Research Report: Not All Move Specifications Are Created Equal

A Case Study on the Formally Verified Diem Payment Network

Meng Xu
University of Waterloo

***Abstract*—Software developers who are newly introduced to formal verification often have subtle but distinct interpretations on the roles of specifications (specs). Unsurprisingly, these different interpretations affect the types of specs developers tend to write, which, without coordination, can lead to fragmented assurance guarantee and ultimately impairs the overall effectiveness of the entire formal verification effort.**

This paper is an experience report on rolling out a formal verification system in a smart contract setting (where correctness matters financially). In this process, we discover three popular views about what "a spec is operationally" among experienced industrial developers yet novice in formal methods: 1) specs are contracts between implementation and functionalities conveyed to end users; 2) specs are extensions to the type system; and 3) specs are definitions of high-level state machines.

While which interpretation is closer to the real purpose(s) of specs is a judgement call, one important view is missing: some specs are *abstracting specs* that lock in requirements or intention (hence the more the better), some specs are *proof assistance* that facilitates the refinement proof between implementation and abstracting specs and hence, should be written on an as-need basis; while other specs are *denotational specs* and do not add assurance from a formal methods perspective. Absence this distinction, it is easy to fall into a trap in formal verification where specs accumulate but little assurance is added holistically.

1. Introduction

Smart contracts manage and operate on data stored over a blockchain network. Such data is not only critical to the functionalities of a smart contract, but can also be of substantial monetary value when representing crypto assets. Unfortunately, storing data with smart contracts does not mean the data is safe. In just the past few years, crypto assets worth millions of USD have been either locked in blockchain with no availability to anyone, or stolen by malicious parties, both due to subtle flaws in smart contracts [12], [13].

In light of these non-trivial financial losses, developing high-assurance (or even bug-free) smart contracts has emerged as an exciting field for both researchers and practitioners with formal verification background. In this context, the essence of formal verification is to answer this question: *does a smart contract satisfy all intended requirements?* This question can be further decomposed into four sub-problems:

① extracting explicit and implicit requirements extensively from all potential stakeholders;
② devising a set of specifications (specs) that formally and fully captures the requirements and nothing more;
③ proving whether a concrete implementation (i.e., the code of a smart contract) conforms to the specs; and
④ ensuring correctness of proofs produced in ③, i.e., trustworthy verification with the smallest possible trust.

While much of the research effort has been focusing on solving problems ① requirement engineering [29], [50], ③ (semi-)automated verification [2], [4], [14], [17], [41], [43], and ④ foundational correctness [1], [39], [52], problem ② remains largely unattended: how to develop specs? More precisely, **how to write or review specs in a way that maximizes return on investment (ROI)?**

In other words, the consequences of a low quality set of specs is exacerbated by the high costs of adopting formal methods for assurance. After all, formal verification is still an expensive and exotic approach due to the extra effort in writing, proving, and maintaining specs—a task that usually requires the support from a dedicated team of experts with years of training and practice (including PhD degrees) in formal methods [18], [19], [25], [28].

However, despite the high costs, more industries are willing to pursue this route, with an expectation that formal verification offers more assurance than other approaches (e.g., property-based testing [21], fuzzing [53], or symbolic execution [3]). This can be a dangerous assumption! While it may be true that formally verified code are in better quality, the assurance is not necessarily higher and the formally verified "stamp" should not be blindly trusted without a good understanding of what is really specified.

Take Figure 4 as a preview (details discussed in §3.1). The set of specs (4b) merely shadows the imperative implementation (4a) with a declarative/functional notation *without abstraction*. The code might be advertised as extensively specified with a 1:1 spec-to-code ratio and with formally verified "stamp", but the specs do not help distinguish the boundary between trusted oracle and untrusted implementation. In fact, the specs is still useful from an N-version programming [11] perspective, i.e., spec and code writers are unlikely to make the same mistake if they are indeed developed independently, but this is not assurance from formal methods.

One way to avoid falling into this false assurance trap is for the spec language to distinguish specs that abstract the software system from specs that don't (although they

might be useful in other ways). Auditors, regulators, or whomever scrutinizing the correctness of the program should only examine abstracting specs. Intuitively, if the system uses abstraction so that N lines of scrutinized spec constrains $10N$ lines of untrusted implementation, then it is a 10x return on the investment of writing or reviewing the spec. In short, **abstracting specs** helps to improve ROI in formal verification and should be promoted.

While the intuition of writing abstracting specs is trivial amongst formal methods experts, in this research report, we examine whether the same intuition is well-known in practice through a case study of the Move language and the Diem Payment Network (DPN) project [15]. In particular, we seek to answer the following research questions:

RQ1: Are all specs from the Move spec language abstracting? If not, what other purposes do they serve?

RQ2: Can a spec language be designed to distinguish abstracting specs and other types of specs?

RQ3: How to apply the findings to guide future development of specs at least in Move?

In short, DPN is a smart contract that aims to function as a full-fledged and versatile payment/banking system with capabilities of handling multiple currencies, account roles, and rules for transactions. DPN was initially developed at Meta and later open sourced in June 2019. The entire commit history of how specs co-evolve with the implementation is visible on GitHub [16]. At the time of writing, DPN has accumulated 11,813 lines of code, including 8,372 lines of specs, making a 7:5 code-to-spec ratio.

Through exemplary specs in DPN, we show that specs that do not abstract the software system can be either **proof assistance**, which should only be provided on an as-needed basis to overcome practical limitations of automated verification tools; or **denotational specs**, which only shadows implementation details and should be discouraged (see §A for reasons why these specs are termed denotational).

Following the co-evolution of code and spec in DPN yields some interesting observations that may shed some lights on how to (and how not to) design a spec language and formal verification system that distinguishes *abstracting specs*, *proof assistance*, and *denotational specs*:

❶ Not all specs in DPN are equally important in providing assurance. A core suite of specs ($\approx$ 800 LoC) guard the most critical safety properties (e.g., access control); a small portion of specs ($\approx$ 500 LoC) serve as proof assistance to the core specs; while others, mostly pre- and post-conditions for internal functions, merely shadow the implementation and do not contribute to formal proof.

❷ Consequently, despite being advertised as extensively specified in several occasions [17], [32], [51], the 7:5 ratio indicate neither completeness nor deficiencies of the spec. In fact, such a ratio is hardly an objective measurement on the usefulness of specs if the spec language does not distinguish abstracting specs and proof assistance.

❸ Specs in the early days of DPN are almost entirely denotational in nature and are strongly coupled with the implementation (e.g., spec and code change almost in lock-step). The introduction of *global invariants* (details in §5) changes the whole situation and is the key ingredient in the spec language that empowers it to differentiate abstracting specs and denotational specs. In fact, global invariants are fairly stable in DPN, indicating that they are closer to capture requirements and intention.

❹ Denotational specs can be useful in documentation and property-based testing. In fact, documentation in DPN is generated by MVP based on the specs [33] and the executable (quantifier-free) fragment of specs is used as properties in property-based testing [34].

In summary, we notice that "not all specs are abstracting the system" is trivially known in the formal methods community but can be lesser known outside the community. This research report makes the following contributions to promote the development of abstracting specs in Move community:

- A categorization of specs (abstracting spec, proof assistance, and denotational spec) in the Move language based on how they help boost assurance through verification.
- Highlight the importance that the spec language should be designed to distinguish specs in different categories and to promote the writing of abstracting specs.
- Showcase how to promote the development of abstracting specs with concrete advice on the Move language.

In the rest of this research report, we first introduce different types of specs available in DPN (§2) and dive into each spec type to illustrate how industrial developers, especially those without formal verification experience, perceive them (§4, §3, §5). Through this, we show that despite various interpretations about specs exist, one important view is missing: some specs are abstracting specs that lock in requirements or intention while others are denotational and should only serve as proof assistance on an as-need basis. Absent understanding that boundary, we show some anti-patterns that fell into the trap of false assurance, while interested readers may jump to §6 for a summary of how a spec language can help highlight this missing view.

2. A Primer on Move and Its Spec Language

DPN is developed in Move [7], [36], a programming language for writing safe smart contracts Move was initially developed for the Diem blockchain [15] but has recently transitioned into an open-source community-backed language and now powers several Layer-1 and Layer-2 blockchains [37]. The language features robust safety [42] and formal verification at its core through its home-grown verification tool Move Prover (MVP) [17]. MVP statically verifies the correctness of Move smart contracts modeled with the Move Specification Language (MSL) [35].

