 58.2% improvement over MSG$_{\text{AIO-naive}}$ for GPT-4o-mini. This demonstrates the value of these enhancements, particularly for less capable LLMs, and shows that static analysis effectively builds the correct context for specification generation, partially answering **RQ5**.

Best-effort Function Inlining. We assess whether inlining functions affects MSG's performance, which ultimately depends on the LLM's understanding of function calls. Inlining is a double-edged sword: it reduces the number of functions processed, but may increase the target function's complexity—especially when the inlined function contains branches (e.g., if-else statements). Table II includes results for the normal (with conversation context **V1**) and inlined versions (**V1** and **V2**) of MSG and MSG_{AIO}: $\text{MSG}_{\text{inline}}$ and $\text{MSG}_{\text{AIO-inline}}$. Overall, the aggregated performance difference is minimal: MSG generates one more specification than $\text{MSG}_{\text{inline}}$, and MSG_{AIO} generates two fewer than $\text{MSG}_{\text{AIO-inline}}$.

	Normal-only	Common	Inline-only	
MSG	8	292	7	$\text{MSG}_{\text{inline}}$
MSG_{AIO}	18	173	20	$\text{MSG}_{\text{AIO-inline}}$

TABLE V: Table representation of Venn diagrams comparing the generated specifications between the normal and inlined variants for MSG and MSG_{AIO}. "Common" means the specifications that could be found by both normal and inlined versions.

Table V further uses Venn diagram [44] (in table form) to illustrate the specification sets produced by the normal and inlined versions for both MSG and MSG_{AIO}. The evaluation results confirm our hypothesis: inlining has both positive and negative effects on the LLM's specification-generation capability. Specifically, $\text{MSG}_{\text{inline}}$ generates 7 specifications that are missed by MSG, while it misses 8 specifications that MSG does produce. Similarly, $\text{MSG}_{\text{AIO-inline}}$ generates 20 specifications not produced by MSG_{AIO}, but fails to generate 18 ones that MSG_{AIO} does.

Analysis of these discrepancies reveals that the inlined version could be overwhelmed by the complexity of inlined functions (e.g. deeply-nested calls or branches). In such cases, attempting to cover all aspects of the function in a single large body could fail, even with the corrective help of Move Prover.

On the positive side, we observe that the inlined version sometimes offers better reasoning than the normal version, particularly when dealing with *reasonably-nested* calls. In scenarios where multiple functions with small bodies are combined into a single larger function, the unified context can be easier for the LLM to understand, which highlights our design choice for best-effort function inlining.

In conclusion, our results confirm that inlining is beneficial for certain programs, which partially addresses **RQ4**. A promising approach to optimize performance is to combine the advantages of both the normal and inlined versions. Running them in parallel and subsequently selecting or merging the best results using a specification ensembler can lead to overall improved generation verifiability.

Summary. Our ablation study shows that the agentic design of MSG significantly improves accuracy by leveraging the compositional nature of various Move specifications (**RQ2**). With Move Prover feedback, both MSG and MSG_{AIO} gain the ability to self-fix erroneous specifications, drastically improving accuracy and quality (**RQ3**). Finally, scaffolding utilities like static analysis and best-effort function inlining effectively enhance the accuracy of generated specifications by providing well-scoped conversation contexts (**RQ4**).

C. Comprehensiveness for Generated Specifications

To evaluate the comprehensiveness of the generated specifications, we utilize the specification coverage component in MSG. During the evaluation, the specification coverage component was enabled in the `ensures` ClausesGen agent, which attempts to identify missing `ensures` clauses, similar to those illustrated in the simplified example in Figure 3. By examining the execution traces of MSG and MSG_{AIO}, we filter cases where specification coverage feedback is triggered (i.e., non-empty). Using `o3-mini`, we conducted total 1,071 trials (*i.e.*, 357 functions with 3 trials each in evaluation). Among these, only 66 trials for MSG and 77 trials for MSG_{AIO} triggered specification-coverage feedback. This is primarily due to the fact that the Move Prover helps MSG generate sufficiently complete specifications to pass verification, which is the prerequisite for specification coverage computation. As there are a non-trivial number of unique clauses (specifically, unique `ensures` clauses) that MSG generates as shown in Table III, it's not surprising that generated specifications are comprehensive enough, which leads to few observations of specification coverage feedback.

For most of the observed cases, the feedback was triggered by the deletion of an isolated assertion statement. Since the LLM can capture the intention behind such assertions very well, as it's a common feature for most programming languages, the LLM typically already generates specifications that inherently cover these cases.

