MEng Individual Project

Imperial College London

Department of Computing

Automatic Formal Verification of Flint Smart Contracts

Supervisor:
Professor Susan Eisenbach

Author:
Ioannis Gabrielides

Second Marker:
Dr Robert Chatley

June 16, 2019

Abstract

Smart contract authors must take care when writing their software. Ethereum smart contracts are immutable once deployed, they cannot be patched or updated, so it is imperative that they are thoroughly checked for bugs before deployment. There exist many tools and systems which attempt to minimise bugs in smart contracts. For example, systems which search for known security flaws, or unit testing frameworks, or formal verification, are all current approaches to reduce bugs in smart contracts. However, only formal verification provides the strong guarantees of correctness, required by smart contract authors.

We present an automatic formal verification system, integrated into the compiler of, the smart contract programming language, Flint. We show that our verification system, which supports a subset of Flint smart contracts, can verify smart contracts against user-supplied functional and holistic specifications. Our verification system also detects potential array-out-of-bounds, divide-by-zero and unreachability errors, to prevent a class of immutability bugs.

Acknowledgements

I would like to thank the following people for their insight and support throughout this project.

- I would first like to thank my supervisors Professor Susan Eisenbach and Dr Robert Chatley, as well as Professor Sophia Drossopoulou, for our weekly meetings. Their feedback and patience was invaluable.
- I would like to thank Zubair, for our many interesting discussions and his friendship.
- I would also like to thank my friends Tom, Dennis, Sarah and Kieran, without whom I would have lost my sanity a long time ago.
- Finally I would like to thank my family for their unwavering support throughout my time at Imperial. This would not have been possible without them.

Contents

Chapter 1

Introduction

A large number of Solidity[1], [2] smart contracts contain bugs[3]. This is particularly problematic when smart contracts are deployed on immutable blockchains, such as Ethereum, where they cannot be updated or patched once deployed. Ethereum smart contracts typically hold the digital currency *Ether*, which is put at risk when held within buggy smart contracts.

Ethereum has a history of attackers exploiting insecure Solidity smart contracts, draining them of their Wei. High profile attacks, such as the DAO[4] and Parity Wallet[5] attacks, have lost Ethereum users approximately $70 million. Clearly, bugs in smart contracts have a negative affect on Ethereum users, but they can also impact the reputation of the larger blockchain community.

The vulnerabilities responsible for many Ethereum smart contracts, are due to the design choices of the Solidity programming language. Solidity is a JavaScript inspired language, designed with the intention to make writing smart contracts accessible to a broad audience. However, this approach has resulted in the use of a variety of unsafe patterns, and can make reasoning about smart contracts difficult. This seems at odds to what the goals of a smart contract language should be: making it hard to introduce bugs; and difficult to lose money. These goals require it to be easy to reason about the behaviour of a smart contract. As a result, Franklin Schrans supervised by Professor Susan Eisenbach and Professor Sophia Drossopoulou, introduced Flint[6], a Swift based programming language aimed directly at making programming smart contracts safer and easier to reason about. Our work here targets the Flint language, we feel that Flint's approach to smart contracts is safer and represents the future of smart contract development.

To mitigate bugs, a suite of tools and approaches have been developed for Solidity smart contracts. Currently, the main method which smart contract authors prevent bugs is through unit testing. The Truffle Suite[7] is a popular testing framework which allows contract authors to write unit tests for Solidity smart contracts. However, unit testing does not provide the strong guarantees which are required by smart contracts. They do not verify that the contract exhibits the specified behaviour, for all execution paths, only the execution paths explored by the tests suite are checked. Formal verification provides a stronger guarantee, checking the behaviour of the contract for all execution paths, that the contract exhibits the correct behaviour. However, formal verification is a time consuming process which requires expertise and complete knowledge of the semantics of the language. Automatic formal verification is a technique which provides formal proofs of correctness, with the same guarantees as manual formal verification, but the verification is performed by an automatic verification system. This approach is less time consuming and more accessible to smart contract authors, and could provide compile-time verification results and run-time performance benefits.

We aim to extend the work done by Schrans et al by introducing automatic formal verification of Flint smart contracts to the compiler. Verifying software is typically a time consuming, tedious and manual process, and must be re-done after any change to the software's implementation. We aim to make Flint smart contract verification fast, automatic and meaningful for smart contract authors. Formal verification can be used to detect bugs and avoid potential vulnerabilities during the development of the smart contract. The goal for this project is to develop a system, within the Flint compiler, which automatically verifies Flint smart contracts against user-supplied specifications.

This project introduces a verification system to Flint, powered by formal methods, which aim to improve the quality of smart contracts written in Flint. We attempt to design verification features which detect and inform the contract author of any dangerous patterns that they use. We focus on four main points:

- **Verification of user-supplied functional specifications**. For the verifier to detect bugs in the smart contract, authors must be able to specify the intended functional behaviour of their smart contracts. The Flint verifier must be able to consume and verify contracts against functional specifications, which describe the sufficient conditions of the smart contract.

- **Verification of user-supplied holistic specifications**. Authors must be able to specify the intended emergent behaviour of their smart contracts, behaviour the contract may exhibit over multiple transactions, for the verifier to detect bugs in the smart contract. The Flint verifier must be able to consume and verify contracts against these specifications, which describe the necessary conditions of the smart contract.

- **Detect against common immutability bugs**. For the verifier to prevent Flint contracts from using dangerous patterns, it must be able to detect them. In particular we are interested in potential out-of-bounds, division by zero and unreachability errors.

- **Meaningful verification feedback**. For the Flint verifier to be useful to smart contract authors, it must return meaningful verification information back to the user. While developing smart contracts, we expect that the contract implementation will frequently not satisfy it's specification. Therefore, it is important for the verifier to return information which allows for easy diagnosis of a failing specification. For example, the location of the responsible violating code, or the specific specification predicate being violated.