Language overview. Move is similar to Rust in look and feel and shares important features such as strong typing (i.e., no static or dynamic casts), a linear type system [24], and borrow semantics [26] which eliminates the complication on memory modeling and pointer aliasing tracking—all making formal verification easier on Move relative to C/C++. A detailed description about the memory model in Move can be found in §B. More importantly, Move builds-in an elegant

```
module 0x42::Demo {
  // constants
  const MIN_BALANCE: u64 = 5;
  const MAX_BALANCE: u64 = 10000;

  // user-defined data type
  struct Account has key {
    balance: u64,
  }

  // one entrypoint for a blockchain transaction
  public entry fun transfer(
    from: address, into: address, amount: u64
  ) acquires Account {
    // input validation
    assert!(from != into);

    // withdraw
    let bal = &mut borrow_global_mut<Account>(from).balance;
    assert!(*bal - MIN_BALANCE >= amount);
    *bal = *bal - amount;

    // deposit
    let bal = &mut borrow_global_mut<Account>(into).balance;
    assert!(MAX_BALANCE - *bal >= amount);
    *bal = *bal + amount;
  }

  // syntactic sugars to improve spec readability
  spec fun is_user(addr: address): bool {
    exists<Account>(addr)
  }

  spec fun balance(addr: address): num {
    global<Account>(addr).balance
  }
}
```

(a) code

```
spec 0x42::Demo {
  // unit spec on functions
  spec transfer {
    requires true;
    aborts_if from == into;
    aborts_if !is_user(from)
    aborts_if !is_user(into);
    aborts_if balance(from) - amount < MIN_BALANCE;
    aborts_if balance(into) + amount > MAX_BALANCE;
    ensures balance(from) == old(balance(from)) - amount;
    ensures balance(into) == old(balance(into)) + amount;
  }

  // invariants on data types (DI)
  spec Account { invariant balance > 0; }

  // invariants on global states (SI)
  invariant [state]
    forall addr: address where is_user(addr):
      MIN_BALANCE <= balance(addr) <= MAX_BALANCE;

  // invariants on global state transitions (TI)
  invariant [update]
    exists from: address: (
      is_user(from) && old(is_user(from)) &&
      old(balance(from)) > balance(from)
    ) ==> exists into: address: (
      is_user(into) && old(is_user(into)) &&
      balance(into) - old(balance(into)) ==
        old(balance(from)) - balance(from) &&
      forall rest: address
      where rest != from && rest != into:
        !old(is_user(rest)) ==> !is_user(rest) &&
        old(is_user(rest)) ==>
          (is_user(rest) && balance(rest) == old(balance(rest)))
    );
}
```

(b) spec[1]

Figure 1: A fully specified Move program for balance transfer between two accounts

data store model that boosts not only type safety but also the expressiveness of specs: *a persistent global state ($\mathbb{G}$)*.

In fact, the concept of $\mathbb{G}$ is not an invention of Move but a fundamental concept in virtually every blockchain that involves consensus-backed data storage (e.g., the ledger in Bitcoin [38]). A blockchain transaction calls into a smart contract which evolves $\mathbb{G}$ in its code via a fixed set of pre-defined primitives. Any runtime error results in aborting the transaction and leaves $\mathbb{G}$ unchanged (except for gas consumption). This is a programming model shared by, in our observation, all smart contract programming languages (e.g., Solidity, Vyper, Yul), including Move.

What is unique in Move is the organization of $\mathbb{G}$. In Move, $\mathbb{G}$ is a map:

$$\mathbb{G} : (\texttt{T}, \texttt{address}) \rightarrow \texttt{Optional<T>}$$

which is indexed by a composition of both a data type `T` defined in the program and an account address. The index is mapped to a value which holds either an instance of `T` or a `None` marker at the composite index. The complete semantics of $\mathbb{G}$ and the operations on $\mathbb{G}$ can be found in §C. Briefly, $\mathbb{G}$ can only be accessed via the following fixed set of primitives:

- `exists<T>(address)` $\rightarrow$ `boolean`: checks whether an instance of type `T` exists at the given address.
- `move_from<T>(address)` $\rightarrow$ `T`: grabs ownership of the instance of type `T` from the address (and hence $\mathbb{G}$) to the current calling context and leaves a `None` mark in $\mathbb{G}$. Aborts if the instance does not exist.
- `move_to<T>(address, T)`: transfers ownership of the instance of type `T` from the current calling context to the address in $\mathbb{G}$. Aborts if the instance already exists, as required by linear typing.
- `borrow_global<T>(address)` $\rightarrow$ `&T`: temporary borrows an immutable reference to the instance of type `T` at the address in $\mathbb{G}$ in the current calling context. Aborts if the instance does not exist. The type system ensures that the reference does not become dangling (e.g., via `move_from` or `move_to`) nor alias with any mutable reference.
- `borrow_global_mut<T>(address)` $\rightarrow$ `&mut T`: temporary borrows a mutable reference to the instance of type `T` at the address in $\mathbb{G}$ in the current calling context. Aborts if the instance does not exist. The type system ensures that the reference does not become dangling (e.g., via `move_from` or `move_to`) nor alias with any other reference.

A Move smart contract can be highly modular and developers are free to define their own data types, constants, functions and encapsulate or compose them (including others' libraries) in arbitrary ways as long as the code type checks. Each transaction must starts from an `entry` function. This

1. Numbers and variables of numerical type in spec are represented as mathematical integers (unlimited precision) instead of bit-vectors (fixed size). Hence in spec, there is neither arithmetic overflow nor underflow.

is similar to the `main` function in conventional settings with the exception that there might be multiple `entry` functions in the same Move program.

A taste of Move programming can be found in Figure 1. The example is a simple Move contract that defines an `Account` type to hold balances and offers an `entry` function `transfer` to manipulate data of this type. The smart contract also comes with a complete suite of spec shown side-by-side.

Unit spec on functions. Functional spec is arguably the most common form of specs in software verification (as it is essentially a Hoare triple) and certainly takes the majority in DPN. For these specs, the target scope is typically a single function only. Therefore, they are also referred to as unit spec in this research report, similar to the "unit" in unit tests. In Move, a unit spec contracts a set of pre- and post-conditions for a function f where each clause in the contract can be semantically summarized into three categories:

- `requires` p: f can be called only when condition p is met. This is essentially the precondition of f which has the semantics of being asserted at every call site of f as well as being assumed upon the entry of f.
- `aborts_if` p: f will abort when condition p is satisfied. This is the first half of the postcondition of f.
- `ensures` p: p will hold if f ever reaches the end of its execution (i.e., not hitting any aborting conditions). This is the second half of the postcondition of f.

In unit spec, a predicate p might refer to function parameters, return placeholders, and/or $\mathbb{G}$ but not the actual implementation (i.e., the function body). It is, however, possible to use constructs that do not have counterparts in executable semantics, such as universal ($\forall$) and existential ($\exists$) quantifiers over unbounded domains. Line 3-12 in Figure 1b is a unit spec sample for the `transfer` function.

Invariants on data types (DI). A DI expresses a well-formedness property of a user-defined algebraic data type (ADT). Any instance of this data type must satisfy this property at any time in execution, unless the instance is explicitly marked for a temporary mutation session (i.e., by a mutable borrow `&mut`). In the mutable borrowing case, the invariant is re-checked when the reference ceases to exist (which is known statically). Line 15 in Figure 1b is an illustrative DI which requires the account balance to be non-zero. Syntactically, a DI is specified as:

- `invariant` p: while p can be expressed in first-order logic, it can only refer to symbols visible in the ADT.

Invariants on global states (SI). The safety and correctness of any smart contract is largely about whether the blockchain persistent storage, represented by $\mathbb{G}$, remain in desired and consistent states *before and after any transaction*. Based on this insight, Move comes with first-class support on specifying what is a legal (or illegal) state for $\mathbb{G}$ in MSL. Syntactically, a SI can be expressed in the following format:

- `invariant [state]` p: where p can only refer to $\mathbb{G}$. p is often universally quantified by $\forall$ `a: address` to allow a predicate to apply on all accounts on the blockchain.

Line 18-20 in Figure 1b shows a SI which requires that all registered accounts on the blockchain must have a balance that falls within a legally required range. Note that this SI is slightly different from a DI on the `Account` type (e.g., line 15) as the DI is enforced on every account instance, even a temporary one that is not registered to $\mathbb{G}$ yet.

Invariants on global state transitions (TI). If we take the view that a Move smart contract is a concretization of an infinite number of communicating finite-state machines (CFSM) in developers' mind (or derived from business logic), then SI provides only half of the finite-state machine (FSM) spec — the nodes — and what is missing are the spec for edges, i.e., under what condition a state transition is allowed. This gap is filled by TI. Syntactically, a TI is specified as:

- `invariant [update]` p: where p can only refer to $\mathbb{G}$.

Line 23-36 in Figure 1b shows a TI. While seemly convoluted, the TI states that if an account balance is deducted, the exact same amount must be transferred to a different account; in the meanwhile, all other accounts not involved in this transaction should not be affected.

Verification process. Each function defined in the Move program is a verification unit, and a Move program is verified if all its defined functions are verified. To verify whether a single function `f` conforms to the spec, MVP instruments the unit spec (if any) associated with `f` as well as all relevant DI, SI, and TI specs as assumptions and assertions in the function and subsequently attempt to prove that none of the assertions can be triggered. Internally, MVP runs the weakest liberal precondition calculus [6] to reduce the instrumented function into a first-order formula and invokes satisfiability modulo theories (SMT) solvers to prove or disprove it. Interested readers may refer to MVP [17] for a detailed explanation on the verification process.