Thus, while specification coverage did not lead to noticeable improvements in our current evaluation, its presence and traces confirm that MSG already produces high-quality, comprehensive specifications. Moreover, the use of specification coverage to evaluate comprehensiveness of specifications is *fully-automatic* without the need to manually compare generated specifications with expert-written ones as we conduct in §V-A. The results answer **RQ5**: non-trivial specification coverage is not observed for *verifiable* specifications generated by MSG and MSG_{AIO}: the generated verifiable specifications are comprehensive enough when they pass the Move Prover.

VI. Discussion

A. Adaptation to Other Verification Languages

MSG's *language-agnostic* approach combines compositional generation with an automated specification-coverage metric to ensure comprehensiveness.

Compositional Generation. Compositional generation scales by isolating orthogonal property groups such as post-conditions, aborts (which serve similar purposes to pre-conditions[1] in other languages), global-state updates in Move for blockchains. The approach can be adapted to other languages (*e.g.*, Dafny [45], Why3 [46] for general programs, or solc-verify [47], Certora [48] for Solidity [49] used to verify Ethereum [50]

[1]actual pre-conditions specified by `require` appear in just $\approx 2\%$ of Aptos functions—typically to assume external assumptions (*e.g.*., "blockchain is running").

smart contracts) by grouping properties around domain-specific concerns.

Certora. For example, a straightforward adoption is verifying contracts in Ethereum with Certora Prover. Due to the different natures of Move/Ethereum and Move Prover/Certora Prover, the specific syntax/implementation might be different; however, these blockchain-specific concerns are shared between both platforms (*i.e*., abortion/revert, global-state changes). To be more concrete, in Certora Verification Language (CVL), we could combine `require` (pre-condition), `fun@withrevert` annotation (asks the prover to set `lastReverted` based on whether the function `fun` reverted), and `assert lastReverted` to test whether a function will abort under certain conditions, which corresponds to `aborts_if` in MSL. Testing global-state updates requires more scaffolding due to the lack of `modifies` clauses; this can be addressed by using ghost variables [51] to record additional variable updates during verification, after which the modification of final states can be checked with `assert` by comparing with original states. As long as the verification languages have enough constructs (`withrevert`, and ghost variables in Certora, for example), the adaptation will be easy and straightforward for other blockchain smart contract verifiers.

For other domains (*e.g*., operating systems, network protocols), it requires the experts to identify distinct groups of domain-specific concerns that could be verified independently.

Specification Coverage. Specification coverage only requires language-specific mutators for random deletion used in §III-C.

B. Threats to Validity: Data Exposure to LLM

LLMs show strong generalization across domains and a solid grasp of Move, despite its limited training data. However, our evaluation shows that baseline MSG$_{\text{AIO-naive}}$ with advanced o3-mini model proves only 38.3% of specifications without advanced context engineering and an agentic approach. Thus, while familiarity with mainstream languages aids general capabilities, it does not necessarily ensure strong performance in niche languages like Move, especially for specification generation without proper system design.

VII. Related Works: LLM-based Specification

As briefly discussed in §I, MSG is built upon the same insights highlighted in the recent LLM for specification generation works [11], [12], [13], [14], [15], [16], [17], [18], [23], [24], [25], [26]: heuristic or rule-based specifications synthesis templates [21], [22] can hardly match the code comprehension and reasoning capabilities of modern LLMs. And yet, MSG provides more insights on generality of this research direction, while also independently rediscovering and validating parts of previous works in low-resource languages such as Move: ① LLM for specification generation has the potential to generalize to even non-mainstream programing languages (e.g., Move); ② an LLM-based specification generator should leverage features in the underlying specification language at the design stage (e.g., a separation of concerns of different clauses in MSL enables MSG to generate specifications in a modular and agentic way); ③ leveraging feedback, especially counterexamples [26], from the verification toolchain (not only the compiler) can significantly improve the quality of generated specifications; ④ simple metrics such as specification coverage can be used to gauge the comprehensiveness of generated specifications; ⑤ scaffolding utilities (e.g., static analysis tools or function inliners) can be helpful to scoping the context [27] for LLMs to generate specifications, but the engineering effort to build such utilities should not be underestimated. We hope the experience and insights from MSG can inspire future works in specification generation and LLM4FM research in general. We highlight key differences with two particularly relevant works:

AutoVerus. AutoVerus [23] uses specialized repair agents, each addressing a specific prover error, based on the idea that specification repairing can be decomposed into subproblems aligned with common failure modes. They implemented 10 such agents with tailored prompts. Similarly, MSG leverages the compositional structure of the Move Specification Language, assigning different prompts to different specification parts in an integrated way. The two approaches are orthogonal and could be fruitfully combined. Notably, besides specialized prompts for different classes of clauses, MSG already uses specialized prompts for some prover errors, but does not (yet) isolate them into separate agents as AutoVerus does.