1.1 Challenges

In this section, we outline the main challenges of this project:

- **Developing a verification system**. The development of a verification system requires significant design, implementation and understanding of verification methods. Furthermore, the verification system must support as many Flint constructs as possible, for it to be useful to Flint contract authors.

- **Designing the specification language syntax and semantics**. A verification result is only as meaningful as the specification it was verified against. We must design an expressive specification language which allows smart contract authors to completely express the expected behaviour of their contracts, but also be intuitive and clear to understand.

- **Implementing an extensible verification system**. As more language constructs are added, and Flint is developed, the verifier must be able to quickly support the new features. Furthermore, as more dangerous Flint patterns are detected, the verifier must be extensible to it may be updated quickly.

- **Evaluating the verifier**. Applying automatic formal verification to smart contracts is a rapidly developing area. At the beginning of this project we found groups which had published their intention[8], [9], or began and not completed[10], work on automatic formal verification of smart contracts. By the end of the project, we found only one group with a completed tool[1]. As such, evaluating the impact of our contribution presented a challenge.

[1]The VerX system by ChainSecurity, found here `https://verx.ch`, publication pending.

1.2 Contributions

We present the Flint verification system.

1. **Flint to Boogie translation scheme**. We design a translation scheme from Flint to the Boogie intermediate verification language [11], which preserves enough of Flint's semantics to verify useful properties of Flint code. We detail our translation scheme in chapter 3.

2. **Flint Verifier**. We present the Flint verification system. Integrated within the Flint compiler, a smart contract author is able to write and verify Flint smart contracts at compile-time.

   (a) **Proof obligation syntax**. We extend Flint's grammar to include a functional and holistic specification language, including contract and structure invariants; assertions; and pre and post-conditions. This enables Flint smart contracts to be verified against user-supplied specifications. We detail our introduction of functional and holistic specifications in sections 4.1 and 4.7.

   (b) **Automatic verification**. We implement the translation scheme, mentioned earlier, and use it to automatically translate Flint contracts. We implement a system which verifies functional specifications using the Boogie verifier[11], and holistic specifications using the Symbooglix engine[12]. We discuss this further in chapter 5.

   (c) **Dangerous pattern protection**. We harness the Flint verification system and automatically detect dangerous patterns, such as array out-of-bounds, division by zero, and unreachability errors. This is discussed in chapter 4.

   (d) **Verification output**. When a verification failure occurs, we provide output which allows contract authors to identify the problematic areas within their contract. We discuss our method in section 5.4.

3. **Evaluation**. We compare the functionality of Flint's verification system against Solidity verification tools, such as the VerX system and the Solidity compiler. We find that the Flint verifier combines functionality from both of these systems, specifically in functional and holistic specification verification, and the detection of dangerous design patterns. We also evaluate the expressiveness of Flint's specification language and find it more expressive than the Solidity compiler's verification system, as Flint's verifier supports a larger set of language constructs. Our evaluation of Flint's verification system in found in chapter 6.

Chapter 2

Background

In this chapter, we introduce the concepts required to automatically verify Flint smart contracts, and the motivation for verification. We outline the Flint language and the current state of smart contract verification. We also describe two approaches to specifications, classical and holistic. We introduce satisfiability modulo theory (SMT) solvers, and symbolic execution, which can be used for automatic verification. Finally we introduce intermediate verification languages (IVL), which can be used to encode verification problems from high-level languages, such as Flint.

2.1 Ethereum Basics

Ethereum[13] is a decentralised smart contract technology. Smart contracts are deployed to the Ethereum blockchain, and users or other smart contracts are able to execute functions on it. Interactions with Ethereum smart contracts occur through transactions. Users submit a transaction to the Ethereum network which can invoke functions on smart contracts, and can also transfer *Ether*.

Ethereum users and contracts can hold the Ethereum digital currency, *Ether*, of which its smallest denomination is *Wei*. A user can get some Ether by either running a full Ethereum node and mining a block on the blockchain or the user can be sent some Ether from another Ethereum user who owns some Ether. Contracts are able to receive money from users and can send money to other contracts or back to users. At time of writing, 1 Ether is worth $256.88 USD.

In the next two sections, we discuss two high-level smart contract languages which can be used to write Ethereum smart contracts.

2.2 Solidity

Here we briefly mention the Solidity smart contract programming language. Solidity[1] is currently the official[2] high-level language used to write Smart Contracts. However the design of the Solidity programming language can make it easy to introduce bugs or vulnerabilities into a smart contract. Bugs and vulnerabilities are always unwanted, especially when your smart contract is handling money. We describe some of the problematic design choices of the Solidity language and we also mention some of the most famous hacks and exploits of Ethereum smart contracts, which occurred as a result of these poor design choices.

The language was inspired by other programming languages such as JavaScript and C++ [1]. However, writing smart contracts in this language makes it hard to reason about the state of the smart contract, which can lead to the introduction of bugs and vulnerabilities[14]. For example the ‘fallback’ method, which is the function that is called if no function matches the signature of the function being called. The fallback function makes it hard to reason about which function of a smart contract will be called at run-time. This introduces uncertainty when calling a contract, which makes it hard to reason about the state of that contract.

Another design issue is the lack of distinction between assets and integers. In Solidity, the value of Wei held in a smart contract is represented directly as integers, rather than as their own type. This allows for contamination of assets with other non-asset types, for example this could lead to the accidental deletion of 100 Wei from the variable representing the value of the contract.

```
pragma solidity >=0.4.22 <0.6.0;

contract Bank {
    uint public value;

    constructor() public {
        value = 0;
    }

    function deposit() public payable {
        value += msg.value;
    }

    function buggyAccounting() public {
        // This operation simply removes 100 Wei from 'value'.
        /* There is now a discrepancy between 'value' and
        the Wei actually held by the contract */
        value -= 100;
    }
}
```

Listing 2.1: Example of a dangerous Bank smart contract written in Solidity

2.2.1 Smart contract vulnerabilities

Here look at the various types of vulnerabilities that are possible, with smart contracts in general, and Solidity contracts in particular. Atzei at al[14] conducted survey on the state of attacks on Ethereum smart contracts, and proposed the following taxonomy of vulnerabilities.

