Not All Move Specifications Are Created Equal

A Case Study on the Formally Verified Diem Payment Network

Meng Xu *(University of Waterloo)*

May 23, 2024

About me

- 2014 – 2020: *PhD*, Georgia Institute of Technology
  - fuzzing
  - symbolic execution
  - moving-target defense
- 2020 – 2021: *Research Scientist*, Meta
  - member of the Move language team in Diem
  - primarily worked on Move Prover (formal verification tool)
  - also worked on Move VM, file format, and bytecode verifier
- 2022 – now: *Assistant Professor*, University of Waterloo
  - anything on finding bugs or fixing bugs interests me!

About me

- 2014 – 2020: *PhD*, Georgia Institute of Technology
  - fuzzing
  - symbolic execution
  - moving-target defense
- 2020 – 2021: *Research Scientist*, Meta
  - member of the Move language team in Diem
  - primarily worked on Move Prover (formal verification tool)
  - also worked on Move VM, file format, and bytecode verifier
- 2022 – now: *Assistant Professor*, University of Waterloo
  - anything on finding bugs or fixing bugs interests me!

About this research report

This research report is more like a **reflection** from a **power user** of formal verification tools.

About this research report

This research report is more like a **reflection** from a **power user** of formal verification tools.

- Power user: not an expert, but just someone really likes the technology and wants to use it properly.
- Reflection: things that align and do not align with my initial expectations about formal verification.

About the Move language

- Move is based on the concepts of
  - borrow semantics (like Rust) and
  - linear types (like the `unique_ptr` in C++)
- Move does not support high-order functions (at least initially).
  - a.k.a., dynamic dispatching
  - Not all callsites have to be determined statically
- A Move program interacts with external states through a small and fixed set APIs

About the Move language

- Move is based on the concepts of
  - borrow semantics (like Rust) and
  - linear types (like the `unique_ptr` in C++)
- Move does not support high-order functions (at least initially).
  - a.k.a., dynamic dispatching
  - Not all callsites have to be determined statically
- A Move program interacts with external states through a small and fixed set APIs
- Comes with a fully integrated **specification language** supporting pre/post conditions and global state invariants
  - Uses full first-order predicate calculus
  - including universal and existential quantification

Look and feel: programming in Move

```
module 0x1::Account {
    // resources that can be stored in global storage
    struct Account has key {
        balance: u64
    }

    fun withdraw(account: address, amount: u64) acquires Account {
        // creates a mutable reference to an item in the global storage
        let ptr: &mut Account =
            borrow_global_mut<Account>(account);
        ptr.balance = ptr.balance - amount;
    }

    // the "main" function for a transaction
    public entry fun transfer(
        from: address, into: address, amount: u64
    ) acquires Account {
        withdraw(from, amount);
        deposit(into, amount);
    }
}
```

Look and feel: specifying in Move

```
module 0x1::Account {
    // specification helper function
    spec fun bal(acc: address): u64 { global<Account>(acc).balance }

    // specification of function transfer (aborts and post-conditions)
    spec transfer {
        aborts_if bal(from) < amount;
        aborts_if bal(into) + amount > MAX_U64;
        ensures bal(from) == old(bal(from)) - amount;
        ensures bal(into) == old(bal(into)) + amount;
    }

    // inductive invariant on global state:
    // - an account balance is never under minimal value
    invariant
        forall acc: address where exists<Account>(acc):
            bal(acc) >= MIN_BALANCE;

    // pre-post state invariant on global state:
    // - account value cannot decrease more than specified
    invariant update
        forall acc: address where exists<Account>(acc):
            old(bal(acc)) - bal(acc) <= MAX_DECREASE;
}
```

About the Diem Payment Network

The Diem Payment Network (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.
- DPN has accumulated 11,813 lines of code, including 8,372 lines of specs, making a 7:5 code-to-spec ratio.

About the Diem Payment Network

The Diem Payment Network (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.
- DPN has accumulated 11,813 lines of code, including 8,372 lines of specs, making a 7:5 code-to-spec ratio.

Move Prover formally verifies DPN in a few minutes. In fact, the Prover is enforced in continuous integration—every change of Move code needs to pass the verification.

Motivation

Formal verification is expensive,

Motivation

Formal verification is expensive, so, how to help DPN developers write and review specifications in a way that maximizes return on investment (ROI)?

Motivation

Formal verification is expensive, so, how to help DPN developers write and review specifications in a way that maximizes return on investment (ROI)?

(A mirror side of this question from a user's perspective):
if you see a codebase carries a formally verified "stamp" (sometimes even advertises $x : y$ spec-to-code ratio), how much can you trust that "stamp"?