Testcase-based Evaluation of Specifications. Testcase-based evaluation used in [25] helps gauge specification comprehensiveness through differential testing, where both buggy and correct implementations are checked against the specification to see whether it captures the intended behavior. However, this approach relies on the availability of such paired implementations, which are often scarce in emerging languages like Move. Our specification coverage metric is implementation-agnostic, making it particularly useful where suitable code versions are lacking. It assesses specification quality via code mutation (random deletion) and verifiability. Both approaches are complementary, and ideally should be used together for a thorough evaluation.

VIII. Conclusion

In this paper, we present MSG, a fully automated, end-to-end Move specification generation agent for Move smart contracts. Building on an agentic design that leverages the compositional characteristics of the Move Specification Language, MSG integrates several scaffolding utilities, including static analysis, best-effort function inlining for a well-scoped LLM conversation context, Move Prover feedback as a verifier-in-the-loop oracle, and a simple metric—specification coverage—to automatically evaluate the comprehensiveness of the generated specifications during the generation process. Evaluation results demonstrate that MSG not only produces verifiable expert-level specifications but also generates unique specifications, proving its capability to ease the burden of formal verification for Move smart contracts.

Acknowledgment

We thank the anonymous reviewers for their insightful feedback. This research was supported by Sui Foundation, Coya Innovation, ONR under grant N00014-23-1-2095, NSERC (RGPIN-2022-03325), and NCC (2024-1488).

References

[1] Concourse Open Community, "DeFi Pulse," https://defipulse.com/, 2022, accessed: Mar 1, 2025.

[2] CryptoSec Group, "Documented timeline of defi exploits," https://cryptosec.info/defi-hacks/, 2022, accessed: Mar 1, 2025.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[4] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[5] M. Felderer, D. Gurov, M. Huisman, B. Lisper, and R. Schlick, "Formal methods in industrial practice - bridging the gap (track summary)," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Rhodes, Greece, Oct. 2018.

[6] A. Ferrari and M. H. T. Beek, "Formal methods in railways: A systematic mapping study," *ACM Computing Surveys*, vol. 55, no. 4, Nov. 2022.

[7] M. Gleirscher and D. Marmsoler, "Formal methods in dependable systems engineering: A survey of professionals from europe and north america," *Empirical Software Engineering*, vol. 25, no. 6, 2020.

[8] T. Kulik, B. Dongol, P. G. Larsen, H. D. Macedo, S. Schneider, P. W. V. Tran-Jørgensen, and J. Woodcock, "A survey of practical formal methods for security," *Formal Aspects of Computing*, vol. 34, no. 1, Jul. 2022.

[9] Certota Team, "Certora securiy report - mayan fastmctp," https://www.certora.com/reports/mayan-smart-contract-security-report, 2025, accessed: May, 2025.

[10] ——, "Certora securiy report - https://www.certora.com/reports/texture-finance-security-report," https://www.certora.com/reports/mayan-smart-contract-security-report, 2025, accessed: May, 2025.

[11] C. Wen, J. Cao, J. Su, Z. Xu, S. Qin, M. He, H. Li, S.-C. Cheung, and C. Tian, "Enchanting program specification synthesis by large language models using static analysis and program verification," in *International Conference on Computer Aided Verification*. Springer, 2024, pp. 302–328.

[12] L. Ma, S. Liu, Y. Li, X. Xie, and L. Bu, " SpecGen: Automated Generation of Formal Program Specifications via Large Language Models ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 666–666. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00129

[13] D. Xie, B. Yoo, N. Jiang, M. Kim, L. Tan, X. Zhang, and J. S. Lee, "How effective are large language models in generating software specifications," *arXiv preprint arXiv:2306.03324*, 2025.

[14] S. Li, J. Jiang, T. Zhao, and J. Shen, "Osvbench: Benchmarking llms on specification generation tasks for operating system verification," 2025. [Online]. Available: https://arxiv.org/abs/2504.20964

[15] J. Pascoal Faria, E. Trigo, and R. Abreu, "Automatic generation of loop invariants in dafny with large language models," in *International Conference on Fundamentals of Software Engineering*. Springer, 2025, pp. 138–154.

[16] R. Liu, G. Li, M. Chen, L.-I. Wu, and J. Ke, "Enhancing automated loop invariant generation for complex programs with large language models," *arXiv preprint arXiv:2412.10483*, 2024.

[17] A. Kamath, A. Senthilnathan, S. Chakraborty, P. Deligiannis, S. K. Lahiri, A. Lal, A. Rastogi, S. Roy, and R. Sharma, "Finding inductive loop invariants using large language models. corr abs/2311.07948 (2023)," 2023.