- **Call to the unknown.** Some of the primitives in Solidity can have a side effect of invoking the fallback function, namely `call`, `send` and `delegatecall`. For example, if there is a typo in the calling function's code and attempts to call `ad(uint256)`, which doesn't exist, instead of `add(uint256)` then the callee's fallback function will be invoked, instead of throwing an error.

- **Exception disorder.** In Solidity, depending on how you call an external function will affect how exceptions thrown in the external contract are handled. Using a direct call, `c.externalFunction(42)`, will propagate exceptions from the external contract and throw an exception in the caller's contract. Using the `call` method, does not do this.
`c.call.value(42)(bytes4(sha3("ping(uint256")),n)` instead returns a boolean value, where `false` indicates that an exception was thrown in the called contact. Many contracts do not check the return value of `call` invocations [15].

- **Gassless send.** When using the Solidity function `send` to transfer Ether, it is possible to incur an out-of-gas exception. This is because, the `send` function will invoke the receiving contract's fallback function. However, by default the number of units of gas available to the callee is 2300, which only allows a set of bytecode instructions, which do not modify contract state, to be executed. In most cases, the `send` will end up with an out-of-gas exception.

- **Type casts.** The Solidity compiler does not check that the expected interface of an external contract matches the contract's actual interface. Since the compiler does not check the declared interface of external functions, the declared interface must be completely correct otherwise it is likely that the external function's callback function will be called. All of this will occur with not exceptions being thrown.

- **Re-entrancy.** This occurs when a function makes an external call, and allows an external call to re-enter the original function. This is particularly bad if an action, such as sending money, happens before updating contract state, which allows an attacker to perform repeated operation on the smart contract. As was the case with the DAO [4] attack, where $60 million was stolen

- **Keeping secrets.** Fields in contracts can be public, directly readable by everyone, or private. However, declaring fields as private does necessarily make it so. For example, to set the value of a field a transaction, containing the field's value, must be published to the blockchain. Since the blockchain can be inspected by anyone, the field's value can be deduced.

- **Immutable bugs.** Once deployed to the blockchain, the smart contract is immutable and so are it's bugs. If the contract contains unintended functionality, there is no direct way of fixing it.

- **Ether lost in transfer.** One must specify the recipients address, if they wish to send them some Ether. However, many of these addresses are not associated with any smart contract or user. If Ether is sent to any such address, it is lost forever. Since, it is not possible to detect whether a smart contract or user is associated with an address, smart contract authors must manually ensure the correctness of a recipient's address.

- **Unpredictable state.** When a user sends a transaction to interact with a smart contract, they cannot be sure of the contracts state (value of fields and contract balance). This is because Ethereum provides no guarantee that the transaction is executed when it is submitted, instead it is as the discretion of miners to decide which transaction to include in their current block. This allows for another transaction, targeting the same smart contract, to be executed before your transaction, even if you submitted your transaction first.

- **Generating randomness.** There are many applications where generating random numbers is useful. However, the execution of smart contract code (EVM bytecode) is deterministic, therefore pseudo-random generators are used to address this need. However, smart contract authors must decide what value to provide as a 'seed' to the number generator. A common way of providing a seed, is to use the hash or time stamp of a block on the blockchain. However, miners control when a block in mined and the transactions which are put in the block, and thus could craft a malicious block to bias the outcome of the random number generator.

- **Time constraints.** Smart contracts use time constraints to determine which actions are permitted, in the current contract state. This is usually done using block time stamps. However, the miner has some leeway in choosing the time stamp for that block, approximately 900 seconds. If the miner has a stake in a contract, they may choose an appropriate time stamp to gain an advantage.

With the taxonomy of vulnerabilities above, we see that some of the vulnerabilities are inherent to the design of the Ethereum system, such as 'Ether lost in transfer'. However we note that many are due to poor Solidity language design, and unexpected behaviour. We see that the application of formal verification to smart contracts could mitigate the 'call to the unknown'; 'exception disorders'; 're-entrancy'; and 'immutable bugs' vulnerabilities. Allowing the user to specify the correct behaviour of the contract, and automatically checking whether the contract conforms to its specification would allow the contract author to detect these vulnerabilities at compile time.

2.2.2 Remarks

As we can see, even though Solidity is the most popular and accessible smart contract programming language, for Ethereum, it provides very little safety. It is hard to reason about the behaviour of the contract, and this can lead to exploits which drain smart contracts of vast sums of money. We discussed how formal verification can mitigate these issues, preventing vulnerabilities such as 'immutable bugs' and re-entrancy.

In the next section we look at the smart contract language Flint, which attempts to address some of the problems with Solidity we introduced here.

2.3 Flint

The Flint smart contract programming language was designed and developed by Franklin Schrans in 2018[6]. Flint was designed to prevent smart contract authors from introducing bugs and vulnerabilities into their smart contracts. Flint introduces constructs which significantly improve the contract author's ability to reason about the state of a smart contract. Here we introduce the key syntax and semantics of Flint. You can find complete examples of Flint contracts in appendix B.

