# The Move Prover

Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L. Dill

Presentation by Daniel Msika and Tom Ben-Dor

# Motivation

- The Move Prover is a formal verification tool for the Move programming language on the Diem (Libra) blockchain.
- It works by anotating the code with “specifications”, which are written in a seprate language and describe the desireable behavior of the program, similar to an interface or testing.
- Developed by Meta.
- Runs “just slower than a compiler” and is meant to be used in checks every time new code is pushed.

# The Diem (Libra) Blockchain

- The Libra blockchain is a permissioned blockchain that is designed to be secure, scalable, and compliant with global financial regulations.
- It is backed by many leading financial institutions and technology companies, including Facebook, Mastercard, Visa, and PayPal.
- The Libra blockchain is designed to support a wide range of digital currency applications, including payments and financial services. It is also designed to be interoperable with other blockchains, which will allow for the transfer of value across different platforms.

# The Move Programming Language

- Executable bytecode language (Java like) for writing smart contracts and custom transaction logic. Minimal
- It has Primitives, Records, Vectors, References.
- To interact with the blockchain, a programmer can write a Move transaction script (like a main script), which is then packaged into a transaction which is executed by validators in Libra blockchain.
- If a transaction fails - everything is reverted, and no harm is caused.

# Find the danger

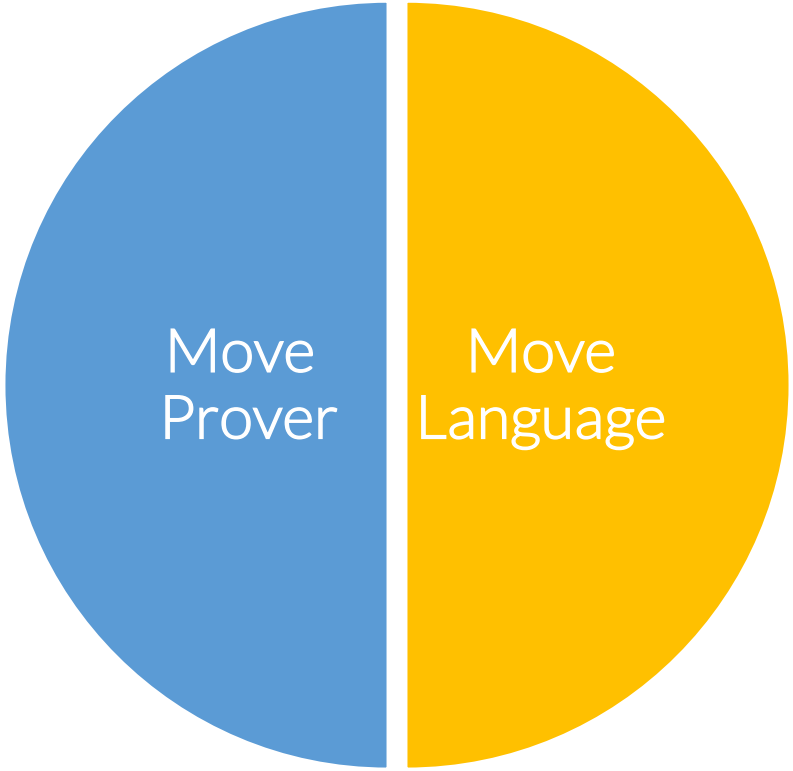
```
module LibraCoin {
  resource struct T { value: u64 }

  public fun join(coin: &mut LibraCoin::T, to_consume: LibraCoin::T) {
    let  T { value } = to_consume; // MoveLoc(1); Unpack
    let c_value_ref = &mut coin.value; // MoveLoc(0); MutBorrowField<value>; StLoc(0)
    *c_value_ref = *c_value_ref + value; // CopyLoc(0); ReadRef; Add; MoveLoc(0); WriteRef
    return; // Ret
  }
}
```

# The Move project (language + verifier)

- We already have languages for writing contracts, why do we need Move?
- Move has some cool features:
  - It does not depend on hard-coded platform conventions, making it cross-platform.
  - The move compiler incorprates type checking designed for digital assets.
  - Basic verification is done both during compilation and at runtime by the bytecode verifier.
  - Safe arithmetics.