By default, unit specs are not compositional and verification of Move functions is not incremental. This means if function `f` invokes function `g` in its code, verification of function `f` will effectively inline the *implementation* of `g`, even if `g` has its associated unit spec. Users of MVP may request to inline the specs of `g` instead of its implementation in order to make verification compositional and incremental, but the request needs to be annotated explicitly.

3. Unit Spec as Implementation Contract

The purpose of unit spec is to unambiguously pin down the semantics for a function in some program logic such that current implementation and future changes to this function can be mechanically checked to ensure that they don't deviate from the semantics pinned-down by the spec (and the logic). Ideally, a unit spec is a mathematical way of stating a contract: *this function does X and only X*. As introduced in §2, X is expressed in the forms of pre- and post-conditions. However, the current Move unit spec is not expressive enough to fully specify the *"only X"* part, which will be discussed in §7.2.

Unit spec is arguably the most basic type of spec in software verification and, in certain spec systems, the only type

of spec supported (e.g., NatSpec [47] and SMTChecker [48] in Solidity). Unsurprisingly, unit spec contributes the largest number of LoC (over 90%) in DPN. However, in this report, we argue that unit specs alone offer very limited assurance from a formal method perspective, unless the specs:

I contract a user-facing interface, or
II confine implementation to a set of possibilities, or
III serve as an indispensable element in establishing a proof of refinement[2] from abstracting specs.

Consequently, merely counting LoC for specs or highlighting the spec-to-code ratio indicate neither success nor failure of the formal verification effort. A formal verification system should help distinguish unit specs that fall in different categories mentioned above from specs that are only denotational. Based on this, we can (1) promote the writing of specs in categories I and II (which increase assurance brought by formal verification), (2) check that specs in category III is only written when absolutely needed, and (3) discourage the writing of denotational specs. Counting LoC for specs in categories I and II will have practical values after the spec language can highlight this distinction.

3.1. Empirical Evidence from DPN

We support our claims with empirical examples from the DPN codebase. We acknowledge that the interpretation of these code snippets, especially on the usefulness of specs in these examples, is subjective, and we plan to substantiate our claims with more objective evidence, such as rigorously designed user studies, in follow-up work.

Category-I specs. The Move spec language (MSL) actually does not differentiate user-facing interfaces and internal functions. The Move language does. As briefly discussed in §2, all end-user facing interfaces are marked as "`public entry`" while library interfaces intended for external application developers are marked as "`public`". Therefore, unit specs for these external-facing functions can be considered as Category-I which provides assurance if the implementation is proved to be a refinement of the specs.

Figure 1 contains a Category-I spec which is the unit spec for the `transfer` function (line 3–12 in Figure 1b). The unit spec, together with the function type signature, contracts the semantics of the `transfer` function with end users. Ideally, this unit spec should not even be changed without notifying the users (or other stakeholders) in advance.

Category-II specs. Besides external-facing interfaces, developers might also want to make contracts with internal sub-teams, community contributors, or future team members on concrete implementation details. The unit specs supporting these contracts belong to Category-II. However, this might be the most ambiguous category in unit spec as it is usually a judgement call on whether is it worthwhile to pin down the semantics of an internal function with unit specs.

2. Program refinement is the verifiable transformation of an abstract (high-level) formal spec into a concrete (low-level) executable program. Proving the refinement is essentially producing a proof to show that the refined program conforms to the abstract spec in all aspects, i.e., all properties listed in spec hold in the program.

Unfortunately, in MSL, unit specs on internal or private functions share the same syntax with unit specs on public functions. Therefore, it is hard to quantify how many unit specs in DPN are truly contracting an internal function without interviewing the developers.

With that said, MSL does have constructs that can help identify Category-II specs: `[abstract]` and `[concrete]` annotations in the conditions of unit specs. Figure 2 presents an example on how these annotations can be used. In this example, the implementation (the `error_code` function in Figure 2a) is an encoder for error code that merges the category and reason identifier uniquely. The unit spec for this function, however, comes in two flavors:

- the `[concrete] ensures` clause is used when verifying the function body, and this essentially pins down the actual encoding mechanism ⟨left shift by 8 then sum⟩; while
- the `[abstract] ensures` clause is used in the callers of `error_code` in lieu of inlining the function body. This abstract property essentially describes the high-level requirement only via an axiom without dictating the actual encoding scheme. In other words, a different encoding scheme, ⟨left shift by 16 then sum⟩, will still refine the `[abstract] ensures` but not the `[concrete] ensures`.

The fact that both `[abstract]` and `[concrete]` exist is a strong indicator that the developer intends to pin down some or all implementation details in this function.

```
1 fun error_code(category: u8, reason: u8): u64 {
2   (category as u64) + (reason as u64 << 8)
3 }
```

(a) code

```
1 spec error_code {
2   ensures [concrete] result == category + (reason * 256);
3   ensures [abstract] result == spec_error(category, reason);
4 }
5
6 // axiomatized uninterpreted function
7 spec fun spec_error(category: u8, reason: u8): u64;
8
9 // declarative definition for the `spec_error` function
10 axiom forall c1: u8, c2: u8, r1: u8, r2: u8:
11   (c1 != c2 || r1 != r2) ==>
12     spec_error(c1, r1) != spec_error(c2, r2);
```

(b) spec

Figure 2: An illustration of implementation confinement spec

Category-III specs. Unit specs in this category are function summaries *with abstraction*, and the purpose of abstraction is to enable compositional verification of other specs, such as specs in Category-I and II or data, state, and transition invariants §2. Operationally, the unit spec allows the caller to embed an abstract semantics of a callee function with *intentional* omissions on irrelevant implementation details (e.g., whether sorting is done with quick sort or merge sort). This is not only useful but necessary when the implementation is too complex to inline for verification.

A typical usage of proof assistance is the axiomatization of cryptographic primitives, as shown in Figure 3. Without this modeling, it will be extremely hard if not impossible to verify any properties that involve `encrypt()` or `decrypt()`. This justifies the necessity of the specs in this example.

```
fun encrypt(key: AESKey, data: vector<u8>): vector<u8> {
  // ... implementation details redacted ...
}
spec encrypt {
  ensures [abstract] result == spec_encrypt(key, data);
}

fun decrypt(key: AESKey, data: vector<u8>): vector<u8> {
  // ... implementation details redacted ...
}
spec decrypt {
  ensures [abstract] result == spec_decrypt(key, data);
}

// axiomatized uninterpreted functions
spec fun spec_encrypt(key: AESKey, data: vector<u8>): vector<u8>;
spec fun spec_decrypt(key: AESKey, data: vector<u8>): vector<u8>;

axiom forall key: AESKey, data: vector<u8>:
  spec_decrypt(key, spec_encrypt(key, data)) == data;
```

Figure 3: An illustration of proof assistance specs

```
public fun foo(
    val: u64, addr: address
) {
  assert!(
    exists<S>(addr),
    errors::not_published
  );
  run_foo(
    val,
    borrow_global_mut<S>(addr)
  );
}

fun run_foo(
    val: u64, s: &mut S
) {
  assert!(
    val <= 100 && val >= s.val,
    errors::invalid_argument
  );
  s.val = val;
}
```

(a) Code

```
spec schema RunFooAbortsIf {
  val: u64;
  s: S;
  aborts_if val > 100;
  aborts_if val < s.val;
}
spec schema RunFooEnsures {
  val: u64;
  s: S;
  ensures s.val = val;
}
spec foo_internal {
  include RunFooAbortsIf;
  include RunFooEnsures;
}
spec foo {
  aborts_if !exists<S>(addr);
  include RunFooAbortsIf
          {s: global<S>(addr)};
  include RunFooEnsures
          {s: global<S>(addr)};
}
```

(b) Spec

Figure 4: An illustration of denotational specs

In contrast, Figure 2 which showcases a Category-II unit specs via the `[concrete]` clause, might not be an example of proof assistance specs, despite the fact that it also has an `[abstract]` clause. The reason is that the implementation (Figure 2a) is relatively simple; thus an automated verifier might still be able to verify properties involving `error_code()` without the abstraction this function.

Denotational specs that shadow implementation. From a formal verification perspective, this is the type of specs that *passively* shadow the imperative implementation with side effects (e.g., memory and the global storage $\mathbb{G}$) with a functional language in which every operation is pure (e.g., with explicit modeling for memory and $\mathbb{G}$).

Figure 4 presents an example of denotational unit specs that are almost 1:1 shadow of the code. In this example, the spec for the outer function `foo` is effectively stating that:

- `foo` will abort whenever callee `run_foo` aborts; and
- `foo` returns exactly what `run_foo` returns.

This is not contracting how `foo` should behave to external users. Instead, it simply exposes and dictates internal implementation details, i.e., `foo` calls an internal function `run_foo`. And yet, unit specs in like this create an "illusion" that the functions are comprehensively specified and verified.

And yet, with a large investment on specs, what is the assurance we get in return? Do unit specs actually serve the purpose of *contracting* with current and future developers and *actively challenging* their implementation? Or are they just *passively describing* what the code is doing? Obviously, a mirror, even in formal terms, does not add much assurance.