2.3.1 Contract declaration

Contracts in Flint are declared using the `contract` keyword. All a contract's state properties must be declared within a contract declaration.

```
// Contract declarations contain only their state properties.
contract Bank {
  var manager: Address
  var balances: [Address: Wei] = [:]
  var accounts: [Address] = []
  var lastIndex: Int = 0

  var totalDonations: Wei = Wei(0)
  event didCompleteTransfer (from: Address, to: Address, value: Int)
}
```

Listing 2.2: Bank contract declaration[1]

2.3.2 Traits

Traits describe the interface, partial behaviour, of contracts or structs which conform to them. For example, Flint's Wei struct conforms to the Asset trait, shown in appendix A.4. Furthermore, traits can also be used to describe the interfaces of external contracts. These external traits, are used to call functions on external contracts. We give an example below:

```
external trait ExternalBank {
  @payable
  func pay() -> int256
  func withdraw(amount: int256) -> int256
}
```

Listing 2.3: Flint external trait example[2]

2.3.3 Control flow

Flint contains control flow constructs: for-loops, if-else statements, do-catch statements and function calls. We give an example Flint function, which exercises most of Flint's control flow statements, below. We discuss the do-catch statement and external calls, in the following subsection.

```
func myName() -> String {
    for let userName: String in users {
        returnDeposit(userName)
    }

    if self.x == 2 {
      // ...
    } else {
      // ...
    }

    return "Name"
}
```

Listing 2.4: Example extract of Flint control flow

[1]Taken from https://github.com/YianniG/flint/blob/master/examples/casestudies/Bank.flint
[2]Taken from https://docs.flintlang.org/docs/language_guide#specifying-the-interface

2.3.4 External Calls

Flint supports external calls, the interface of the callee must be defined using an external trait, which we introduce in subsection 2.3.2. If the external call is made using the `call!` keyword, a do-catch block is not required, the whole transaction reverts if the external call fails. Otherwise, the external call must be wrapped within a do-catch block, the catch block allows the contract author to specify the code which should execute if the external call fails. We give an example below.

```
X :: (any) {
  public func callback(externalAddress: Address) {
    let extInstance = Ext(address: externalAddress)
    do {
      call extInstance.someFunction()
    } catch is ExternalCallError {
      // handle the error here
    }
  }
}
```

Listing 2.5: Example external call[3]

2.3.5 Mutation

Flint makes write operations on global state very explicit. Functions which update contract state must use the `mutating` modifier.

```
contract M {
    var value = 0;
}
M :: (any) {
    public func mutating mutates() {
        value = 10;
    }
}
```

Listing 2.6: Example mutating function

2.3.6 Asset type

Asset types separate the representation of assets, such as Wei, from raw integers. Assets also provide atomic transfer operations, `transfer`, this avoids asset contamination and vulnerabilities arising the lack of atomic state updates. The asset type would have prevented the DAO attack, by atomically updating state, mitigating the re-entrancy vulnerability.

```
// Any user in accounts can call these functions.
// The matching user's address is bound to the variable account.
Bank :: account <- (accounts) {
  public func getBalance() -> Int {
    return balances[account].getRawValue()
  }

  public mutating func transfer(amount: Int, destination: Address) {
    // Transfer Wei from one account to another. The balances of the
    // originator and the destination are updated atomically.
    // Crashes if balances[account] doesn't have enough Wei.
    balances[destination].transfer(&balances[account], amount)
  }

  public mutating func withdraw(amount: Int) {
    // Transfer some Wei from balances[account] into a local variable.
    let w: Wei = Wei(&balances[account], amount)

    // Send the amount back to the Ethereum user.
    send(account, &w)
  }
```

[3]Taken from https://docs.flintlang.org/docs/language_guide#calling-functions

```
}
```
Listing 2.7: Example Bank extract showing Wei asset from Flint documentation[4]

2.3.7 Protection blocks

Protection blocks create restrictions on when certain functions can be called on a contract. The restrictions look at ‘caller capability’ and the current state of the program, if these conditions are satisfied the function can be invoked otherwise an exception is thrown.

Caller capability

The caller capability clause specifies whether the caller can authorisation to call the function. Addresses, arrays, dictionaries and functions, can be used in the caller capability clause. In listing 2.8, the address of the caller must match the manager’s address stored in the contract.

```
contract Bank {
  var manager: Address
  ...
}

// Only the manager can call these functions.
Bank :: (manager) {
  public mutating func clear(account: Int) {
    balances[account] = Wei(0)
  }
  ...
}
```
Listing 2.8: Extract of caller capabilities from Flint documentation[5]

Type state

The type state of the contract is an implicit global variable, which represents the state of the smart contact. States are declared when declaring the contract.

```
contract Auction (Preparing, InProgress, Terminated) {}
// Preparing, InProgress, Terminated are State Identifiers
```
Listing 2.9: Example of type state declaration from Flint documentation[6]

`setBeneficiary()` can only be called whilst the contract is in the `Preparing` state. The contract state is changed using the `becoming` keyword, as seen in the `openAuction()` function below.

```
Auction @(Preparing) :: (beneficiary) {
  public mutating func setBeneficiary(beneficiary: Address) {
    self.beneficiary = beneficiary
  }

  mutating func openAuction() -> Bool {
    // ...
    return true
    become InProgress
  }
}
```
Listing 2.10: Example of type states from Flint documentation[7]

2.3.8 Payable functions

Payable functions are required to be annotated with `@payable`, which provides the function with an `implicit` parameter containing the value of Wei received by the smart contract. Payable functions are the only way a Flint contract can acquire Wei.

```
@payable
public func receiveMoney(implicit value: Wei) {
  doSomething(value)
}
```
Listing 2.11: Example payable function from Flint documentation[8]

2.3.9 Structures

Flint contracts can define their own C-like structures or `struct`'s, which allows the grouping of data, and methods.

```
struct Rectangle {
  var width: Int = 0
  var height: Int = 0

  func area() -> Int {
    return width * height
  }
}
```

Listing 2.12: Example struct taken from Flint documentation[9]

2.3.10 Remarks

As we can see, Flint is a much safer smart contract language than Solidity. Explicit asset types, and protection blocks are just a few language constructs which make the behaviour of the smart contract explicit and easy to reason about, and therefore less likely to contain bugs and vulnerabilities. We feel that Flint's approach to smart contracts represents the future of smart contract development. Clearly smart contract verification is within the spirit of the Flint language, which is why our verification system targets Flint smart contracts.