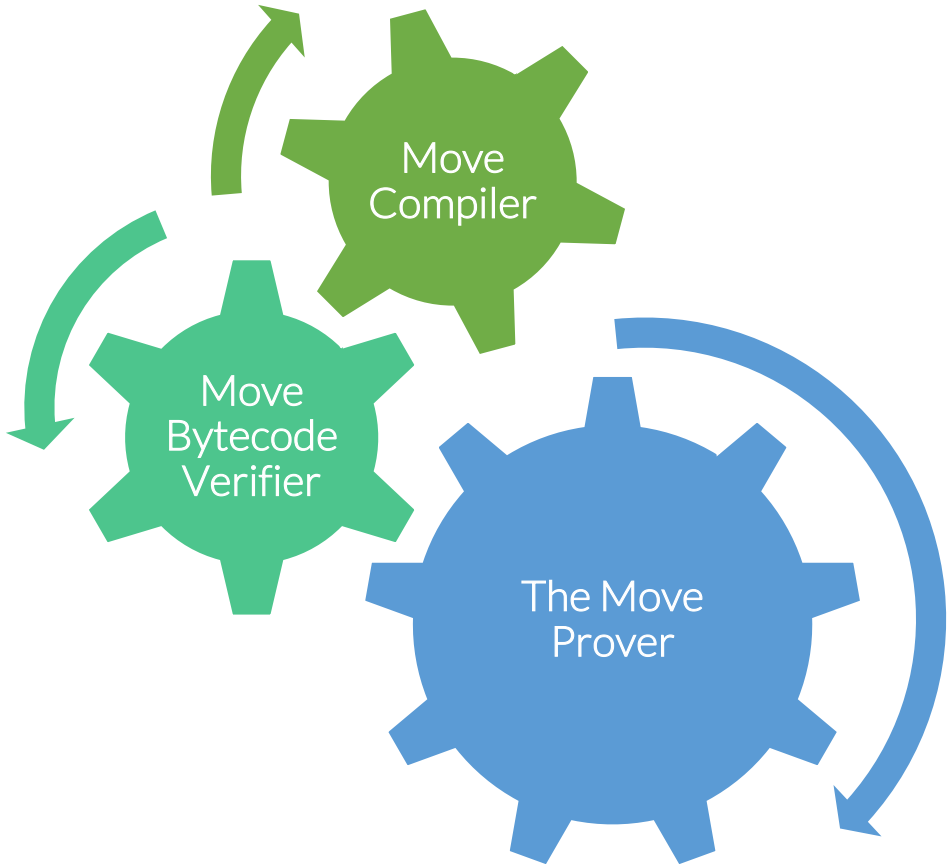
# The Move project (language + verifier)

- Move was meant to be used alongside with The Move Prover!
- It's a minimal language, no:
  - Concurrency.
  - I/O (except for transactions and global storage).
  - Dynamic dispatch.
  - Non-deterministic functions.
  - Exceptions.
- All these pose great challenge in verification, therefore omitted in Move.



# Move verification process

- In Move we have three mechanisms to verify the code and avoid bugs:
  - **Compilation:** Type checking, ownership
    - Protects the programmers from themselves.
  - **Runtime:** Bytecode verifier – Malicious code, invalid bytecode
    - Protects the programmers from other programmerts.
  - **The Move Prover:** A formal verification tool that can be used to verify the correctness of Move programs.
    - Can check for predefined safety properties.



# The bytecode verifier

- Not an actual verifier (FVT).
- Runs a static analysis on the bytecode in polynomial time, so the compiler is out of the trust-chain.
- Malicious programmers can write bytecode without going through the compiler.
- Being run before execution.
- The Prover relies on these checks.

# Formal Verification

- Proving a program doesn’t violate given constraints.
- Formal verification achives this by considering **every** state and **every** input.
- The goal is to to prove correctness of a program, which is defined by the programmer, like testing.
- Not meant for finding bugs.

# Specifications

- Annotations in the Move source code (can also reside in a separate file).
- Like interfaces (Java) or testing, specifications should be met with matching implementations.
  - We expect the prover to alert when it's not the case.
- Writing good specifications is indispensable.
  - Specifications language is designed to be as close as possible to pure logic.
  - High-level language.
- Never affecting the execution of the module.