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

[19] Y. Liu, Y. Xue, D. Wu, Y. Sun, Y. Li, M. Shi, and Y. Liu, "Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation," *arXiv preprint arXiv:2405.02580*, 2024.

[20] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[21] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *Formal Methods Europe (FME 2001): Formal Methods for Increasing Software Productivity*, ser. Lecture Notes in Computer Science, vol. 2021. Springer, 2001, pp. 500–517.

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

[23] C. Yang, X. Li, M. R. H. Misu, J. Yao, W. Cui, Y. Gong, C. Hawblitzel, S. Lahiri, J. R. Lorch, S. Lu *et al.*, "Autoverus: Automated proof generation for rust code," *arXiv preprint arXiv:2409.13082*, 2024.

[24] T. Chen, S. Lu, S. Lu, Y. Gong, C. Yang, X. Li, M. R. H. Misu, H. Yu, N. Duan, P. Cheng, F. Yang, S. K. Lahiri, T. Xie, and L. Zhou, "Automated proof generation for rust code via self-evolution," 2025. [Online]. Available: https://arxiv.org/abs/2410.15756

[25] S. K. Lahirie, "Evaluating llm-driven user-intent formalization for verification-aware languages," in *2024 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2024, pp. 142–147.

[26] A. Kamath, N. Mohammed, A. Senthilnathan, S. Chakraborty, P. Deligiannis, S. K. Lahiri, A. Lal, A. Rastogi, S. Roy, and R. Sharma, "Leveraging llms for program verification," in *2024 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2024, pp. 107–118.

[27] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 693–31 715.

[28] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, Rain, D. Russi, S. Sezer, T. Zakian, and R. Zhou, "Move: A language with programmable resources," https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf, 2020, accessed: 2025-03-01.

[29] D. Association, "Diem blockchain: A scalable and secure blockchain for the digital economy," https://www.diem.com/, 2019, originally launched as Libra. Accessed: April 1, 2025.

[30] Aptos Foundation, "Aptos," https://aptoslabs.com, 2022, accessed: Feb, 2025.

[31] Sui Foundation, "Sui," https://sui.io, 2022, accessed: Feb, 2025.

[32] Movement Network Foundation, "Movement," https://www.movementnetwork.xyz, 2025, accessed: May, 2025.

[33] D. Dill, W. Grieskamp, J. Park, S. Qadeer, M. Xu, and E. Zhong, "Fast and reliable formal verification of smart contracts with the move prover," in *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Munich, Germany, Apr. 2022.

[34] C. Tinelli and C. Barrett, "Satisfiability modulo theories," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 305–343.

[35] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[36] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli *et al.*, "cvc5: A versatile and industrial-strength smt solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 415–442.

[37] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[38] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," Microsoft Research, Technical Report MSR-TR-2005-70, 2005, https://www.microsoft.com/en-us/research/publication/boogie-a-modular-reusable-verifier-for-object-oriented-programs/.

[39] N. Shinn, F. Cassano, A. Gopinath, K. R. Narasimhan, and S. Yao, "Reflexion: language agents with verbal reinforcement learning," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=vAElhFcKW6

[40] R. Ji and M. Xu, " Finding Specification Blind Spots via Fuzz Testing ," in *2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp.

2708–2725. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.10179438

[41] Aptos Foundation, "Move Specification Language (Abstract Specification)," https://aptos.dev/build/smart-contracts/prover/spec-lang#abstract-specifications, 2024, accessed: 2025-08-25.

[42] Wikipedia contributors, "Uninterpreted function," https://en.wikipedia.org/wiki/Uninterpreted_function, 2023, [Online; accessed April 1, 2025].

[43] G. D. Plotkin, "A note on inductive generalization," in *Machine Intelligence 5*, B. Meltzer and D. Michie, Eds. Edinburgh: Edinburgh University Press, 1970, pp. 153–163.

[44] J. Venn, "I. on the diagrammatic and mechanical representation of propositions and reasonings," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 10, no. 59, pp. 1–18, 1880.

[45] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, ser. Lecture Notes in Computer Science, vol. 6355. Springer, 2010, pp. 348–370. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-17511-4_20

[46] J.-C. Filliâtre and A. Paskevich, "Why3—where programs meet provers," in *European symposium on programming*. Springer, 2013, pp. 125–128.

[47] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Working conference on verified software: theories, tools, and experiments*. Springer, 2019, pp. 161–179.

[48] Certora, "Certora: Formal verification for smart contracts," https://www.certora.com/, accessed: 2025-08-19.