These denotational unit specs are in fact, counted in the spec-to-code ratio in DPN (5:7) and other formally verified smart contracts as well. For example, the Ethereum 2.0 Deposit Contract [40] advertises a 5:3 ratio—even more specs than code. Note that even writing such denotational spec is equally if not more costly than writing code, as developing in an exotic spec language like MSL usually require a dedicated consulting team with years if not decades of experience in formal methods to bootstrap and maintain. Such a team is way more costly than a dev or QA team.

In a mature software verification system, the lifting of an imperative implementation to a pure functional denotation should be automated by a compiler, transpiler, or more commonly referred in the verification toolchain, verification condition generator (VCGen), provided the semantics of the underlying programming language is complete. With all the criticism, we acknowledge that denotational unit specs can be useful and even crucial in the bootstrapping phase of applying formal verification to a project, as manually lifted code reduces the burden on VCGen. Based on GitHub commits, this seems to be true in the DPN case. What is omitted, however, is the removal of the scaffolding after the verification system enters a mature stage.

3.2. Alternative Views

The categorization of unit specs presented in §3.1 takes a narrow view on the assurance brought by unit specs. In fact, it only examines whether a unit spec abstracts (Category-I and II) or helps to abstract (Category-III) the implementation. In practice, more specifically, in the DPN project, stakeholders typically have alternative understandings on the assurance provided by unit specs.

View 1: a unit spec is exhaustive unit testing, i.e., a suite of unit tests that achieves perfect coverage in whatever possible metric. More importantly, when checked into a continuous integration (CI) system, a unit test effectively pins down a very small fraction of the semantics for the target function f. This is evident by the fact that any change to f will be re-tested by existing test cases and if there is a failure, either the code change needs to be rejected or the failed test cases, which indirectly serve as a contract of the agreed semantics, needs to be updated. Similarly, when hooked with CI, if any change to f causes the unit spec to fail, either the code change is problematic or the spec needs to be updated.

View 2: a unit spec is multi-version programming, Formal specs typically require a declarative way of programming which is hopefully different from the imperative code. Therefore, even if bugs can occur in both code and spec, these bugs

are unlikely to manifest in the same way. In other words, unit specs of the same functionality provide *security through diversity* by supplying a "diversified" set of semantics of the same feature. Therefore, a discrepancy between code and spec typically signals a fundamental problem and provokes a deep discussion. This represents assurance brought by N-version programming [11] and specs written with this mindset usually change with code in lockstep.

View 3: a unit spec is documentation in precise terms, especially for external users (including auditors) as clauses in the spec exhaustively enumerate all possible failures this function may run into and the associated conditions. The post-conditions also clear room for any unintended side effects. Compared with the type signatures, unit specs are a stronger form of binding agreements between the users and the developers, especially for compatibility checking. In fact, in DPN, the document generator indeed includes specs by default for external users' consumption.

4. DI as Type System Extensions

Among all flavors of specs in MSL (see §2), invariants over data types are perhaps the most friendly type of specs for developers who do not have much experience on formal verification. This is partially facilitated by the fact that Move enforces a very restrictive, strong, and static type system [42], i.e., the type information for all variables is known at compile-time and a variable cannot change its type in any way at runtime. In this regard, a data invariant (DI) is an extension to an object type by enabling the user to further restrain the set of possible values an object can have during execution.

This is evident in line 15 of Figure 1b. This DI can be viewed as filling a gap in the type system that there is no primitive type such as a `NonZeroU64`, although it is unwise to expect such a primitive type in any practical type system.

4.1. Expressiveness of DI

DI in MSL is effectively a refinement type system [22] and is more expressive and customizable than type systems in most mainstream programming languages (e.g., C++, C#, Java, or Rust) in at least three ways:

Cross-field relation. A DI can specify, in theory, arbitrarily complex relations across multiple fields of a data structure or an ADT. As demonstrated in the crafted example in Figure 5a, the DI for the `SumIsConst` data type requires that the sum of its fields `a` and `b` are always 100 and whether `a` or `b` is larger is decided by the boolean field `c`.

Use of quantifiers. Quantifiers in first-order logic can be used in DI, as demonstrated in Figure 5b, where a mathematical set is modeled after the built-in `vector` type. This is indeed how an unordered set is modeled in DPN.

Temporary violations. In MSL, a DI targeting a data type `T` does not apply to a mutable reference of `T` (i.e., `&mut T`). This is intentional, as Move does not support atomic updates to multiple fields. Therefore, temporary violations of the DI are permitted. The DI, however, will be re-checked once the mutable reference reaches end-of-life and the modifications have all been written back to the owned variable. This is shown in example Figure 5c where the invariants (in Figure 5a) are temporarily violated in lines 6–8 but are re-checked in line 9. Conceptually, the scope of the mutable reference (`p` in this example) marks a critical section within which the DI associated with the data type can be violated temporarily.

```
1  struct SumIsConst {
2    a: u64,
3    b: u64,
4    c: bool,
5  }
6  spec SumIsConst {
7    invariant c ==> (a >= b);
8    invariant !c ==> (b >= a);
9    invariant a + b == 100;
10 }
```

(a) DI on multiple fields[3]

```
1  struct Set<T> {
2    members: vector<T>,
3  }
4  spec Set {
5    invariant forall
6      i in 0..len(members),
7      j in 0..len(members):
8        (members[i] == members[j])
9          ==> (i == j);
10 }
```

(b) DI with quantifiers

```
1  public fun swap(
2      mut arg: SumIsConst
3  ): SumIsConst {
4    let s = &mut arg;
5    let t = s.a;
6    s.a = s.b;
7    s.b = t;
8    s.c = !s.c;
9    // s reaches end-of-life
10   input
11 }
12 spec swap {
13   ensures
14     result.c == !arg.c;
15 }
```

(c) Merge DI and unit spec

```
1  spec swap {
2    requires
3      arg.a + arg.b == 100 &&
4      arg.c ==> (arg.a >= arg.b) &&
5      !arg.c ==> (arg.b >= arg.a);
6    ensures
7      result.a + result.b == 100 &&
8      result.c ==> (
9        result.a >= result.b
10     ) &&
11     !result.c ==> (
12       result.b >= result.a
13     );
14   ensures result.c == !arg.c;
15 }
```

(d) Equivalent unit spec without DI

Figure 5: Examples showing the expressiveness of DI

4.2. Assurance from DI

It is of no doubt that DI can help reduce the lines of specs written. To illustrate, with DI specified in Figure 5a, the unit spec for the `swap()` function is as simple as a single `ensures` clause (see Figure 5c); while expressing the same semantics without DI will be much more involved as shown in Figure 5d, with an extra pair of pre- and post-condition clauses to confine the input and return value, respectively. More importantly, duplications of `SumIsConst`-related specs grow linearly with the number of occasions in which `SumIsConst` instances are involved.

However, expressiveness and spec-reuse do not imply assurance. DI can help shape better code but this effect can be explained by at least two reasons:

- *formal verification*: DI contracts user-facing data types or other critical data types in the system;
- *compile-time social coordination* [9], DI enhances the type system by explicitly capturing developers' intention, such as coding conventions, assumptions, or linting rules.

3. In Figure 5a, the first two invariants overlap when `a == b`. This is intentional as this DI should permit both `{a:50, b:50, c:true}` and `{a:50, b:50, c:false}`.

And the key question remains: do all DI specs add assurance from a formal verification perspective (i.e., the more the better)? If not, how to distinguish DI specs that provide assurance from specs that provide coordination?

Categorization on the unit spec (see §3.1) can shed lights in answering this question. Briefly, we consider the DI to provide formal verification assurance if the DI

- targets a data type that appears in the function signature (i.e., argument and return types) of an external-facing interface (i.e., `public` and `public entry`);
- targets a data type that appears in a unit spec of category-I, II, or III — the DI may be needed to prove refinement.
- targets a data type that may be persisted into the blockchain state (i.e., $\mathbb{G}$). Such data types are marked with either the `key` or `store` ability in the Move language (e.g., `struct Account` in Figure 1).

In all other cases, the DI targets an ephemeral data type which is used only in the internal processing of a blockchain transaction and is oblivious to both users and the blockchain state. Such DI specs are considered providing social coordination than assurance.

5. SI and TI as State Machine Building Blocks

Global invariants are the most recent addition to MSL in terms of language constructs. These specs are introduced based on the insight that the correctness of a blockchain system is not only about each individual transaction (which can be specified using unit specs), but also about the correctness of the entire blockchain state ($\mathbb{G}$), which can be evolved in arbitrary ways depending on the implementation. Ensuring that data persisted in $\mathbb{G}$ is always in integral states and remain integral through state transitions is the goal for state invariants (SI) and state transition invariants (TI), respectively, as both specify directly over $\mathbb{G}$.