2.4 Current Methods of Verifying Smart Contracts

In order to automatically verify Flint smart contracts, we must first understand the different approaches to smart contract verification. This is an active area of development, in this section we look at a variety of tools which test or verify smart contracts. We look to understand the approach taken by each tool, and consider the approach's guarantees and limitations. As we explore the different techniques, we take note of the approaches we could adopt for Flint's verification system.

2.4.1 Testing Frameworks

Testing frameworks and unit tests are ways in which the program author can test that behaviour of a smart contract, for specific test cases. As formal verification methods for smart contracts are in their infancy, the majority of smart contract behaviour is verified using unit tests. We briefly introduce two testing frameworks, and outline the limitations and correctness guarantees of each.

Remix

Remix [16] is a web-based Solidity IDE. It provides a local browser-based blockchain, which a smart contract author can use to deploy Solidity smart contracts. This tool enables easy interactions, and manual testing of the smart contract. Once deployed, to test the smart contract, the author can submit transactions and observe its behaviour.

Truffle Suite

The Truffle Suite [7] is a comprehensive smart contract development framework. It provides set of tools for the automated testing, deployment and management of Ethereum smart contracts. The Truffle Suite automatically tests smart contract against a suite of user-supplied unit tests. It does this by deploying the contract to a local blockchain and invoking functions on it, as described in the unit tests.

Remarks

We can see that test suites are valuable tools to smart contract authors, they provide an indication of the correctness of a smart contract and, in the case of the Truffle Suite, the testing is quick and automated.

However, unit tests do not provide the strong correctness guarantees we are looking for. Unit testing only verifies the behaviour of the execution paths activated by the given unit tests. Therefore, unit testing does not guarantee that the smart contract behaves correctly far all possible execution paths, only the tested ones. We are looking to show contract correctness for all execution paths, therefore unit testing is not a viable approach for Flint's verification system.

2.4.2 Security Scanners

Another set of common tools used by smart contract authors is security scanners. Security scanners such as Oyente[17], Mythril [18] and Securify[19] analyse Solidity, or EVM byte-code, and search for potential vulnerabilities. We discuss these systems because they are used by smart contract authors to verify whether their contract code is safe.

Below we briefly introduce the most common security scanners, Oyente, Mythril and Securify.

Securify

Developed by ChainSecurity[10], Securify[19] was released in July 2018. Securify's approach to security analysis involves decompiling the smart contract's EVM byte-code and extracting 'semantic facts' about the contract. The system then uses a curated database of 18 security patterns[19] and checks the semantic facts for any security violations. The security patterns include:

- **Ether liquidity.** Detect if Ether can become 'stuck' within a contract.
- **No writes after call.**
- **Restricted write.** Detect unconditional writes to contract storage.
- **Transaction ordering dependency.** Described in section 2.2.1.

Mythril

Proposed by B. Mueller in 2018[18], Mythril's approach to detecting security vulnerabilities involves symbolically executing (see later section 2.5.2) EVM byte-code. Mythril has a set of 19 pre-loaded security violations, which the symbolic execution engine searches for violations against[11]. The set includes:

- **Mishandled Exceptions.** Described in section 2.2.1.
- **Re-entrancy.** Described in section 2.2.1.
- **Integer overflow.**

Oyente

Oyente [17] is a symbolic execution tool to find potential security bugs, it also operates on EVM bytecode. The Oyente system can detect specifically 4 classes of vulnerabilities.

- **Transaction Ordering Dependence.** This is when there is a dependency between two transactions, interacting with the same smart contract, mined in the same block.
- **Time stamp Dependence.** It is problematic if a smart contract uses the block time stamp in any kind of condition, as the miner for any transaction which targets that contract can vary this value by 900 seconds.
- **Mishandled exceptions.** Described in section 2.2.1.
- **Re-Entrancy.** As described in section 2.2.1

[10]Found here: https://chainsecurity.com/
[11]Complete list of Mythril detection capabilities https://github.com/ConsenSys/mythril/wiki/Mythril-Detection-Capabilities

Remarks

Security scanners are popular tools with smart contract authors as they detect common security vulnerabilities, such as re-entrancy and mishandled exceptions vulnerabilities which have previously been found and exploited in smart contracts.

To do this, we see that each smart contract security scanner has a database of known vulnerabilities and code patterns, which they use to identify dangerous code. Indeed, they can only detect vulnerabilities which are part of their database. It is not possible to check smart contracts against user-supplied specifications, as a user cannot supply their own specifications. As a result, these security scanners cannot detect 'safe' but incorrect behaviour of smart contracts. It is important for our verification system to detect incorrect contract behaviour, to mitigate immutability bugs, even if it is not a security vulnerability. Therefore the Flint verification system does not take this approach.

2.4.3 K

Another approach to automatically verify smart contracts is to express its formal semantics, and execute them with respect to a specification. The K framework[20] has been used to express the formal semantics of many programming languages[21], both Solidity and the Ethereum EVM's semantics have been expressed in K[22][23]. The main benefit of the K framework, is the 'executable formal semantics'. This means that the K framework uses the language semantics to generate an interpreter, which can be used to show that the semantics of the language is correct[12].