# Types of specifications

- **Pre-conditions:** Verify the state before execution of certain function.
  - Using the *requires* specification:

```
spec increment {
    requires global<Counter>(a).value < 255;
}
```

- **Post-conditions:** Verify the state after execution of certain function.
  - Using the *ensures* specification:

```
spec increment {
   ensures global<Counter>(a).value == old(global<Counter>(a)).value + 1;
}
```

# Types of specifications

- A new condition specifying when a function aborts.
  - Verify the function aborts when it should (and only then).

```
spec divide {
    aborts_if divisor == 0;
}
```

- An abort is not an error!
  - Most formal verifiers treat any run-time error as a bug.
  - In Move, aborts is just another way to mark the transaction as invalid.
  - Aborts happed automatically during overflow, division by zero, etc.
- Multiple `aborts_if` specifications will translate as a disjunction.
- Exists<M::T>(A)

# Types of specifications

- Invariant conditions:
  - Verify a condition holds before and after function execution:

```
spec increment {
    invariant global<Counter>(a).value < 128;
}
```

  - Struct invariant:

```
spec Counter {
    invariant value < 128;
}
```

  - Global invariant:

```
module addr::M {
    invariant forall a: addr where exists<Counter>(a):
                            global<Counter>(a).value > 0;
}
```

# Global Invariant example

```
fun decrement_ad(addr: address) acquires Counter {
    let counter = borrow_global_mut<Counter>(addr);
    let new_value = counter.value - 1; // Will not abort because counter.value > 0
    *counter.value = new_value; // Fails verification since value can drop to zero
}
```

# Specifications example

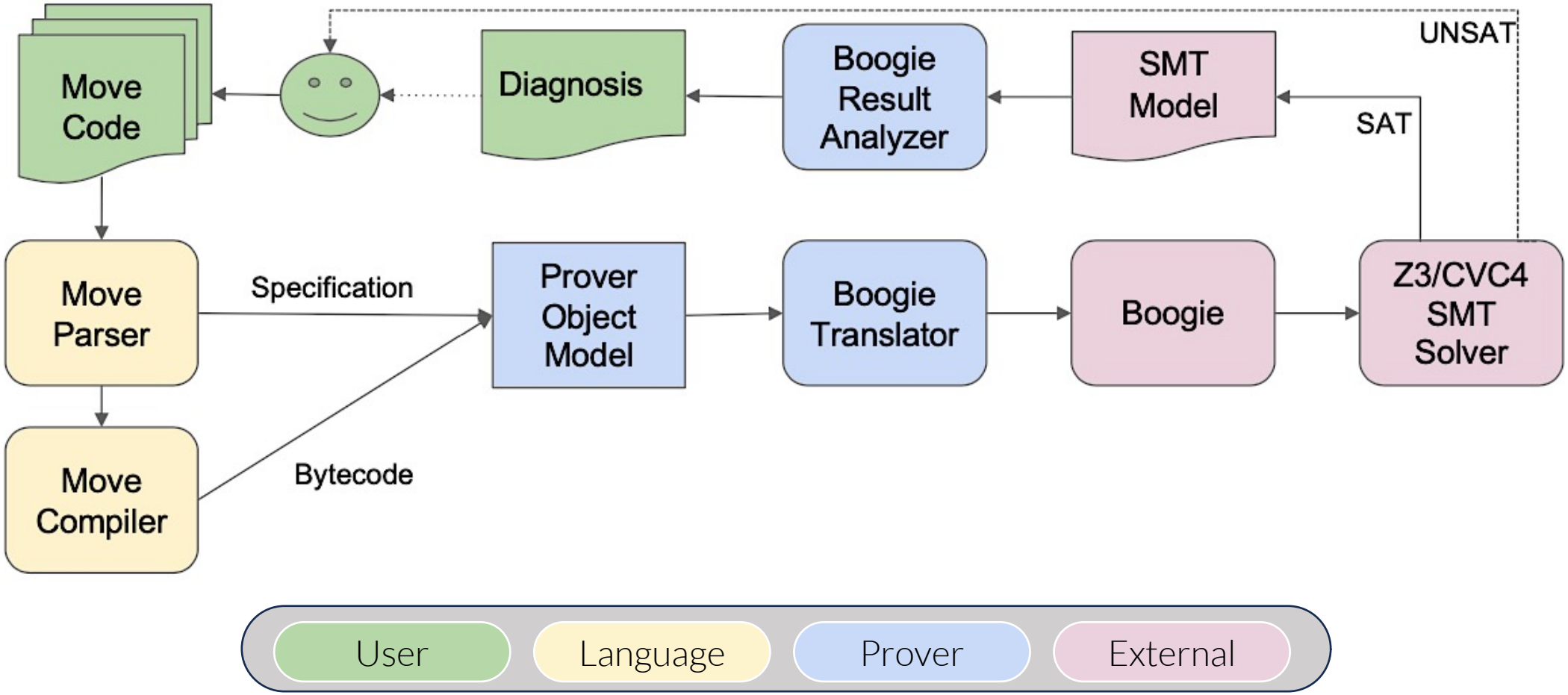
```
public fun pay_from_sender(payee: address, amount: u64) acquires T
{
  Transaction::assert(payee != Transaction::sender(), 1);  // new!

  if (!exists<T>(payee)) {
    Self::create_account(payee);
  };
  Self::deposit(
    payee,
    Self::withdraw_from_sender(amount),
  );
}
```

```
spec fun pay_from_sender {
//  ... omitted aborts_ifs ...
  aborts_if amount == 0;
  aborts_if global<T>(sender()).balance.value < amount;
  ensures exists<T>(payee);
  ensures global<T>(sender()).balance.value
      == old(global<T>(sender()).balance.value) - amount;
}
```