5.1. Expressiveness of SI And TI

Unlike unit spec which is attached to a concrete implementation (§3) or DI which is attached to a concrete data type (§4), SI and TI are high-level specs that are oblivious to most of the implementation details. The only details SI and TI need to be aware of are user-defined data types that may appear in $\mathbb{G}$. Such data types are clearly marked by the `key` or `store` ability modifier in their declarations.

Further recall from §2 that in Move, the blockchain state, $\mathbb{G}$, is essentially a map indexed by both `address` and type, most SI and TI specs will be universally or existentially quantified over `address` to indicate that the invariant applies generically to all addresses, as shown in both Figure 1b and Figure 6b. But still, it is possible to write unquantified expressions in SI or TI. For example, the following SI specifies that block time starts after genesis is done:

```
invariant [state] exists<Genesis>(@ROOT) ==> exists<Time>(@ROOT);
```

In general, in MSL, SI and TI expressions are allowed to refer to any and multiple addresses or data types, as long as the data types may appear in $\mathbb{G}$. Generic types will be monomorphized to concrete types by VCGen in the context when they become relevant. In addition, TI expressions are allowed to use the `old(..)` operator to retrieve a snapshot of $\mathbb{G}$ in pre-state, i.e., before the transaction. In short, any first-order logic properties about $\mathbb{G}$ can be expressed with SI or TI, making such specs extremely expressive.

Relationship with unit specs. As higher-level spec facilities, any SI and TI can refine to clauses in unit specs. One refinement algorithm is that every SI is translated to an assumption at the beginning of a `public entry` function (f) and an assertion at the end of f—an induction proof; while every TI is translated to an assertion at the end of f with `old(..)` capturing the pre-state $\mathbb{G}$ before the transaction.

It is, however, sometimes desirable that SI and TI are asserted more frequently during the execution of a transaction. For example, if f calls another function a during the execution, it might be useful to check that an SI/TI holds before and after a. This is function-level granularity and is the default granularity of enforcing SI in MVP. An alternative is to enforce an SI/TI after every instruction that may modify the $\mathbb{G}$ covered in the SI/TI spec. This is the finest granularity which is also how TI is enforced in MVP.

Relationship with DI. SI or TI cannot be refined to DI and vice versa. SI can be similar to DI in the simplest cases, when a type `T` only appears in $\mathbb{G}$ and the SI invariant inv about `T` is universally quantified over address. However, even in this simple case, inv is only applicable when $\mathbb{G}$`[(address, T)]` is involved; while a DI over `T` is applicable whenever a variable of type `T` is involved. On the other hand, invariants in SI can involve multiple addresses (quantified or constant) across multiple types—these are not the concerns for DI.

5.2. The Essence of SI And TI

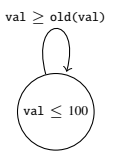
While the operational semantics of SI and TI are intuitive and publicly documented [17], [35], the rationale and the entity modeled by SI and TI are not. State abstractly, SI and TI are used to formally define **an infinite number of communicating finite-state machines** (CFSM) that collectively operate over a common $\mathbb{G}$.

Specifically, a single FSM is associated with one address and operates over a slice of the global storage owned by this address, denoted as $\mathbb{G}$`[(*, address)]` where `*` represents all data types stored under this slice. Given that $\mathbb{G}$ can host an infinite number of addresses in theory, there will be an infinite number of FSMs. A transaction that updates the states of one FSM (e.g., the `foo()` function in Figure 4a) can be modeled by a state transition within the FSM; while a transaction that updates the states of multiple addresses (e.g., the `transfer()` function in Figure 1a) can be modeled by involved FSMs first updates its states locally and then communicating with each other to synchronize their updates.

In this view, the expressiveness of SI and TI can be leveraged to define the states and state transitions of the infinite number of CFSMs, respectively. More specifically,

- SI defines the nodes (i.e., legal and illegal states) of each FSM associated with an address as well as the relations among the values of the nodes, represented as SI_predicate($\mathbb{G}$`[(a1, T1)]`, $\mathbb{G}$`[(a2, T2)]`, ...): `bool`
- TI defines the edges (i.e., legal and illegal state transitions) of each FSM associated with an address as well as the relations among the values of the edges, represented as: TI_predicate($\Delta\mathbb{G}$`[(a1, T1)]`, $\Delta\mathbb{G}$`[(a2, T2)]`, ...): `bool`

To illustrate how SI and TI can define an FSM, Figure 4 is a simple example without cross-FSM communications. The FSM associated with each individual address will be identical and is shown in Figure 6, which is defined by the SI and TI in Figure 1b. As evident by the comparison, the FSM spec is significantly more succinct, elegant, and readable than the unit specs equivalent. The running example in Figure 1 is slightly more complicated as it does involve FSMs communicating upon state update. More specifically, as required by the TI, the reduction of balance in one `Account` should be match the increase of balance in a different `Account`. This relation is captured in Figure 7 in the dashed line between the two state transitions of two FSMs. In particular, the TI requires that the change of balances (Δ) in these two state transitions should be equal.



(a) FSM

```
invariant [state] forall addr: address
  where exists<S>(addr):
    global<S>(addr).val <= 100;

invariant [update] forall addr: address
  where exists<S>(addr):
    global<S>(addr).val >= old(global<S>(addr)).val;
```

(b) Spec

Figure 6: Intended FSM and its specs for code in Figure 4

The type system that might be used to reason about SI and TI specs is **dependent multi-party session types over an infinite number of parties**, which is not defined to the best of our knowledge, although the ingredients of such a type system (dependent session type [49], multi-party session type [45], and infinite session type [23]) have been studied. We leave the integration of ingredients into future work.

5.3. Assurance from SI And TI

In this section, we ask the same question as we asked for unit specs and DI: do SI and TI specs provide assurance from a formal verification perspective (and hence, the more the better)? We are leaning towards a positive answer to this question: SI and TI provide assurance unconditionally and hence, should be encouraged or prioritized when specifying a Move-based smart contract.

Our answer is based on three empirical reasons:

- For smart contracts, the blockchain storage $\mathbb{G}$ is the only way to persist states. Therefore, any implementation of the smart contract can be abstracted as a series of infinite CFSMs operating over $\mathbb{G}$. And the fact that SI and TI can directly specify the CFSMs indicates that they can be an optimal way to abstract and contract the implementation.
- In addition, on most blockchains, storing information in $\mathbb{G}$ incurs costs, and these costs force developers to keep only critical information on the chain. Hence, the design of data types that can be stored on-chain are closer to capture core and abstracted states of the implementation.
- As evident empirically in DPN, SI and TI are used to capture the most critical safety requirements such as the access control scheme [8].

However, it is worth-noting that SI and TI might not be always feasible in formal verification systems beyond smart contracts, as $\mathbb{G}$ might not be readily defined. For example, for programs written in C/C++ or Rust, $\mathbb{G}$ (which could be heap or filesystem) needs to be explicitly marked and defined before SI and TI can be applicable.

6. Summary

We summarize the answers to the key question raised in §1: **how to write specs in a way that maximizes ROI?**

Generic answer. Developers should be encouraged to write specs that abstracts the underlying systems. Specs of the proof assistance nature should be developed on an as-need basis (e.g., when automated verification fails) while denotational specs should be avoided.

Practical answers specific to Move. Among the three types of specs discussed in this research report:

- SI and TI should be encouraged as these specs target the CFSM directly and hence have the chance to abstract the most number of lines in code.
- Unit specs of the following types should be encouraged:
  - contracting a user-facing interface (i.e., a `public` or `public entry` function), or
  - confining the implementation to a set of possibilities (i.e., with `[concrete]` annotated clauses), or
  - assisting the refinement proof for other abstracting specs (i.e., with `[abstract] annotated clauses`).
- DI should be encouraged when it targets a data type that
  - appears in the function signature of a user-facing interface (i.e., a `public` or `public entry` function), or
  - appears in a unit spec of interest (listed above), or
  - can be persisted into the blockchain state (i.e., $\mathbb{G}$), as marked by the `key` or `store` ability modifier.

7. Future Work

We have several ongoing and future work planned to bring this research report into a full academic paper.