This approach could allow for the verification of user-supplied specifications, however it requires that the semantics of Flint are expressed in K. This is challenging for two reasons, Flint's semantics have not been formalised thus far, the Flint compiler currently defines the semantics of each Flint language construct. To express Flint in K, we must first formalise Flint's semantics. Secondly, once the formal semantics of Flint have been expressed in K, we would need to show that the K translation is correct. The expression of the formal semantics of Flint, and associated proof, is a large endeavour. This is out of scope of what we aim to achieve in this project, we acknowledge that this is a potential area for future work and we discuss it in section 7.1.1.

2.4.4 Manual verification

To verify the implementation of any smart contract, the smart contract author could write a proof, by hand, to show this. We introduce the Hoare logic in section 2.6, which can be used to do this. This requires the formal semantics of the smart contract language to be defined, and that the smart contract author as the expertise and time to prove the specification. Formal proof assistants may be used, such as Coq [24] to speed up, automate and verify components of the proof, but ultimately it is still a manual process.

Unlike security scanning systems, manual formal verification also allows smart contracts to be verified against user-supplied specifications. Unlike unit tests, formal verification provides the strong guarantees that we are looking for. It can show that the specification of a smart contract holds for all possible execution paths. However, the limitations of manual formal verification are that the smart contract author requires expertise and time to construct a formal proof to show that the contract satisfies its specification. Therefore, this approach is not accessible to a wide audience of smart contract authors. Furthermore, the proof would need to be re-created and checked every time the implementation of the smart contract changed, as the new implementation may not satisfy the smart contract's specification.

We can see that manual verification requires expertise and a large time investment. Furthermore, manually re-checking whether a contracts satisfies it's specification is a tedious process, especially during rapid smart contract development. However, due to the strong correctness guarantees of formal verification, and that it can be used to verify user-supplied specifications, we can see the

[12]More information about the K framework's executable semantics can be found here: `http://www.kframework.org/index.php/Main_Page`

benefits of formal verification.

In the following two subsections, we discuss automatic formal verification systems.

2.4.5 Solidity SMT Checker

Automatic formal verification can mitigate the limitations of manual formal verification, which we discussed in the previous section, whilst providing the same guarantees and flexibility of manual formal verification. We are interested in the flexibility of the verification system to support user-supplied smart contracts, and the guarantees showing contract correctness for all execution paths in the smart contract.

In 2018, the Ethereum foundation sponsored a project to explore automatic verification of the Solidity language, with the goal to add automatic formal verification to the Solidity compiler [25]. The Solidity compiler's SMT Checker module introduces this verification functionality into the Solidity compiler[13]. The verification module uses Solidity's `assert` and `require` statements as a way for contract authors to express specifications. We discuss writing specifications in more detail in section 2.6 below. The `assert` statement allows the contract author to specify properties which must be true at that point in the contract execution. Similarly, the `require` statement allows the contract author to specify properties which can be assumed to be true, at that point in the contract execution. The verification system is safe to make this assumption, as both `require` and `assert` are translated into run-time checks, which fail if the properties are not satisfied.

In order to perform the formal verification, Solidity's SMT Checker module translates the Solidity contract code into an SMT encoding, using the Hoare logic [26]. An example is shown below in fig. 2.1. We discuss SMT solvers in section 2.5.1.

```
contract C
{
  function f(uint256 a, uint256 b)
  {
    if (a == 0)
      require(b <= 100);
    else if (a == 1)
      b = 1000;
    else
      b = 10000;
    assert(b <= 100000);
  }
}
```

1. $a_0 \geq 0 \land a_0 < 2^{256} \land$
2. $b_0 \geq 0 \land b_0 < 2^{256} \land$
3. $(a_0 = 0) \rightarrow (b_0 \leq 100) \land$
4. $b_1 = 1000 \land b_2 = 10000$
5. $b_3 = ite(a == 1, b_1, b_2) \land$
6. $b_4 = ite(a == 0, b_0, b_3) \land$
7. $\neg b_4 \leq 100000$

Figure 2.1: Demonstration of solidity SMT encoding, taken from [25]

The SMT encoding is fed into the Z3 [27] or CVC4 [28] SMT solvers, which perform the formal verification automatically.

L. Alt and C. Reitwiessner introduce the state of the current work being done on verification in the Solidity Compiler [25]. They outline what the compiler can currently verify, and areas of future work. We list the currently supported features below:

- **Detection of infinite while loops.** The loop condition is always true.
- **Detection of unreachable code.** Whether the if-statement condition is provably true or false.
- **Detection of arithmetic overflow.**
- **Detection of divide by zero errors**

[13]The SMT Checker module is under development. It's most up-to date documentation can be found here: https://solidity.readthedocs.io/en/latest/layout-of-source-files.html#smt-checker

- **Prove assertions.**
- **Pre-conditions.** Through the of the `require` statement.

Areas of further work:

- **Introduction of loop body pre and post-condition syntax.**
- **Automatic detection of loop bounds.**
- **Automatic loop unrolling, for bounded loops.**

As we can see, Solidity's verification system is in development. Currently only a small subset of Solidity language constructs are supported, for example the `mapping` data structure is not supported. We outline each construct that the Solidity verification system supports, in our evaluation, in chapter 6.

Nonetheless, the approach taken by Solidity's verification system, which provides automatic formal verification of smart contracts, gives the guarantees and flexibility we are looking to add to Flint's verification system. Clearly Solidity's verification system can verify user-supplied specifications, which can be expressed through the use of `assert` and `require` statements. Furthermore, the formal verification performed by the underlying Z3 SMT solver ensures the correctness of the behaviour of the smart contract, for all execution paths, with respect to its specification. Finally, Solidity's automatic verification system does not require the same level of expertise as manual verification, the smart contract author does not have to construct a proof of correctness, as the verification system absorbs most of this complexity.

In the following subsection, we introduce the VerX system which is another automatic formal verification system for Solidity smart contracts.