What is specification (spec)?

What is specification (spec)?

From my colleagues in the Move team:

- Specs are contracts between implementation and functionalities
- Specs are extensions to the type system
- Specs are definitions of high-level state machines

What is specification (spec)?

From my colleagues in the Move team:

- Specs are contracts between implementation and functionalities
- Specs are extensions to the type system
- Specs are definitions of high-level state machines

One question remaining: if I am reviewing some spec, how do I know whether it is useful?

Example 1: code

```
1  const MIN_BALANCE: u64 = 5;
2  const MAX_BALANCE: u64 = 10000;
3
4  struct Account has key {
5      balance: u64
6  }
7
8  public entry fun transfer(
9      from: address, into: address, amount: u64
10 ) acquires Account {
11     assert!(from != into);
12
13     // withdraw
14     let bal = &mut borrow_global_mut<Account>(from).balance;
15     assert!(*bal - MIN_BALANCE >= amount);
16     *bal = *bal - amount;
17
18     // deposit
19     let bal = &mut borrow_global_mut<Account>(into).balance;
20     assert!(MAX_BALANCE - *bal >= amount);
21     *bal = *bal + amount;
22 }
```

Example 1: spec

```
1  // syntactic sugar to improve spec readability
2  spec fun balance(addr: address): num {
3      global<Account>(addr).balance
4  }
5
6  spec transfer {
7      requires true;
8      aborts_if from == into;
9      aborts_if !exists<Account>(from);
10     aborts_if !exists<Account>(into);
11     aborts_if balance(from) - amount < MIN_BALANCE;
12     aborts_if balance(into) + amount > MAX_BALANCE;
13     ensures balance(from) == old(balance(from)) - amount;
14     ensures balance(into) == old(balance(into)) + amount;
15 }
```

Example 1: spec

```
1  // syntactic sugar to improve spec readability
2  spec fun balance(addr: address): num {
3      global<Account>(addr).balance
4  }
5
6  spec transfer {
7      requires true;
8      aborts_if from == into;
9      aborts_if !exists<Account>(from);
10     aborts_if !exists<Account>(into);
11     aborts_if balance(from) - amount < MIN_BALANCE;
12     aborts_if balance(into) + amount > MAX_BALANCE;
13     ensures balance(from) == old(balance(from)) - amount;
14     ensures balance(into) == old(balance(into)) + amount;
15 }
```

This spec is useful because it contracts a user-facing interface (`public entry fun`).

Example 2

```
1  fun encrypt(key: AESKey, data: vector<u8>): vector<u8> {
2    // ... implementation details redacted ...
3  }
4  spec encrypt {
5    ensures [abstract] result == spec_encrypt(key, data);
6  }
7
8  fun decrypt(key: AESKey, data: vector<u8>): vector<u8> {
9    // ... implementation details redacted ...
10 }
11 spec decrypt {
12   ensures [abstract] result == spec_decrypt(key, data);
13 }
14
15 // axiomatized uninterpreted functions
16 spec fun spec_encrypt(key: AESKey, data: vector<u8>): vector<u8>;
17 spec fun spec_decrypt(key: AESKey, data: vector<u8>): vector<u8>;
18
19 axiom forall key: AESKey, data: vector<u8>:
20   spec_decrypt(key, spec_encrypt(key, data)) == data;
```

Example 3

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

```
spec error_code {
  ensures [concrete] result == category + (reason * 256);
  ensures [abstract] result == spec_error(category, reason);
}

// axiomatized uninterpreted function
spec fun spec_error(category: u8, reason: u8): u64;

// declarative definition for the `spec_error` function
axiom forall c1: u8, c2: u8, r1: u8, r2: u8:
  (c1 != c2 || r1 != r2) ==>
    spec_error(c1, r1) != spec_error(c2, r2);
```

Example 3

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

```
spec error_code {
  ensures [concrete] result == category + (reason * 256);
  ensures [abstract] result == spec_error(category, reason);
}

// axiomatized uninterpreted function
spec fun spec_error(category: u8, reason: u8): u64;

// declarative definition for the `spec_error` function
axiom forall c1: u8, c2: u8, r1: u8, r2: u8:
  (c1 != c2 || r1 != r2) ==>
    spec_error(c1, r1) != spec_error(c2, r2);
```

These specs are useful because they summarize functions with abstraction, and the purpose of abstraction is to enable compositional verification of other specs.

Example 4

```
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;
}
```

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

Example 4: reflection

The specs in this example is not very useful from a formal verification perspective as the specs passively shadow the imperative implementation with the goal of making side effects (e.g., changes to global states) explicit.