[49] Solidity Contributors, "Solidity: A smart contract-oriented programming language," https://soliditylang.org/, 2014, version accessed: April 1, 2025.

[50] V. Buterin *et al.*, "Ethereum: A decentralized platform for applications," https://ethereum.org/, 2015, accessed: April 1, 2025.

[51] Certora Ltd., "Ghosts in Certora verification language," https://docs.certora.com/en/latest/docs/cvl/ghosts.html, 2024, accessed: 2025-08-19.

Appendix

A. Abstract Specification Example

```
fun mod_exp(base: u64, exponent: u64, modulus: u64): u64 {
  if modulus == 1 {
    return 0;
  }

  var result: u64 = 1;
  var b: u64 = base % modulus;
  var e: u64 = exponent;

  // Binary exponentiation
  while e > 0 {
    if e % 2 == 1 {
      result = (result * b) % modulus;
    }
    b = (b * b) % modulus;
    e = e / 2;
  }

  return result;
}

/// Uninterpreted function representing modular exponentiation
spec fun mod_exp_spec(base: u64, exponent: u64, modulus: u64): u64;

spec mod_exp {
  ensures [abstract] result == mod_exp_spec(base, exponent, modulus);
}
```

Fig. 5: Abstract specification example for modular exponentiation function.

Figure 5 shows an abstract specification for a complex mathematical function—modular exponentiation with binary exponentiation—which is beyond the capabilities of the Move Prover due to limitations in the underlying SMT solvers. By specifying the behavior abstractly using "[abstract]", we can refer to the desired mathematical result without providing a fully-verified implementation. This approach also allows developers to use a provisional specification as a placeholder until a concrete, prover-friendly version can be written, either through approximation or with further tool improvements. We provide a real example generated by MSG for Aptos in the case studies (§B).

B. Case Studies

To provide concrete insights into MSG's specification generation capabilities, we present a detailed analysis of selected examples that illustrate the system's operational mechanisms and demonstrate its practical effectiveness. Specifically, we examine two representative instances where Move Prover feedback enables iterative refinement of initially imperfect specifications, along with a third case that exemplifies how MSG can autonomously draft sophisticated specifications for mathematically complex functions, thereby providing valuable scaffolding for subsequent expert review and refinement.

```
public fun enable_ungated_transfer(
  ref: &TransferRef) acquires ObjectCore {
  assert!(!exists<Untransferable>(ref.self),
    error::permission_denied(
      EOBJECT_NOT_TRANSFERRABLE));

  let object = borrow_global_mut<ObjectCore>
                                    (ref.self);
  object.allow_ungated_transfer = true;
}
```

(a) Move Function

```
spec enable_ungated_transfer {
  modifies global<ObjectCore>(ref.self);

  aborts_if exists<Untransferable>(ref.self);
  // missing!

  let post flag =global<ObjectCore>(ref.self)
    .allow_ungated_transfer;
  ensures flag == true;
}
```

(b) Partial specification missing an abort clause

```
spec enable_ungated_transfer {
  modifies global<ObjectCore>(ref.self);

  aborts_if exists<Untransferable>(ref.self);
  aborts_if !exists<ObjectCore>(ref.self);// added

  let post flag = global<ObjectCore>(ref.self)
    .allow_ungated_transfer;
  ensures flag == true;
}
```

(c) Corrected full specification

Fig. 6: Demonstration of iterative specification completeness enhancement through systematic incorporation of Move Prover diagnostics.

Iterative Refinement of Missing `aborts_if` Clauses. The empirical results presented in Table II demonstrate the substantial utility of Move Prover feedback in facilitating specification correctness through systematic error detection and remediation. Figure 6 presents a particularly clear example where the synergistic interaction between LLM reasoning and formal verification feedback enables the identification and subsequent correction of an omitted `abort_if` clause. The function under consideration exhibits abort behavior when the requisite `ObjectCore` resource is absent from the blockchain's global storage at the address specified by `ref.self`. In the initial generation phase, MSG produces a specification (Figure 6b) that, while syntactically well-formed, omits the crucial `aborts_if` clause corresponding to this failure condition, consequently rendering the specification unverifiable by the Move Prover. The diagnostic output generated by the Move Prover (with `o` representing `object`, abbreviated for space) illustrates this deficiency:

```
error:abort not covered by any `aborts_if` clauses
|
|let o = borrow_global_mut<ObjectCore>(ref.self);
|        -- abort happened here with execution failure
```