2.4.6 VerX

The approach taken in the previous subsection does not allow the user to specify, or verify, the behaviour of their smart contract over multiple transactions. The `assert` statement only allows the smart contract author to specify properties about the current transaction, not possible future transactions. As we look to implement this functionality into Flint's verification system, we discuss the VerX system, as it supports user-supplied multi-transaction specifications.

The VerX system[14], developed by ChainSecurity, was released in April 2019 as an automatic formal verification system for Solidity smart contracts. The work behind the VerX system is currently unpublished, therefore we cannot fully scrutinise it's implementation. However, the comments we make in this section are informed by VerX's documentation[15] and the informal discussions we have had with the VerX team.

The system introduces a specification language, which contract authors can use to write custom specifications for their smart contracts. Furthermore, the specification language allows smart contract authors to express properties about the behaviour of the contract, over multiple transactions. We give an example specification below:

```
// The escrow never allows the beneficiary to withdraw the
// investments and the investors to claim refunds.
property exclusive_claimRefund_and_withdraw {
    always(
        !(once(FUNCTION == Escrow.claimRefund(address))
          && once(FUNCTION == Escrow.withdraw()))
    );
}
```

Listing 2.13: Example VerX specification[16]

[14]VerX system can be found here: https://verx.ch

[15]VerX documentation found here: https://verx.ch/docs/

We briefly introduce VerX's supported specification predicates[17]:

- **Basic arithmetic and logical operators**. For example $a \implies b$, $a \lor b$, $a + b$.
- `prev(A)`. The previous value of global variable `A`. This returns the value of the variable, at the start of the transaction.
- `always(A)`. `A` must be maintained by the end of each transaction.
- `once(A)`. For all possible future transactions, `A` must be true at some point.
- `FUNCTION == F`. Used to test that the function `F` was called by the current transaction.
- `BALANCE`. Represents the amount of Wei held in the smart contract.

We see that the specification predicates supported by VerX allows the smart contract author to express the behaviour of the contract over multiple transactions. SMT solvers cannot be used alone to verify these types of specifications, as these temporal operators do not exist in the Hoare logic. Therefore, VerX uses a combination of abstract interpretation[29] and symbolic execution.

Symbolic execution (see section 2.5.2) allows the verification engine to simulate the behaviour of the smart contract over multiple transactions. However, for this approach to be tractable, the potentially unbounded number of transactions must be translated into a bounded model. VerX's sound abstraction technique allows them to translate Solidity smart contracts into bounded models, maintaining enough of the semantics of the original contract, allowing them to verify these specifications. We cannot go into the details of VerX's sound abstraction technique, as the VerX system paper is currently pending publication. With the semantically sound abstract model of the smart contract, symbolic execution can be used, over the model, to check whether the contract's specification holds for all possible transaction and ordering of transactions.

We see that the VerX system is able to provide formal guarantees of correctness against user-supplied specifications, automatically. In particular, the approach used by the VerX system allows the verification of smart contracts over multiple transactions. We discussed how the use of symbolic execution and abstract interpretation enables this functionality. We note how we can take inspiration from these methods to verify similar kinds of specifications, with Flint's verification system.

2.4.7 Remarks

We discussed how testing frameworks and security scanners do not provide the functionality and guarantees that we are looking to implement in Flint's verification system. We briefly mentioned how the K framework, and it's executable semantics, could be used to automatically verify Flint specifications. However, this approach requires formalising Flint's semantics, which is out of scope of what we aim to achieve in this project. We discussed how manual formal methods enable the verification of user-supplied specifications, and the strong correctness guarantees we are looking for. However, this approach requires expertise and time, which does not make this approach accessible to many smart contract authors. Finally, we discuss how the Solidity SMT Checker and the VerX perform automatic formal verification of user-supplied specifications, both providing the flexibility and guarantees we look to add to the Flint compiler.

We take inspiration from Solidity and VerX's verification systems and note that we can use SMT solvers and symbolic execution to verify user-supplied specifications. To understand the guarantees that SMT solvers and symbolic execution technology provide, we discuss them both in the following section.

[16]Taken from VerX examples: https://verx.ch

[17]More information about VerX's specification language can be found here: http://verx.ch/docs/spec.html

2.5 Automatic Verification Technology

In the previous section, we discussed how SMT solvers and symbolic execution can be used to verify specifications. Therefore, in this section we introduce satisfiablity modulo theory (SMT) solvers, and symbolic execution, and aim to give a brief introduction to each. For our work implementing Flint's verification system, we treat the SMT solver and symbolic execution engine as a black box. Therefore we will not discuss in great detail how each technology works, rather we will discuss the problems that each technology solves, and how it is related to Flint verification.

2.5.1 SMT Solvers

Satisfiability modulo theories (SMT), are a generalised form of boolean satisfiability problems. Boolean satisfiability (SAT) problems are ones which involve determining if a boolean formula is satisfiable. For example, the expression $a \vee b$ can be satisfied with the assignment $a = \top$, solving SAT problems is NP-complete. SAT solvers are tools which attempt to automatically determine if an expression is satisfiable.

SMT solvers solve similar problems to SAT solvers, except they can solve expressions which contain other theories. For example integer theory, an SMT solver can show satisfiability of the following expression $a > 10$. Since solving SMT problems, as with SAT, is NP-complete, it is not possible for an SMT solver to show satisfiability for all expressions. Many SMT solvers, such as Microsoft Research's Z3 solver [27], are sound but incomplete. This means that if the solver is able to show satisfiability, then the expression is definitely satisfiable. However, if they are not able to show satisfiability, it does not mean that the expression is not satisfiable. We mention Microsoft Research's Z3 SMT solver, as it is the SMT solver we use for Flint's verification system. Z3 is a popular and powerful SMT solver [27].