Example 4: reflection

The specs in this example is not very useful from a formal verification perspective as the specs passively shadow the imperative implementation with the goal of making side effects (e.g., changes to global states) explicit.

Essentially, this spec is only trying to bridge the gap between an imperative language (e.g., with memory operations) and a pure language (i.e., the specification language).

Example 4: reflection

The specs in this example is not very useful from a formal verification perspective as the specs passively shadow the imperative implementation with the goal of making side effects (e.g., changes to global states) explicit.

Essentially, this spec is only trying to bridge the gap between an imperative language (e.g., with memory operations) and a pure language (i.e., the specification language). Ideally, such lifting should be done automatically.

Because of this, the spec-to-code ratio might not be a very good indicator of how extensive the codebase is formally specified.

Example 4: reflection

The specs in this example is not very useful from a formal verification perspective as the specs passively shadow the imperative implementation with the goal of making side effects (e.g., changes to global states) explicit.

Essentially, this spec is only trying to bridge the gap between an imperative language (e.g., with memory operations) and a pure language (i.e., the specification language). Ideally, such lifting should be done automatically.

Because of this, the spec-to-code ratio might not be a very good indicator of how extensive the codebase is formally specified.

Maybe we should seek to develop more advanced metrics, such as spec coverage (similar to how code coverage is used to measure the quality of test suites).

Operational advice for Move unit specs

Unit specs (i.e., function pre- and post-conditions) 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 automated proof for other abstracting specs (i.e., with `[abstract]` annotated clauses).

Move data invariants examples

If the unit specs are considered as extensions to function signature, there is also a way to directly extend the data type in Move, which is called data invariants (DI).

Move data invariants examples

If the unit specs are considered as extensions to function signature, there is also a way to directly extend the data type in Move, which is called data invariants (DI).

```
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 }
```

Move data invariants examples

If the unit specs are considered as extensions to function signature, there is also a way to directly extend the data type in Move, which is called data invariants (DI).

```
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 }
```

```
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 }
```

Operational advice for Move data invariants

Data invariants 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, or
- can be persisted into the blockchain state (as marked by the `key` or `store` ability modifier).

Basic view of blockchain states

Smart contracts evolve around **blockchain states** (a.k.a., *global state* or *global storage* in Move).

Basic view of blockchain states

Smart contracts evolve around **blockchain states** (a.k.a., *global state* or *global storage* in Move).

More specifically, executing a transaction t evolves the blockchain from an old state to a new one:

$$\mathbb{G}_i \xrightarrow{t} \mathbb{G}_{i+1}$$

Basic view of blockchain states

Smart contracts evolve around **blockchain states** (a.k.a., *global state* or *global storage* in Move).

More specifically, executing a transaction t evolves the blockchain from an old state to a new one:

$$\mathbb{G}_i \xrightarrow{t} \mathbb{G}_{i+1}$$

Executing a series of transactions, $t_1, t_2, \ldots, t_n$ evolves global states consecutively:

$$\mathbb{G}_0 \xrightarrow{t_1} \mathbb{G}_1 \xrightarrow{t_2} \mathbb{G}_2 \xrightarrow{t_3} \ldots \xrightarrow{t_n} \mathbb{G}_n$$

Basic view of blockchain states

Smart contracts evolve around **blockchain states** (a.k.a., *global state* or *global storage* in Move).

More specifically, executing a transaction t evolves the blockchain from an old state to a new one:

$$\mathbb{G}_i \xrightarrow{t} \mathbb{G}_{i+1}$$

Executing a series of transactions, $t_1, t_2, \ldots, t_n$ evolves global states consecutively:

$$\mathbb{G}_0 \xrightarrow{t_1} \mathbb{G}_1 \xrightarrow{t_2} \mathbb{G}_2 \xrightarrow{t_3} \ldots \xrightarrow{t_n} \mathbb{G}_n$$

Q: How to represent the global state $\mathbb{G}$?

Global state in Move

In Move, the global state $\mathbb{G}$ is represented as a map (or table or dictionary). More precisely, $\mathbb{G}$ is a map of

$$(\texttt{address}, \texttt{T}) \rightarrow \text{optional value of type } \texttt{T}$$