This diagnostic precisely identifies the unhandled abort condition that was inadvertently omitted from the initial specification. Through the incorporation of this formal verification feedback into subsequent generation iterations, the LLM demonstrates remarkable capacity for error recognition and specification refinement, ultimately producing the corrected specification (Figure 6c) that comprehensively addresses the identified deficiency.

```
1  const INIT_GUID_CREATION_NUM: u64 = 0x4000000000000;
2  fun create_object(owner_address: address): ConstructorRef {
3    let unique_address =
4      transaction_context::generate_auid_address();
5    create_object_internal(owner_address, unique_address, true)}
6
7  fun create_object_internal(creator_address: address,
8    object: address, can_delete: bool): ConstructorRef {
9    let object_signer = create_signer(object);
10   let guid_creation_num = INIT_GUID_CREATION_NUM;
11   let transfer_events_guid =
12       guid::create(object, &mut guid_creation_num);
13
14   move_to(&object_signer,
15     ObjectCore {
16       guid_creation_num, owner: creator_address,
17       allow_ungated_transfer: true,
18       transfer_events: event::new_event_handle(
19         transfer_events_guid)});
20   ConstructorRef { self: object, can_delete } }
21
22 /// guid.move
23 public(friend) fun create(
24   addr: address, creation_num_ref: &mut u64): GUID {
25   let creation_num = *creation_num_ref;
26   *creation_num_ref = creation_num + 1;
27   GUID { id: ID { creation_num, addr } } }
28
29 /// wrong
30 spec create_object {
31   ensures global<ObjectCore>(result.self).guid_creation_num
32   == INIT_GUID_CREATION_NUM; }
33
34 ///  The guid_creation_num in the ObjectCore resource
35 ///  equals INIT_GUID_CREATION_NUM + 1,
36 ///  reflecting the increment performed during GUID creation.
37 spec create_object {
38   ensures global<ObjectCore>(result.self).guid_creation_num
39   == INIT_GUID_CREATION_NUM + 1; }
```

Fig. 7: Example of autonomous postcondition refinement through formal verification feedback.

Automated Correction of Erroneous ensures Clauses. Figure 7 exemplifies the sophisticated self-correction capabilities of MSG when confronted with semantically incorrect postcondition specifications, demonstrating the system's ability to leverage formal verification feedback for iterative specification refinement. The `create_object` function implements a complex protocol for object instantiation within the blockchain's global storage namespace, ensuring address uniqueness through careful resource management. The core functionality is encapsulated within `create_object_internal` (Line 7), which performs a multi-stage creation process. Initially, the function constructs a signer object—a cryptographic primitive essential for authenticated modifications to account-specific global storage. Subsequently, at Line 12, the function invokes `guid::create` to generate a globally unique identifier (`transfer_events_guid`), passing a *mutable reference* to `guid_creation_num` initialized with the constant `INIT_GUID_CREATION_NUM`. Crucially, `guid::create` exhibits side-effect behavior: it returns the original value of `guid_creation_num` while simultaneously incrementing the referenced value, thereby ensuring subsequent GUID uniqueness. The function completes by creating an `ObjectCore` object parameterized by the generated GUID, returning a `ConstructorRef` handle for further manipulation. The figure presents two specifications generated by MSG (with extraneous elements elided for clarity), which illuminate the system's iterative refinement process. The initial specification (Line 30) exhibits a subtle but critical semantic flaw: it fails to properly model the side-effect induced by the mutable reference semantics of `guid_creation_num`. This oversight manifests in the Move Prover's diagnostic output (presenting only the pertinent excerpts, with line numbers adjusted to correspond to Figure 7):

```
error: post-condition does not hold
|
|    ensures global<ObjectCore>(result.self)
|            .guid_creation_num == INIT_GUID_CREATION_NUM
| =  at  create_object_internal (Line 12)
| =      guid_creation_num = 1125899906842624
| =  at  guid.move:create (Line 26)
| =      guid_creation_num = 1125899906842625
```

This diagnostic clearly identifies the semantic inconsistency: the difference between `guid_creation_num`'s value at Line 12 (representing the pre-increment state) and Line 26 (reflecting the post-increment state), with the latter serving as the parameter for `ObjectCore` construction. Through MSG's feedback incorporation mechanism, this verification failure is transmitted to the LLM, which demonstrates remarkable capacity for semantic reasoning by identifying the root cause and subsequently generating a corrected specification (Line 37) that accurately captures the increment semantics through an appropriately formulated `ensures` clause, as corroborated by the accompanying explanatory annotations.