7.1. Surveying More Formal Verification Systems

This research report is written based on one case study: DPN developed in Move and verified by MVP. Although we believe that the lessons from DPN would generalize to other systems and other verification languages, supporting evidence is missing. We are currently surveying the following software verification systems: (1) Frama-C, (2) Software Analysis Workbench (SAW), (3) TLA+, (4) KeY, (5) Prusti,

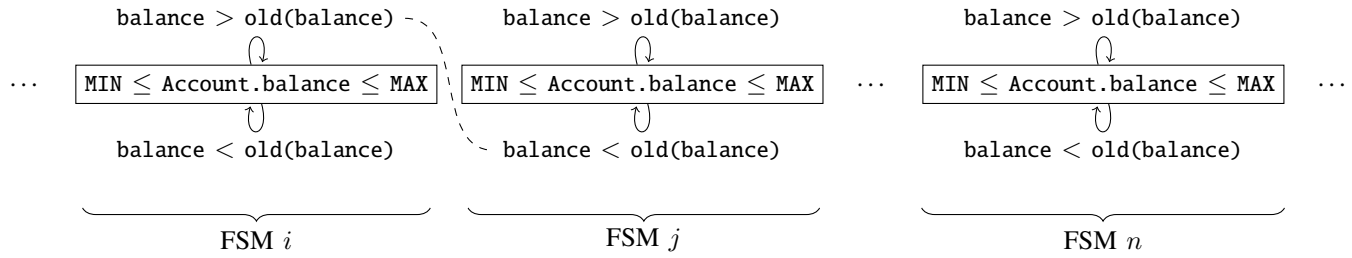


Figure 7: CFSMs of the smart contract example in Figure 1a. (1) Each `𝔾[(*,address)]`, i.e., a slice of global storage owned by one address, can be modeled as an FSM (e.g., FSM i, j, and n). These FSMs are defined by the SI in 1b. (2) Different FSMs "communicate" and "synchronize" their state updates (the dashed line between FSM i and j). The synchronized state update is defined by the TI in 1b

(6) NatSpec and SMTChecker for Solidity, and (7) Certora Prover; with a goal of understanding how their spec language is designed, especially on differentiating abstracting specs, proof assistance, and denotational specs. We also plan to substantiate our recommended practices in spec language design (§6) with objective and well-designed user studies.

7.2. On The Completeness of Specs

Even a program is thoroughly verified with a perfect verification toolchain, this program is only as correct as its spec. But how can we be sure that the spec is correct or at least complete? More specifically, are all *important* aspects encoded in the implementation covered in the specs?

Unfortunately, the Move spec language provides inadequate guarantee on completeness. As discussed in §3, a Move unit spec aims to be a mathematical object representing *this function does X and only X*, but is unfortunately incomplete in confining the *only X* part. This make it easy to (inadvertently) introduce "gaps" in the specs.

Figure 8 is a representative example to illustrate "gaps" in unit specs. The S1 version of the `add1` spec seems very benign and is likely to be the output of someone who just started writing spec. But it has a serious gap, as illustrated by the fact that both C1 and C2 of the `add1` implementation verifies against S1! The complete spec is in fact given in S2.

```
// C1 -- correct
fun add1(v: &mut vector<u64>) {
  for i in 0..len(v) {
    v[i] = v[i] + 1;
  }
}

// C2 -- w/ backdoor
fun add1(v: &mut vector<u64>) {
  for i in 0..len(v) {
    v[i] = v[i] + 1;
  }
  // secret backdoor
  vector::pop(v);
}
```

```
// S1 -- incomplete
spec add1 {
  ensures forall
    i in 0..len(v):
      v[i] = old(v)[i] + 1;
}

// S2 -- complete
spec add1 {
  ensures forall
    i in 0..len(v):
      v[i] = old(v)[i] + 1;
  ensures len(v) ==
          len(old(v));
}
```

Figure 8: An illustration of incompleteness in the spec

A gap in the spec can create verification blind spots in which the behavior of the function is unconstrained. In the worst case, there is nothing to prevent a malicious developer from hiding backdoors behind these blind spots, and yet, such malicious code can survive regardless of how rigorous the verification is — a single blind spot in the spec can easily undermine months if not years of verification efforts.

However, it is worth-noting that this is not a problem unique to Move-based smart contracts, but a potential threat to other formally verified systems as well, including but not limited to Amazon s2n [46], seL4 [27], CompCert [31], etc.

The incompleteness issue is not unique to specs as well: test suites also need a quantitative measurement for its quality and the generally accepted metric is code coverage (e.g., line or branch coverage). Paranoid maintainers may even require that any new code contribution needs to be accompanied by test cases to maintain a high coverage ratio.

Observation. In a contrast to testing, to the best of our knowledge, there is no tooling to measure coverage for specs: i.e., which lines of code are covered (and confined) by which spec. Given the way how the verification conditions are generated in modern provers [5], [10], [20], [30], how to untangle the complicated logical formula is a technical challenge. And yet, we believe that such tooling is necessary when formal verification gains enough traction.

Potential solutions. Based on the disparity that searching for a proof can be hard but verifying a proof is easy (relatively), proof-carrying code (PCC) [39] promises that a straightforward verifier can be designed to efficiently verify properties about the code via a formal proof embedded in the code. In the context of Move, having a PCC compiler can help lower all kinds of specs in the Move formal verification system (unit specs, DI, SI, and TI) into a new type system that can be checked by a lightweight verifier before the execution of Move code.

On the first look, the problem PCC aims to solve is orthogonal to spec incompleteness as the verifier will still accept a proof to an incomplete spec. However, a mechanically-verifiable proof enables data flow analysis on the proof itself. The analysis can determine which variables contribute to the establishment of claimed properties, and more importantly, which variables do not participate in the proof — the latter signals potential gaps in the specs.

Foundational proof-carrying code (FPCC) further minimizes the trust on the verifier and mandates the language semantics to be encoded as part of the proof as well. The no-other-effects policy can be encoded into FPCC as part of the language semantics. This effectively allows the proof

to claim non-interference like *if this instruction updates variable X, no other variables will be updated*; and the verifier will accept it. In other words, FPCC can provide the *"only X"* part that is currently missing in Move unit specs.

7.3. Expanding Scope for SI and TI

Applicability in Rust. Although SI and TI (specs organized around a global state $\mathbb{G}$) are incepted in the Move language, they can be powerful constructs for verifying programs written in other language too. For example, in safe Rust which is strongly and statically typed, the collection of all global variables in the program can be represented as $\mathbb{G}$. Taking immutable or mutable references of a global variable can hence be modeled as `borrow_global<T>(variable)` and `borrow_global_mut<T>(variable)` operations in Move, hinting that writing SI and TI specs around global variables might be feasible in safe Rust.

Relevance to separation logic. Separation logic [44] is not (explicitly) used in the specification and verification of Move programs. This is due to two reasons:

- The entire global state ($\mathbb{G}$) is partitioned by type, i.e., $\mathbb{G}$`[(T1,*)]` and $\mathbb{G}$`[(T2,*)]` are disjoint as long as `T1` and `T2` are not the same type; and
- Borrow semantics apply to each type-based slice of $\mathbb{G}$. For example, if function `a` borrows or modifies $\mathbb{G}$`[(T,*)]` and `a` calls `b`, `b` is not allowed to operate on $\mathbb{G}$`[(T,*)]`.

As a result, SI and TI specs about disjoint slices of $\mathbb{G}$ are naturally composable. However, in system programming languages like C, C++, or Java, even if we can partition the global heap by type, each slice $\mathbb{G}$`[(T,*)]` can be referred to by multiple alive pointers that may refer to any slot in the slice, e.g., $\mathbb{G}$`[(T,address1)]` and $\mathbb{G}$`[(T,address2)]` with no guarantee of `address1 != address2`. Therefore, to port SI and TI to these languages, we might need to express and verify SI and TI around $\mathbb{G}$`[(T,*)]` with separation logic.

Inference. In addition, SI and TI also open new doors for spec inference. For example, in a Move program, invariants over $\mathbb{G}$ can be inferred by statically extracting, aggregating, and abstracting all operations on $\mathbb{G}$. Such invariants might also be mined based on runtime analytics, e.g., via a replay of historical or fuzzing-generated transactions.

8. Conclusion

In this paper, we introduce three types of specs available in the Move language and argue that not all spec are equally important in providing assurance. We further suggest a practical way on writing specs that yield a higher ROI by prioritizing specs that abstract the system (see §6). Despite that, each type of spec has its own conceptual goals and a formal verification system should consciously combine all of them to maximize the assurance provided by the spec suite.

9. Acknowledgment

I would like to thank the entire Move and DPN team for their comments, inspiration, and continued support. This research report is essentially a summarization of years-long discussions happened in the MVP development team, with a non-exhaustive list of members including Wolfgang Grieskamp, David Dill, Junkil Park, Teng Zhang, Shaz Qadeer, Emma Zhong, and Sam Blackshear.

I would also like to pay special thanks to Jon Howell for the inspirational feedback on drawing a boundary between trusted text (spec) and untrusted text (code), the paper shepherd Michael Clark for the time and guidance during the revision process, and the anonymous reviewers for their insightful feedback (e.g., on PCC and FPCC).

This research was funded by research gifts from Meta, Aptos Foundation, Sui Foundation, and Amazon.

References

[1] Andrew W. Appel. Foundational Proof-Carrying Code. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, Boston, MA, jun 2001.

[2] Andrew W. Appel. Verified Software Toolchain. In *Programming Languages and Systems*, Berlin, Heidelberg, 2011.

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of Symbolic Execution Techniques. *ACM Computing Surveys (CSUR)*, 51(3), 2018.

[4] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Munich, Germany, April 2022.

[5] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A Modular Reusable Verifier for Object-oriented Programs. In *Proceedings of the 2005 International Symposium on Formal Methods for Components and Objects (FMCO)*, Amsterdam, The Netherlands, August 2005.

[6] Mike Barnett and K. Rustan M. Leino. Weakest-Precondition of Unstructured Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, sep 2005.