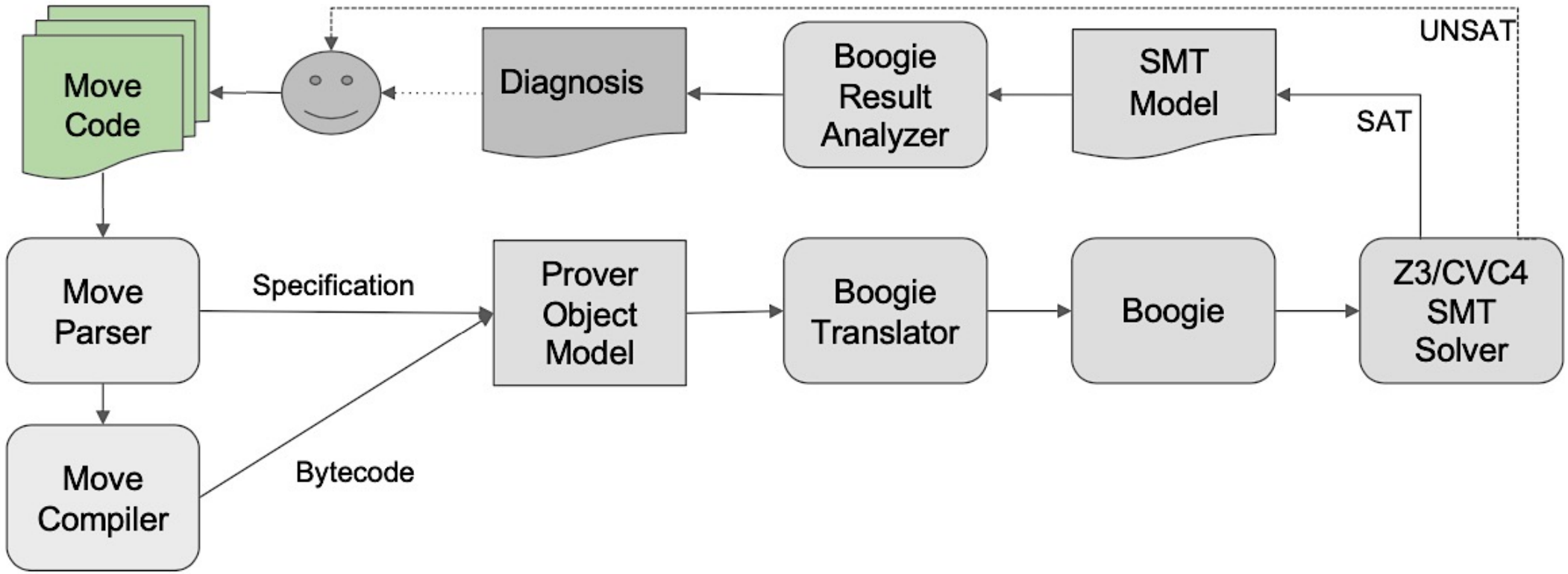
# Specifications example

```
fun verify_reverse<Element>(v: &mut vector<Element>) {
    let vlen = vector::length(v);

    if (vlen == 0) return ();
    let front_index = 0;
    let back_index = vlen -1;

    while ({
        spec {
            assert front_index + back_index == vlen - 1;
            assert forall i in 0..front_index: v[i] == old(v)[vlen-1-i];
            assert forall i in 0..front_index: v[vlen-1-i] == old(v)[i];
            assert forall j in front_index..back_index+1: v[j] == old(v)[j];
            assert len(v) == vlen;
        };
        (front_index < back_index)
    }) {
        vector::swap(v, front_index, back_index);
        front_index = front_index + 1;
        back_index = back_index - 1;
    };

}

spec verify_reverse {
    aborts_if false;
    ensures forall i in 0..len(v): v[i] == old(v)[len(v)-1-i];
}
```