These examples of autonomous specification refinement clearly demonstrate the LLM's sophisticated capacity for formal reasoning and error correction, establishing a feedback-driven methodology that systematically enhances specification accuracy, as empirically validated by the quantitative results in Table II.

```
public fun exp(x: FixedPoint64): FixedPoint64 {
  let raw_value = (fixed_point64::get_raw_value(x) as u256);
  fixed_point64::create_from_raw_value((exp_raw(raw_value) as u128))
}

fun exp_raw(x: u256): u256 {
  // exp(x / 2^64) = 2^(x / (2^64 * ln(2))) =
  // 2^(floor(x / (2^64 * ln(2))) + frac(x / (2^64 * ln(2))))
  let shift_long = x / LN2;
  assert!(shift_long <= 63, std::error::invalid_state(EOVERFLOW_EXP));
  let shift = (shift_long as u8);
  let remainder = x % LN2;
  // At this point we want to calculate 2^(remainder / ln2) << shift
  // ln2 = 580 * 22045359733108027
  let bigfactor = 22045359733108027;
  let exponent = remainder / bigfactor;
  let x = remainder % bigfactor;
  // 2^(remainder / ln2) = (2^(1/580))^exponent * exp(x / 2^64)
  let roottwo = 18468802611690918839; //  2^(1/580)
  // 2^(1/580) = roottwo(1 - eps),
  // so the number we seek is roottwo^exponent (1 - eps * exponent)
  let power = pow_raw(roottwo, (exponent as u128)); // loop
  let eps_correction = 219071715585908898;
  power = power - ((power * eps_correction * exponent) >> 128);
  // x is fixed point number smaller than
  // bigfactor/2^64 < 0.0011 so we need only 5 tayler steps
  // to get the 15 digits of precission
  let taylor1 = (power * x) >> (64 - shift);
  let taylor2 = (taylor1 * x) >> 64; let taylor3 = (taylor2 * x) >> 64;
  let taylor4 = (taylor3 * x) >> 64; let taylor5 = (taylor4 * x) >> 64;
  let taylor6 = (taylor5 * x) >> 64;
  (power << shift) + taylor1 + taylor2 / 2
  + taylor3 / 6 + taylor4 / 24 + taylor5 / 120 + taylor6 / 720
}
```

(a) Move implementation for natural exponential computation (e^x)

```
/// Specification for function "exp".
/// It computes the exponential of the fixed-point number x.
/// The function aborts if the raw value of x is too high
/// to safely compute the exponential.
/// On success, the result equals the fixed-point number
/// created from the abstract exponential
/// computed by spec_exp, applied to the raw value of x.
spec exp {
  // Retrieve the raw value of x from its FixedPoint64 representation.
  let raw = fixed_point64::get_raw_value(x);

  // Abort if the exponential cannot be safely computed on raw.
  // The abstract function spec_can_exp encapsulates the overflow check.
  aborts_if [abstract] !spec_can_exp(raw);

  // On success, the result is the fixed-point number
  // created from the abstract exponential.
  ensures [abstract] result ==
    fixed_point64::create_from_raw_value(spec_exp(raw));
}

/// Abstract specification function
/// for computing the exponential on the raw value.
/// It maps the raw u128 value (extracted from x) to
/// the raw result that is then wrapped
/// as a FixedPoint64 by create_from_raw_value.
spec fun spec_exp(raw: u128): u128;

/// Abstract specification function
/// to decide whether the exponential can be computed.
/// Returns true if computing exp on the given raw value
/// is safe (i.e., no overflow occurs),
/// and false otherwise.
spec fun spec_can_exp(raw: u128): bool;
```

(b) Associated formal specification

Fig. 8: Example of compositional and abstract specification generation by MSG. The `exp` function implements natural exponential computation (e^x) through Taylor series approximation.