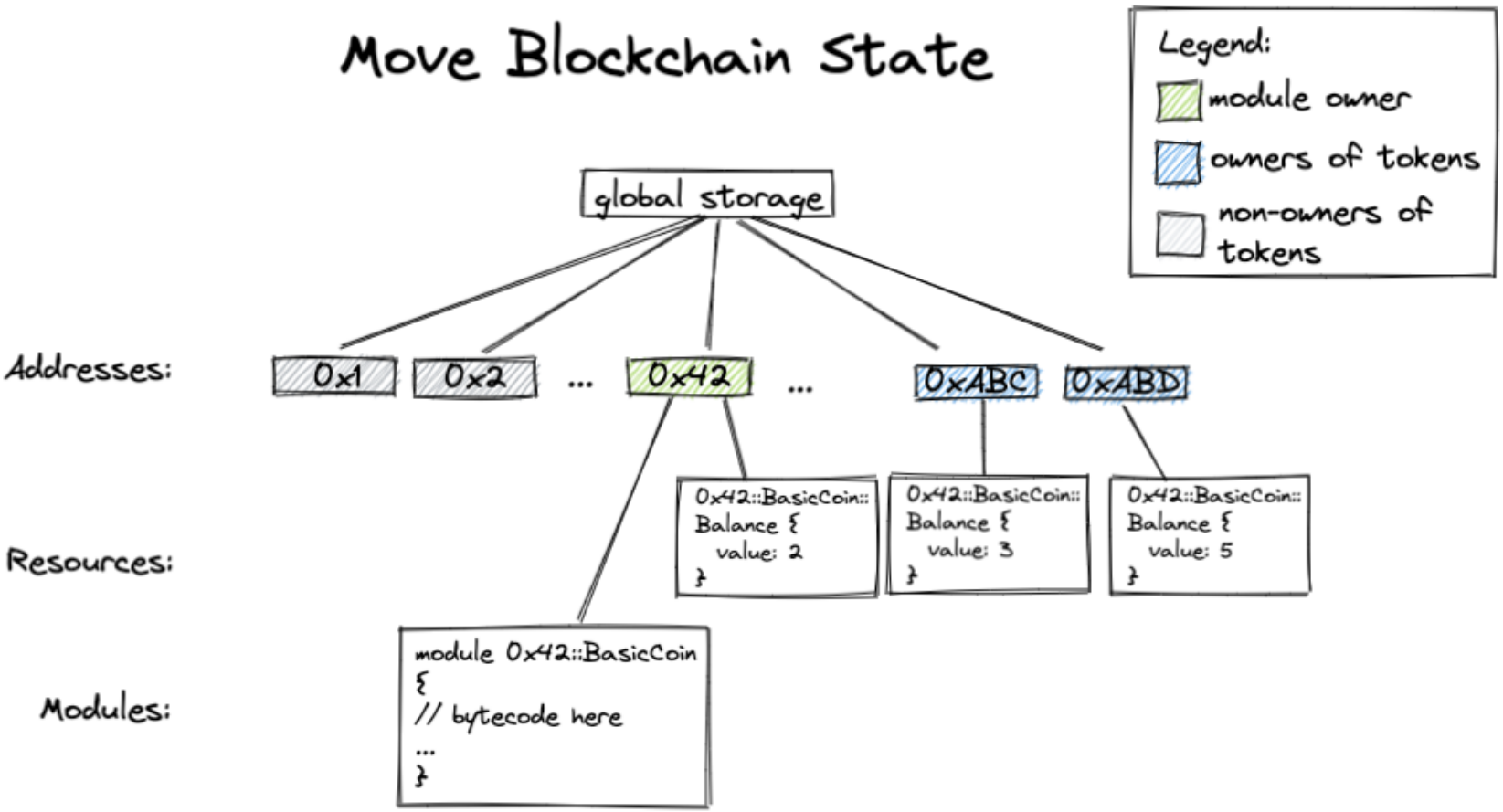
Global state in Move

In Move, the global state $\mathbb{G}$ is represented as a map (or table or dictionary). More precisely, $\mathbb{G}$ is a map of

$$(\texttt{address}, \texttt{T}) \rightarrow \text{optional value of type } \texttt{T}$$

Without initialization, all (*, *) $\rightarrow$ `None`.

Illustration of the global state



Operations on the global state in Move

Basic operations on a `map<K, V>` type:

- `exists(k: K): bool`
- `put(k: K, v: V)`
- `del(k: K)`
- `get(k: K): V`

Operations on the global state in Move

Basic operations on a `map<K, V>` type:

- `exists(k: K): bool`
- `put(k: K, v: V)`
- `del(k: K)`
- `get(k: K): V`

Operations on the global state: $\mathbb{G}$`<(address, Tag<T>), Optional<T>>`:

- `exists<T>(address): bool`
- `move_to<T>(address, t: T)`
- `move_from<T>(address): T`
- `borrow_global<T>(address): &T`