[7] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian 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.

[8] Sam Blackshear, Tim Zakian, and Junkil Park. DIP-2: Diem Roles and Permissions. https://github.com/diem/dip/blob/main/dips/dip-2.md, 2020.

[9] Zac Burns. Compile-Time Social Coordination. https://2021.rustconf.com/talks/compile-time-social-coordination, 2021.

[10] Kyle Carter, Adam Foltzer, Joe Hendrix, Brian Huffman, and Aaron Tomb. SAW: The Software Analysis Workbench. In *Proceedings of the 21st European symposium on programming (ESOP)*, Pittsburgh, PA, March 2013.

[11] Liming Chen and Algirdas Avizienis. N-Version Programming: A Fault-tolerance Approach to Reliability of Software Operation. In *Proceedings of the 8th IEEE International Symposium on Fault-Tolerant Computing (FTCS-8)*, 1978.

[12] Concourse Open Community. DeFi Pulse. https://defipulse.com/, 2022.

[13] CryptoSec Group. Documented Timeline of DeFi Exploits. https://cryptosec.info/defi-hacks/, 2022.

[14] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, March–April 2008.

[15] Diem Association. Diem White Paper v2.0. https://www.diem.com/en-us/white-paper/, 2020.

[16] Diem Association. Diem Core GitHub Page. https://github.com/diem/diem, 2023.

[17] David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Emma 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, April 2022.

[18] Michael Felderer, Dilian Gurov, Marieke Huisman, Björn Lisper, and Rupert Schlick. Formal Methods in Industrial Practice - Bridging the Gap (Track Summary). In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Rhodes, Greece, October 2018.

[19] Alessio Ferrari and Maurice H. Ter Beek. Formal Methods in Railways: A Systematic Mapping Study. *ACM Computing Surveys*, 55(4), nov 2022.

[20] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where Programs Meet Provers. In *Proceedings of the 22nd European symposium on programming (ESOP)*, Rome, Italy, March 2013.

[21] George Fink and Matt Bishop. Property-Based Testing: A New Approach to Testing for Assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4), 1997.

[22] Tim Freeman and Frank Pfenning. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI)*, 1991.

[23] Simon J. Gay, Diogo Poças, and Vasco T. Vasconcelos. The Different Shades of Infinite Session Types. In *Foundations of Software Science and Computation Structures*, apr 2022.

[24] Jean-Yves Girard. Linear Logic. *Theoretical computer science*, 50(1), 1987.

[25] Mario Gleirscher and Diego Marmsoler. Formal methods in dependable systems engineering: a survey of professionals from Europe and North America. *Empirical Software Engineering*, 25(6), 2020.

[26] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, Paris, France, January 2017.

[27] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[28] Tomas Kulik, Brijesh Dongol, Peter Gorm Larsen, Hugo Daniel Macedo, Steve Schneider, Peter W. V. Tran-Jørgensen, and James Woodcock. A Survey of Practical Formal Methods for Security. *Formal Aspects of Computing*, 34(1), jul 2022.

[29] Phillip A. Laplante and Mohamad Kassab, editors. *Requirements Engineering for Software and Systems*. Auerbach Publications, New York, NY, 4th edition, 2022.

[30] K Rustan 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)*, Dakar, Senegal, April 2010.

[31] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7), July 2009.

[32] Libra Engineering Team. Libra Core roadmap #3. https://www.diem.com/en-us/blog/libra-core-roadmap-3/, 2020.

[33] Move Core Team. Documentation Generator. https://github.com/diem/diem/blob/latest/language/move-prover/doc/user/docgen.md, 2021.

[34] Move Core Team. Move Executable Specifications. https://github.com/move-language/move/tree/main/language/move-prover/interpreter, 2021.

[35] Move Core Team. Move Specification Language. https://github.com/move-language/move/blob/main/language/move-prover/doc/user/spec-lang.md, 2022.

[36] Move Core Team. The Move Book. https://move-language.github.io/move/, 2022.

[37] Move Core Team. Awesome Move. https://github.com/MystenLabs/awesome-move, 2023.

[38] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008.

[39] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Paris, France, 1997.

[40] Daejun Park, Yi Zhang, and Grigore Rosu. End-to-End Formal Verification of Ethereum 2.0 Deposit Smart Contract. In *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV)*, Los Angeles, CA, July 2020.

[41] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A Formal Verification Tool for Ethereum VM Bytecode. In *Proceedings of the 26th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Lake Buena Vista, FL, November 2018.

[42] Marco Patrignani and Sam Blackshear. Robust Safety for Move. https://arxiv.org/abs/2110.05043, 2021.

[43] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. VerX: Safety Verification of Smart Contracts. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[44] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17thAnnual IEEE Symposium on Logic in Computer Science (LICS)*, Copenhagen, Denmark, June 2002.

[45] Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. In *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*, Cascais, Portugal, January 2019.

[46] Steve Schmidt. Introducing s2n-tls, a New Open Source TLS Implementation. https://aws.amazon.com/blogs/security/introducing-s2n-a-new-open-source-tls-implementation/, 2022.

[47] The Solidity Authors. NatSpec Format. https://docs.soliditylang.org/en/latest/natspec-format.html, 2024.

[48] The Solidity Authors. SMTChecker and Formal Verification. https://docs.soliditylang.org/en/latest/smtchecker.html, 2024.

[49] Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent Session Types via Intuitionistic Linear Type Theory. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming (PPDP)*, 07 2011.

[50] Liping Zhao, Waad Alhoshan, Alessio Ferrari, Keletso J. Letsholo, Muideen A. Ajagbe, Erol-Valeriu Chioasca, and Riza T. Batista-Navarro. Natural Language Processing for Requirements Engineering: A Systematic Mapping Study. *ACM Computing Surveys*, 54(3), apr 2021.

[51] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L. Dill. The Move Prover. In *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV)*, Los Angeles, CA, July 2020.

[52] Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. VST-A: A Foundationally Sound Annotation Verifier. *Proceedings of the ACM Programming Languages*, 8(POPL), jan 2024.

[53] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A Survey for Roadmap. *ACM Computing Surveys (CSUR)*, 54(11s), 2022.

Appendix

1. Rationale for Calling Specs Denotational

We first clarify the adjectives terms related to semantics:

- Both *operational* and *denotational* semantics are both about a programming language (e.g., the Move language in this research report).
- *Axiomatic* semantics are about the intended behaviors of the program itself. The program can be implemented in any language while the axiomatic semantics is expressed within some program logic (e.g., first-order logic).

In terms of Move, its operational semantics is defined by the interpreter (called Move VM). For example, the operational semantics of the `move_from<T>(addr)` primitive can be sketched in the following algorithm:

Algorithm 1: Operational semantics of `move_from`

Input: *addr*: address
Input: *T*: TypeName
Input: *global_state*: HashMap<address,Account>, where Account is HashMap<TypeName,Bytes>
Output: program aborts or an object of type T and new *global_state*

```
if global_state.has_address(addr) then
    let account ← global_state.get(addr);
    if account.has_type(T) then
        let data ← account.get(T);
        account.delete(T);
        return cast_to_type(data, T)
          with side-effect of updating global_state;
    else
        abort!
else
    abort!
```

The denotational semantics of Move is (implicitly) hardcoded in MVP as part of the VCGen process. More specifically, VCGen translates every Move statement and expression into an SMT representation that encodes the semantics. Take, again, the `move_from<T>(addr)` built-in as an example, the denotational semantics of this primitive is encoded in SMT format as shown in Figure 9.

As a result, the spec in Figure 10 is considered redundant and is the same symbolic representation of the `move_from` semantics expressed in the glsspec language. In other words, writing the spec for `move_from_wrap` is just repeating the denotational semantics of `move_from` primitive in MSL. Hence we call this type of spec denotational spec.

```
(set-logic ALL)

; Account address is modeled as an suninterpreted sort
(declare-sort Address 0)

; Parameter `T` in `move_from<T>` can be monomorphized to either
; - a concrete type, or
; - another type parameter (e.g., of the calling function)
; We show the second case in this modeling as it is more generic
(declare-sort T 0)

; Global state sliced by `T` is modeled as an SMT array that
; - maps from `Address` to an `Option<T>` where
;   - `none` means object does not exist, while
;   - `some` means object exists and can be accessed by `val`
; - the SMT array has `none` as the default value
(declare-datatype Option (
    (none)
    (some (val T))
))

; A compact datatype to represent the result of `move_from`
; - `abort` means the execution terminates abruptly and
;    both the return value and state will be undefined
; - `finish` means the execution finishes and will define
;    both the return value (`ret`) and global state (`state`)
(declare-datatype MoveFromResult (
    (abort)
    (finish (ret T) (state (Array Address Option)))
))

; Semantics of `move_from` can be encoded as an SMT function
(define-fun move_from
    ; domain and codomain
    (
        (stat (Array Address Option))
        (addr Address)
    ) MoveFromResult

    ; rule
    (match (select stat addr) (
        ; abort when address does not exist
        (none abort)
        ; return the value and an updated global state
        ((some val) (finish
            val
            (store stat addr none) ; array extensionality
        ))
    ))
)
```

Figure 9: Denotational semantics of `move_from`

```
fn move_from_wrap<T>(addr: address): T {
  move_from<T>(addr)
}
spec move_from_wrap {
  aborts_if !exists<T>(addr);
  ensures result == old(global)<T>(addr);
  ensures !exists<T>(addr);
  ensures forall a: address:
    a != addr => global<T>(a) == old(global)<T>(a);
}
```

Figure 10: Denotational specs wrapping around `move_from`

Generally speaking, MVP has denotational encoding of each Move statement and expression, including the call instruction that transfers control flow into another function. So another example of a denotational spec is Figure 4

As shown in this example, through the use of `include RunFooAbortsIf` and `include RunFooAbortsIf` in the spec, we encoded the semantics that function `foo` calls `run_foo`. This spec is, again, denotating the meaning of a function call in the `foo` function.