Compositional and Abstract Specification Generation. Rather than constraining specification generation to monolithic constructs, MSG incorporates teaching examples that demonstrate modular and abstract specification methodologies, thereby enabling the generation of more sophisticated and maintainable formal artifacts. Figure 8 presents a representative example: a function implementing natural exponential computation (e^x) through Taylor series expansion within the aptos-stdlib framework. The `exp` function exhibits a layered architectural pattern, delegating computational responsibilities to `exp_raw` for high-precision arithmetic operations on `u256` operands (256-bit unsigned integers), subsequently transforming results into `FixedPoint64` representation—a hybrid numeric type comprising distinct 64-bit integer and fractional components. Within `exp_raw`, the implementation performs a complex sequence of mathematical operations encompassing multiplication, division, modular arithmetic, and bitwise manipulation. While the underlying SMT solvers supporting Move Prover possess robust capabilities for linear integer arithmetic, they encounter fundamental limitations when confronted with non-linear arithmetic constructs (exemplified by variable-to-variable multiplication), which constitute inherent decision procedure bottlenecks. Consequently, rather than attempting to generate concrete specifications for `exp` (which would likely exceed current SMT solver capabilities), MSG adopts a more nuanced approach, producing the draft specification architecture illustrated in Figure 8b. This specification employs a compositional structure comprising one primary `spec` block augmented by two auxiliary `spec fun` definitions. `spec fun` constructs function as specification-level abstractions—analogous to helper functions that undergo expansion within their associated specification contexts, thereby facilitating modular specification design and enhanced maintainability. This particular example shows an interesting feature: MSG generates two `spec fun` entities (`spec_exp` and `spec_can_exp`) accompanied by comprehensive descriptive annotations explaining their intended purposes, yet deliberately omitting concrete definitional content. Such constructs are formally characterized as *uninterpreted functions* [42]. Uninterpreted functions serve as semantic placeholders that abstract away implementation complexities while preserving interface contracts—conceptually analogous to mock objects in software testing methodologies. Within the primary `spec exp` block, MSG employs `aborts_if [abstract]` and `ensures [abstract]` annotations, which explicitly signal to the Move Prover that these specifications represent abstract behavioral contracts rather than concrete verification obligations. This example clearly shows MSG's capacity for generating sophisticated modular and abstract specifications that serve as high-quality drafts, providing substantive semantic scaffolding for subsequent expert elaboration and formal completion.

C. *Evaluation Results of* MSG$_{AIO}$ *Variants for Weaker Models GPT-4o and GPT-4o-mini*

	MSG$_{\text{AIO}}$			MSG$^{-}_{\text{AIO}}$			MSG$_{\text{AIO-naive}}$		
	o3-mini	GPT-4o	GPT-4o-mini	o3-mini	GPT-4o	GPT-4o-mini	o3-mini	GPT-4o	GPT-4o-mini
1st	132	103	84	140	109	92	106	83	57
2nd	33	18	22	6	8	9	18	11	4
3rd	14	7	12	0	6	1	7	2	1
4th	8	6	3	0	6	1	7	2	1
5th	4	7	1	1	3	2	3	2	2
Fail	54	95	221	77	70	240	220	254	291
Success	191	141	122	147	132	104	137	103	66
$> 1^{st}$	59	38	38	7	23	12	31	20	9
Abstract	112	121	14	133	155	13	0	0	0
$\frac{\text{Success}}{Total}$	**53.5%**	39.5%	34.2%	41.1%	37.0%	29.1%	38.3%	28.8%	18.4%
$\frac{1^{st}}{\text{Success}}$	69.1%	73.0%	68.9%	95.2%	82.6%	88.5%	77.4%	80.6%	86.4%
$\frac{>1^{st}}{\text{Success}}$	**30.9%**	**27.0%**	**31.1%**	4.8%	17.4%	11.5%	22.6%	19.4%	13.6%
$\frac{\text{Success}}{\text{Success+Abstract}}$	63%	53.9%	89.7%	52.5%	46%	88.9%	100%	100%	100%

TABLE VI: Result and round detail for different variants of MSG$_{\text{AIO}}$: MSG$^{-}_{\text{AIO}}$, MSG$_{\text{AIO-naive}}$. 1st to 5th means the earliest round that MSG$_{\text{AIO}}$ could generate the correct specification. The center rows show the summary of the results. "Fail" means that the MSG$_{\text{AIO}}$ cannot generate the specification within the five rounds. "Success" is the summation of 1st to 5th, which means that the MSG$_{\text{AIO}}$ can generate the specification within five rounds. "$> 1^{st}$" is the summation of 2nd to 5th, which means that the MSG$_{\text{AIO}}$ can generate the specification within five rounds but not in the first round (i.e. LLM cannot one-shot). "Abstract" is the number of generated abstract specifications. The following 3 rows are the successful rates of MSG, one-shot rates, and non-one-shot rates. The last row is the percentage of the concrete (non-abstract) successful generated specifications.

Models	o3-mini	GPT-4o	GPT-4o-mini
1st	132	109	93
2nd	35	25	22
3rd	15	6	9
4th	7	2	3
5th	4	7	3
Fail	47	81	214
Success	193	149	130
$> 1^{st}$	61	40	37
Abstract	117	127	13
$\frac{\text{Success}}{Total}$	**54.1%**	41.7%	36.4%
$\frac{1^{st}}{\text{Success}}$	68.4%	73.2%	71.6%
$\frac{>1^{st}}{\text{Success}}$	31.6%	26.8%	28.4%
$\frac{\text{Success}}{\text{Success+Abstract}}$	62.3%	54.0%	91.0%

TABLE VII: Result and round detail of MSG$_{\text{AIO-inline}}$. Please refer to Table II for the definition of each column.