Example: specs in global invariants form

```
1  // invariants on global states (SI)
2  invariant [state]
3      forall addr: address where exists<Account>(addr):
4          MIN_BALANCE <= balance(addr) <= MAX_BALANCE;
5
6  // invariants on global state transitions (TI)
7  invariant [update]
8      exists from: address: (
9          exists<Account>(from) &&
0          old(exists<Account>(from)) &&
1          old(balance(from)) > balance(from)
2      ) ==> exists into: address: (
3          exists<Account>(into) &&
4          old(exists<Account>(into)) &&
5          balance(into) - old(balance(into)) == old(balance(from)) - balance(from) &&
6          forall rest: address where rest != from && rest != into:
7              !old(exists<Account>(rest)) ==> !exists<Account>(rest) &&
8              old(exists<Account>(rest)) ==> (
9                  exists<Account>(rest) &&
0                  balance(rest) == old(balance(rest))
1              )
2      );
```

Example: specs with reflection

These global invariants are very powerful specification as they directly specify against the global states $\mathbb{G}$

$$(\texttt{address}, \texttt{T}) \rightarrow \text{optional value of type } \texttt{T}$$

... and are completely oblivious of the actual implementation.

Example: code that conforms to the global invariants

```
1  const MIN_BALANCE: u64 = 5;
2  const MAX_BALANCE: u64 = 10000;
3
4  struct Account has key {
5      balance: u64
6  }
7
8  public entry fun transfer(
9      from: address, into: address, amount: u64
10 ) acquires Account {
11     assert!(from != into);
12
13     // withdraw
14     let bal = &mut borrow_global_mut<Account>(from).balance;
15     assert!(*bal - MIN_BALANCE >= amount);
16     *bal = *bal - amount;
17
18     // deposit
19     let bal = &mut borrow_global_mut<Account>(into).balance;
20     assert!(MAX_BALANCE - *bal >= amount);
21     *bal = *bal + amount;
22 }
```

Practical challenges

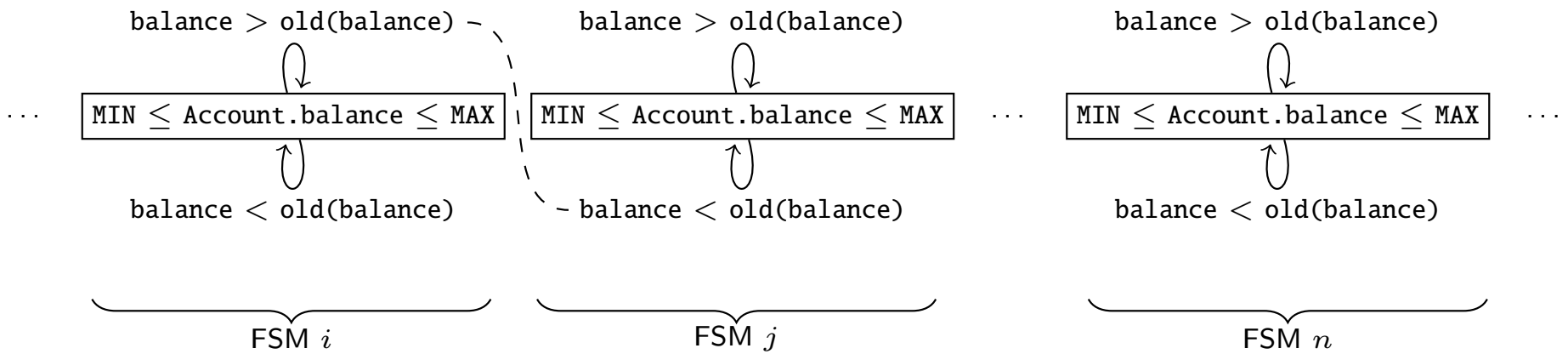
However, although global invariants are powerful, we found in practice that engineers try to avoid them, perhaps because they are too abstract (too powerful).

Practical challenges

However, although global invariants are powerful, we found in practice that engineers try to avoid them, perhaps because they are too abstract (too powerful).

The global invariants attempt to formally define an infinite number of communicating finite-state machines (CFSM) that collectively operate over a common $\mathbb{G}$.

CFSM of the example



Interpretation of the graph:

- Each $\mathbb{G}$[(*,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.
- 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.

Conclusion

Specs that merely shadow the implementation are not very useful from a formal verification perspective.

Specs that are abstracting, i.e., by locking in requirements or intention should be prioritized.

Conclusion

Specs that merely shadow the implementation are not very useful from a formal verification perspective.

Specs that are abstracting, i.e., by locking in requirements or intention should be prioritized.

The spec language should highlight different styles of specs sufficiently such that all stakeholders (e.g., auditors) can spend more time on the abstracting specs.

$\langle$ **End** $\rangle$