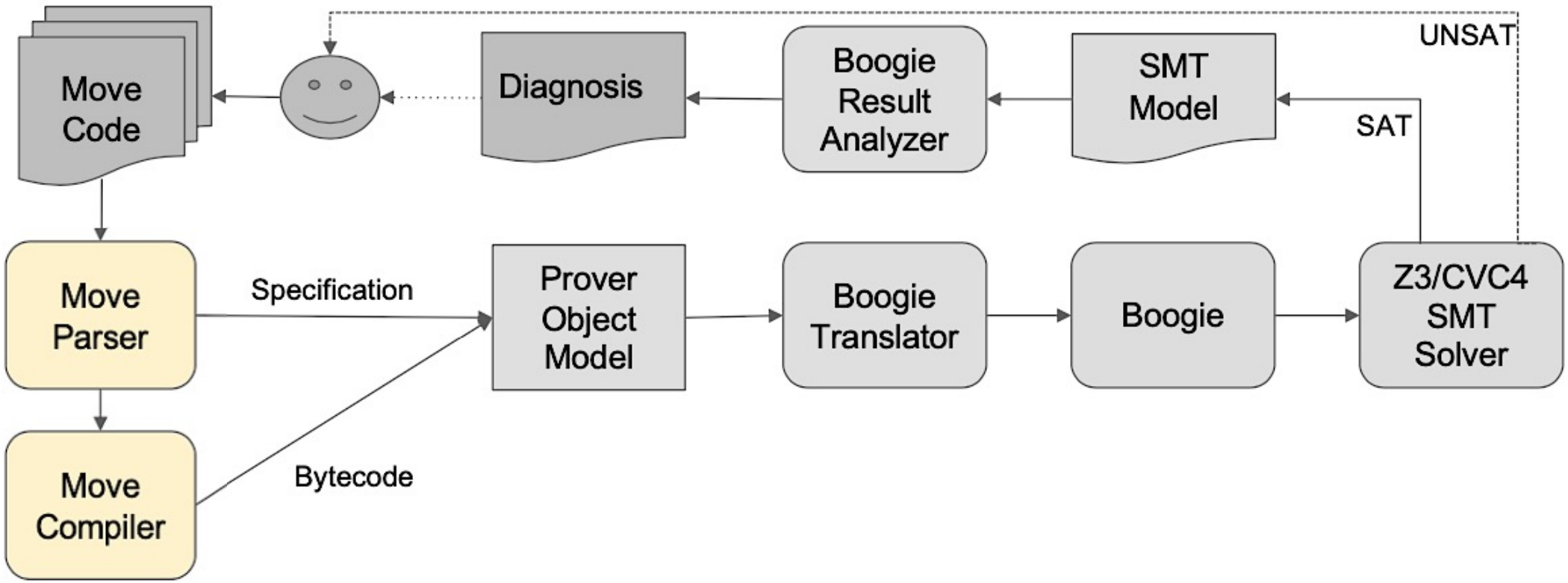
# Move Prover architecture



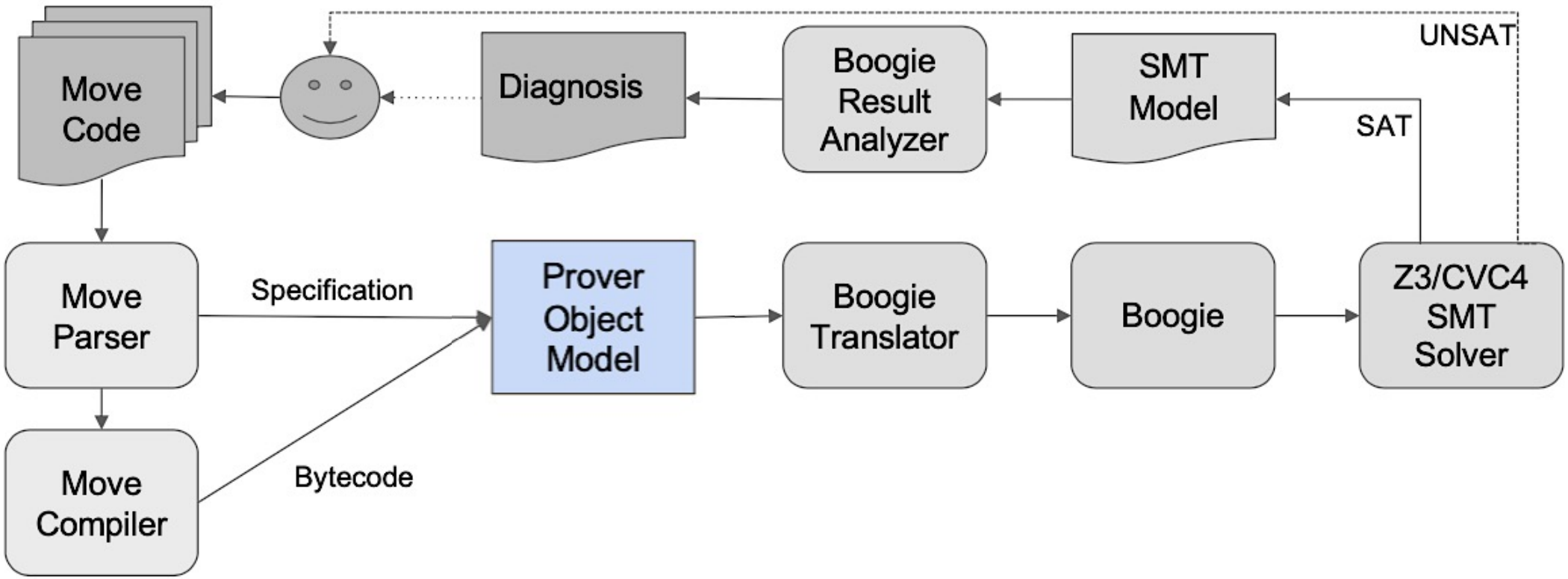
# Move Prover architecture



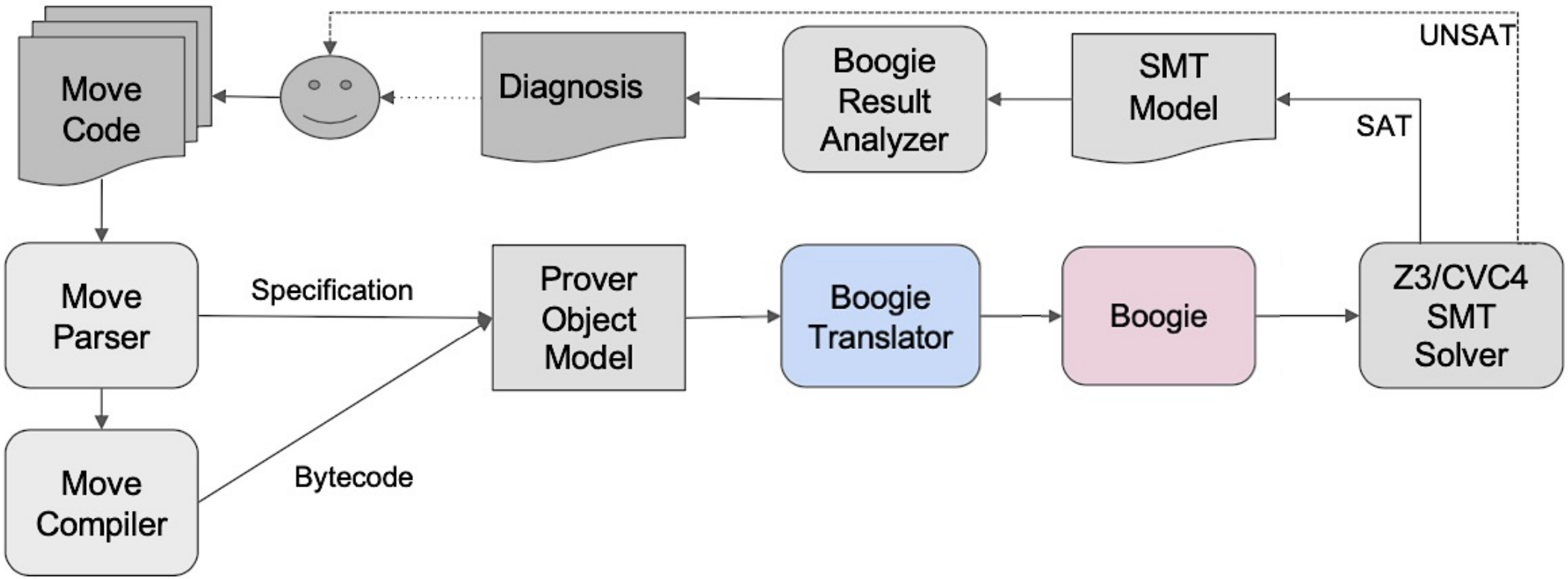
# Move Prover architecture



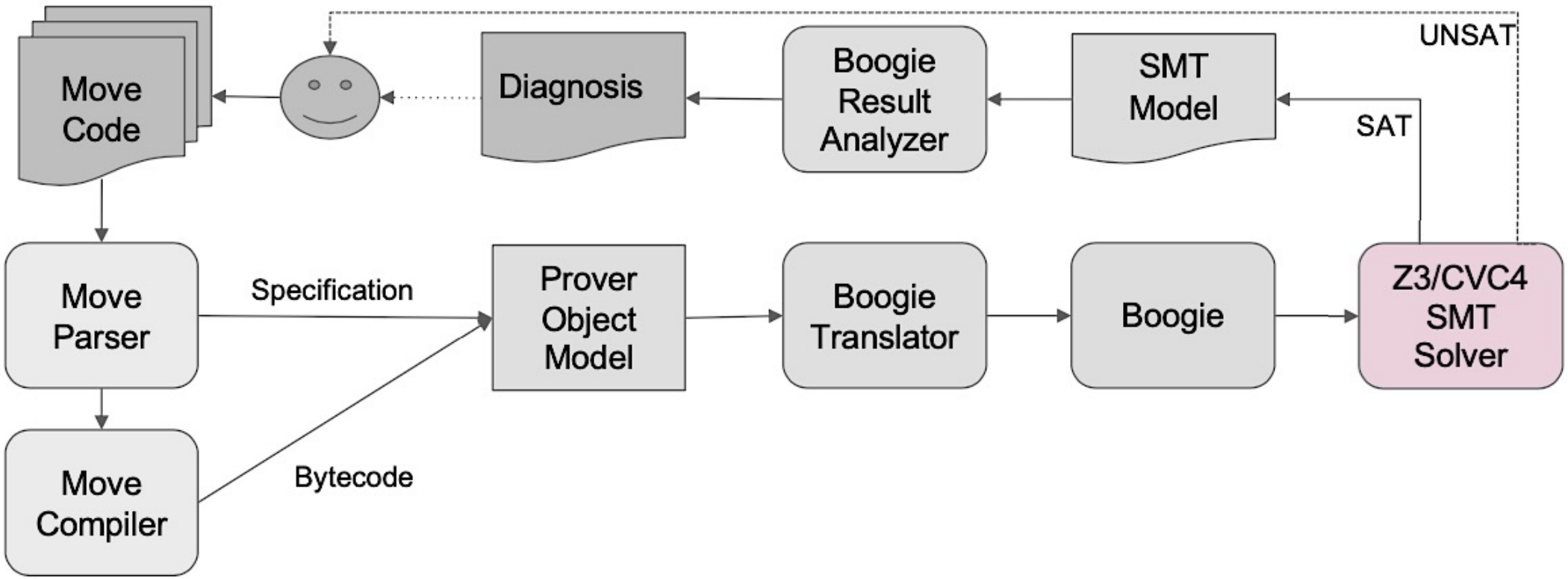
# Move architecture



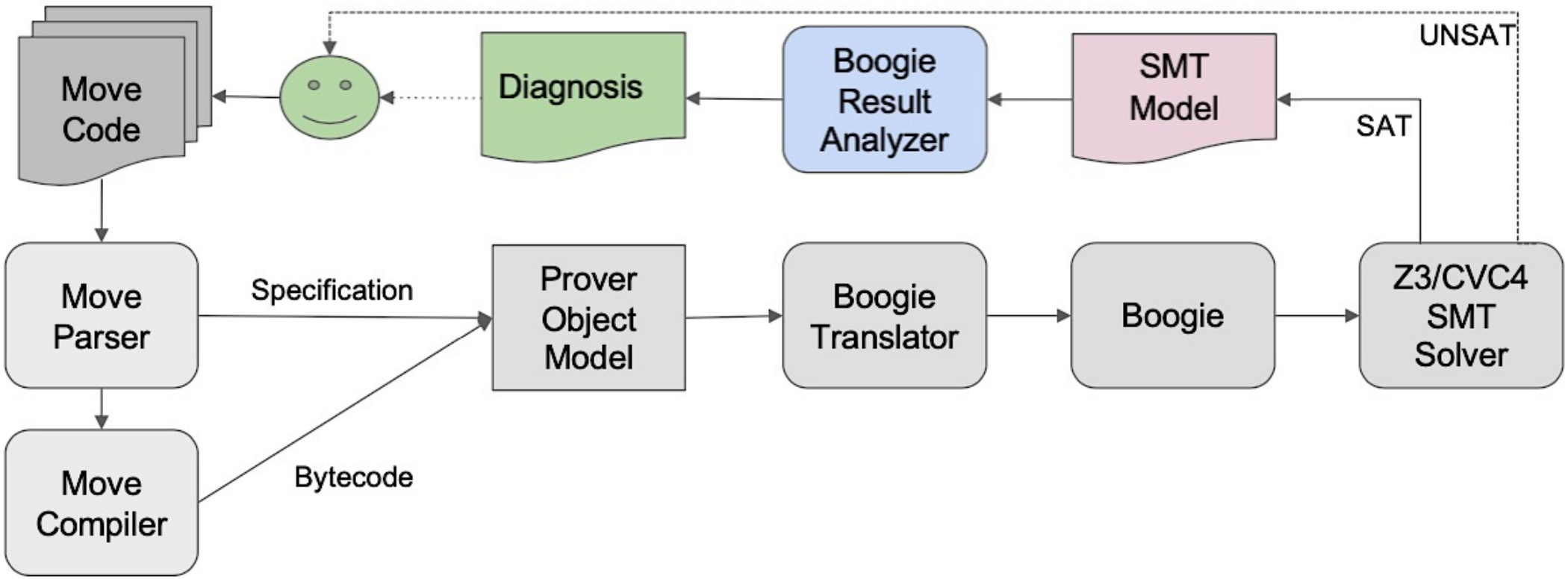
# Move architecture



# Move architecture



# Move architecture



# Boogie

- Boogie IVL is an intermediate language for verification that was developed by Microsoft.
- Supports:
  - Local / Global variable.
  - Assignment.
  - Branching (if, else).
  - Loops.
  - Procedures (functions).
  - Simple data structures (primitives, arrays).
  - Pre / Post conditions.
- Not executable – serve as input to the Boogie Verification System.
- If all of the verification conditions hold, then each procedure ensures its post-conditions, under the assumption that its pre-conditions hold.