In summary: denotational spec is the symbolic representation of function semantics that the MVP can derive itself translated and expressed in the spec language. Hence, it brings no additional information (e.g., abstraction, assumption, domain knowledge, etc) into the verification process.

2. Memory Model And Reference Elimination

In Move, the reference system is based on borrow semantics [26] as in the Rust language:

- The initial borrow must come from either a global memory or a local variable on stack (both referred to as root of borrow from now on).
- For local variables (e.g., `x`), one can create immutable references (`&x`) and mutable references (`&mut x`).
- For global memories, the references can be created via the `borrow_global` and `borrow_global_mut` built-ins.
- Given a reference to a whole struct `x`, field borrowing can occur via `&x.f` and `&mut x.f`.
- Similarly, with a reference to a vector `v`, element borrowing occurs via native functions `Vector::borrow(v,i)` and `Vector::borrow_mut(v, i)`.

Move provides the following guarantees, which are enforced by the borrow checker as part of the type system in Move:

- For any roots of borrow, there can be either exactly one mutable reference or n immutable references, no other combinations are allowed. Enforcing this rule is similar to enforcing the borrow semantics in Rust, except for global memories, which do not exist in Rust. For global memories, this rule is enforced via the `acquires` annotations. Using Figure 1a as an example, function `transfer` `acquires` the `Account` global location, therefore, `transfer` is prohibited from calling any other function that might also borrow or modify the `Account` global memory.
- The lifetime of references to data on the stack cannot exceed the lifetime of the stack itself. This includes global memories borrowed inside a function as well — a reference to a global memory cannot be returned from the function, neither in whole nor in parts.

These properties effectively permit the *elimination of references* from a Move program, eliminating need to reason about aliasing.

Immutable references. Immutable references are replaced by values. An example of the transformation is shown below.

```
fun select_f(s: &S): &T { &s.f }
```

⇝⇝⇝

```
fun select_f_with_ref_elimination(s: S): T { s.f }
```

Notably, when executing a Move program, immutable references are important to avoid copies for performance and to enforce ownership; however, for formal reasoning on Move program correctness, the distinction between immutable references and values is unimportant.

Mutable references. Each mutation of a root of borrow `l` starts with an initial borrow for the whole data stored in this location. This borrow creates a reference `r`. As long as `r` is alive, Move code can either update its value (`*r = v`), or derive a sub-reference (`r' = &mut r.f`). The mutation ends when `r` (and the derived `r'`, if any) go out of scope.

The borrow checker guarantees that during the mutation of data in `l`, no other reference to same data in `l` can co-exist. Hence, it is impossible for other Move code to test whether the value has mutated while the reference is held.

These semantics allow mutable references to be handled via *read-update-write* cycles. One can create a copy of the data in `l` and perform a sequence of mutation steps which can be represented as purely functional data updates. Once the last reference for the data in `l` goes out of scope, the updated value is written back to `l`. This converts an imperative program with references into an imperative program which only has state updates on global memory or variables on the stack, with no aliasing. We illustrate the basics of this approach by an example:

```
fun increment(x: &mut u64) { *x = *x + 1 }
fun update(s: &mut S) { increment(&mut s.f) }
fun caller(): S { let s = S{f: 0}; update(&mut s); s }
```

⇝⇝⇝

```
fun increment(x: Mutation<u64>): Mutation<u64> {
    x[value = x.value + 1]
}
fun update(s: Mutation<S>): Mutation<S> {
    let f = Mutation<u64> {
        root: s.root,
        path: s.path ++ FieldAccess(S.f),
        value: s.f,
    };
    let f2 = increment(f);
    s[value.f = f2.value]
}
fun caller(): S {
    let s = S{f: 0};
    let r = Mutation<S> {
        root: LocalRoot(LocalIndex(s)),
        path: [],
        value: s,
    };
    r2 = update(r);
    r2.value
}
```

The `Mutation` type is an SMT datatype modeled as:

```
(declare-sort Address 0)
(declare-sort T 0)

; Identifiers for parameter, local variables, and fields
(declare-sort ParamIndex 0)
(declare-sort LocalIndex 0)
(declare-sort FieldAccess 0)

; Roots of a mutation
(declare-datatype MutRoot (
    (param (pidx ParamIndex))
    (local (lidx LocalIndex))
    (global (addr Address))
))

; A path segment of the borrow chain
(declare-datatype PathSegment (
    (field (f FieldAccess))
    (element (e Int))
))

; Mutation modeled as an SMT algebraic datatype
(declare-datatype Mutation (
    (Mutation
        (root MutRoot)
        (path (Seq PathSegment))
        (val T)
    )
))
```

3. Semantics of Move Global State Operations

The semantics of Move operations around the global state $\mathbb{G}$ can be modeled as SMT functions shown below.

- `exists<T>(address)` $\rightarrow$ `boolean`: checks whether an instance of type `T` exists at the given address.

```
(define-fun exists
    (
        (stat (Array Address Option))
        (addr Address)
    ) Bool
    (= (select stat addr) none)
)
```

Figure 11: Semantics of `exists`

- `move_from<T>(address)` $\rightarrow$ `T`: grabs ownership of the instance of type `T` from the address (and hence $\mathbb{G}$) to the current calling context and leaves a `None` mark in $\mathbb{G}$. Aborts if the instance does not exist.

```
(declare-datatype MoveFromResult (
    (abort)
    (finish (ret T) (state (Array Address Option)))
))
(define-fun move_from
    (
        (stat (Array Address Option))
        (addr Address)
    ) MoveFromResult
    (match (select stat addr) (
        (none abort)
        ((some val) (finish
            val
            (store stat addr none) ; array extensionality
        ))
    ))
)
```

Figure 12: Semantics of `move_from`

- `move_to<T>(address, T)`: transfers ownership of the instance of type `T` from the current calling context to the address in $\mathbb{G}$. Aborts if the instance already exists, as required by linear typing.

```
(declare-datatype MoveToResult (
    (abort)
    (finish (state (Array Address Option)))
))
(define-fun move_to
    (
        (stat (Array Address Option))
        (addr Address)
        (item T)
    ) MoveToResult
    (match (select stat addr) (
        (none (finish (store stat addr (some item))))
        ((some __) abort)
    ))
)
```

Figure 13: Semantics of `move_to`

- `borrow_global<T>(address)` $\rightarrow$ `&T`: temporary borrows an immutable reference to the instance of type `T` at the address in $\mathbb{G}$ in the current calling context. Aborts if the instance does not exist. The type system ensures that the reference does not become dangling (e.g., via `move_from` or `move_to`) nor alias with any mutable reference. This enables reference elimination (see §B for details) and hence the return value from `borrow_global` from MVP's perspective is just a value of type `T`.

```
(declare-datatype BorrowGlobalResult (
    (abort)
    (finish (ret T) (state (Array Address Option)))
))
(define-fun borrow_global
    (
        (stat (Array Address Option))
        (addr Address)
    ) BorrowGlobalResult
    (match (select stat addr) (
        (none abort)
        ((some val) (finish val stat))
    ))
)
```

Figure 14: Semantics of `borrow_global`

- `borrow_global_mut<T>(address)` $\rightarrow$ `&mut T`: temporary borrows a mutable reference to the instance of type `T` at the address in $\mathbb{G}$ in the current calling context. Aborts if the instance does not exist. The type system ensures that the reference does not become dangling (e.g., via `move_from` or `move_to`) nor alias with any other reference. This enables reference elimination (see §B for details) and hence the return value from `borrow_global_mut` from MVP's perspective is a `Mutation` parameterized by type `T`.

```
(declare-datatype BorrowGlobalMutResult (
    (abort)
    (finish
        (ret Mutation)
        (state (Array Address Option))
    )
))
(define-fun borrow_global_mut
    (
        (stat (Array Address Option))
        (addr Address)
    ) BorrowGlobalMutResult
    (match (select stat addr) (
        (none abort)
        ((some val) (finish
            (Mutation
                (global addr)
                (as seq.empty (Seq PathSegment))
                val
            )
            stat
        ))
    ))
)
```

Figure 15: Semantics of `borrow_global_mut`