# Boogie translation

- Only supports variable types that are supported by SMT solvers, making it fairly simple to translate into SMT formulas.
- It is common for a verifier to model the source code in Boogie, then translate and simply send to the Boogie Verification System.
- How can we translate Move Bytecode to Boogie?
  - Removing stack operations (rewriting as local variables).
  - Each bytecode instruction is implemented in Boogie as a procedure.
  - Bytecode program → a sequence of procedure calls.

# Boogie example

```
type {:datatype} Value;
function {:constructor} Boolean(b: bool): Value;
function {:constructor} Integer(i: int): Value;
function {:constructor} Address(a: int): Value;
function {:constructor} Vector(v: ValueArray): Value;

type {:datatype} ValueArray;
function {:constructor} ValueArray(v: [int]Value, l: int): ValueArray;

type {:datatype} Path;
function {:constructor} Path(p: [int]int, size: int): Path;

type {:datatype} Location;
function {:constructor} Global(t: TypeValue, a: int): Location;
function {:constructor} Local(i: int): Location;

type {:datatype} Reference;
function {:constructor} Reference(l: Location, p: Path): Reference;

type {:datatype} Memory;
function {:constructor} Memory(domain: [Location]bool, contents: [Location]Value): Memory;
var $m : Memory;
```

# Boogie verification system

- Generates an SMT formula in the SMT-LIB format from the Boogie code.
- Then it can be checked using an SMT solver (Z3, CVC4).
- Two possible results:
  - UNSAT – specifications hold.
  - SAT – at least one specification does not hold.
- In case the verification failed, in order to present the user with meaningful errors (including line numbers), reversing the SMT solver error back to Move source code is done.
  - This ability of MVP is another example of why it has an advantage on other FVTs.

# Evaluation

- The Libra and LibraAccount modules contain almost :
- 1300 lines -> 14K lines in Boogie files -> 52K lines in SMT files.
- The move prover was able to prove all the specifications in under a minute.
- The move prover is beginning to be useful to verify contracts in production.
- No serious bugs (yet...).

| Move Module | LoC | Boogie LoC | SMT LoC | Functions | Verified | Runtime |
|---|---|---|---|---|---|---|
| Libra | 420 | 3875 | 11,688 | 25 | 25 | 2.99 s |
| LibraAccount | 867 | 10,362 | 40293 | 38 | 34 | 46.66 s |

# Live Demo

# Thank you for the attention

- Questions ?