SMT solvers use a variety of complex techniques to find their solutions, however for our purposes we consider them as back boxes. Like the Solidity verification system, we can use SMT solvers to formally verify smart contracts. Using the Hoare logic (see section 2.6), we can translate Flint code and specifications into logical statements, where the SMT solver can show (or not) that the contract implementation satisfies it's specification. We discuss this in more concrete terms in section 2.7, where we explore the use of intermediate verification languages to verify smart contracts.

2.5.2 Symbolic Execution

Symbolic execution is a technique used to explore the execution states of a program[30]. It works by representing program variables with symbolic variables, which could be arbitrary values. For every statement, the symbolic execution engine updates its symbolic values. For example, given a variable a, and the statement `a += 1`, the symbolic value of the variable becomes $a + 1$. Furthermore, the symbolic execution engine tracks each state's path constraints, for example the statement `if (a > 0)  b += 1` would add an extra `a > 0` constraint on `b`'s new symbolic value.

The symbolic execution engine can use an SMT solver (such as Z3), should a property about a variable need to be shown. The SMT solver can take the symbolic value of a variable, and it's path constraints, and use them to show whether the variable satisfies a property. This allows symbolic execution engines to be used to verify programs against specifications, they symbolically execute the program and assert that the specification holds. Therefore, the power of the symbolic engine is affected by the proving strength of the SMT solver it relies on. Furthermore as more program states are explored, the more complex the path constraints become, the harder it becomes for the SMT solver to find solutions. Therefore, symbolic execution is only feasible on programs with small numbers of states and conditions, such as smart contracts.

For Flint verification, we can use symbolic execution to check multi-transaction specifications, because we can symbolically execute multiple transactions and assert the specification. We discuss how we formulate this problem, and our implementation, in section 4.7. The symbolic execution engine we use for Flint's verification system is Symbooglix [12]. This decision was based on how we translated Flint code, into a verifiable form, which we discuss in more detail in section 2.7.

2.5.3 Remarks

We discussed how SMT solvers and symbolic execution engines can be used for automatic formal verification. We acknowledged that SMT solvers are sound, but incomplete, which means that they may not be able to verify all possible verifiable specifications. We also mentioned how symbolic execution engines allow us to verify smart contract behaviour over multiple transactions.

In order to use the automatic verification methods described in this section, we must discuss writing specifications that contracts can be verified against. In the next section, we look two different approaches to writing specifications.

2.6 Writing Specifications

Specifications are a necessary component of the formal verification process. Specifications allow the smart contract author to express the correct behaviour of the contract. When verifying smart contracts, we use the contract specification to check if the contract implementation satisfies it. In this section we discuss two approaches to writing specifications, classical (also known as functional) and holistic specifications.

2.6.1 Classical specifications

Classical specifications, consisting of pre-conditions, post-conditions and invariants are common ways of writing specifications. Pre and post-conditions were introduced by Hoare in 1969 [26], as were monitor invariants and loop invariants. We discuss pre and post-conditions because we introduce them into Flint's syntax. They can be used by contract authors to express the correct behaviour of their functions. We discuss how we introduce them into Flint's syntax in section 4.1.

Pre and Post-Conditions

The pre-condition expresses the properties required for the function to behave correctly and the post-condition expresses the properties which should hold after all execution paths through the function have completed. A formal proof is required to show how the pre-condition and function implementation guarantee that the post condition holds. Hoare introduced, what is now known as, the Hoare triple [26], a concise way of expressing pre and post-conditions.

```
P {Q} R
```

Hoare triple as introduced in [26], where P is the pre-condition, Q is a program and R is the post condition.We give a concrete example below. We can see how the pre-condition and implementation guarantee the post-condition is true.

```
x + 1 = 10 {y := x + 1}  y = 10
```

Invariants

C. Hoare introduced monitor invariants and B. Meyer extended this work and proposed class invariants [31] which are properties that must hold throughout the lifespan of an object.

The idea of class properties being maintained throughout its lifespan, could be applied to smart contracts. It seems quite natural to want to express certain properties about the state of a smart contract, contract invariants, which must hold forever. For example, a smart contract shouldn't 'lose' money. The amount of money it holds, should be the amount of money it received take the money it sent. It seems very useful to check that the functions of a smart contract maintain the contract invariants.

Similarly, maintaining invariants about the data structures defined within a smart contract would be useful to understand their state. For example, expressing `struct` invariants in Flint may also be useful, which would require that by the end of each struct method the struct invariant must be maintained.

Currently, it is not possible to express any kind of specification in Flint. The inclusion of pre and post-conditions and contract and struct invariants would aid the contract author in reasoning about the state of the contract. The formal verification techniques, discussed in the previous section, could be used to verify that each Flint function satisfies its invariant.

2.6.2 Holistic Specifications

Introduced by S. Drossopoulou, J. Noble and S. Eisenbach [32], holistic specifications aim to specify the broader behaviour of a smart contract. Holistic specifications specify the necessary conditions required for an outcome, such as someone being paid, or the conditions without which an effect will not happen.

Holistic specifications are typically of the form `Effect` $\implies$ `Cause`, in contrast to functional specifications which are typically of the form `Cause` $\implies$ `Effect`.

The use of holistic specifications is a novel idea to create robust smart contracts. By creating a specification language which allows authors to explicitly state the high-level behaviour of their contract, holistic specifications aim to prevent unintended behaviour from not being detected.

S. Drossopoulou et al propose some holistic operators [32], which we informally introduce below:

- **Access**(x, y). Holds if x can access y.
- **Changes**(e). Holds if at some point the value of e changes.
- **Will**(A). Holds if A holds at some point in the future.
- **Was**(A). Holds if A held at some point in the past.
- A **in** S. Holds if A holds in the set of S.
- x.**Calls**(y, m, *z1*, *z2*, .. *zn*). Holds if x called m, on object y, with parameters *z1*, *z2* .. *zn*

We believe holistic operators allow for more expressive specifications which are useful for smart contracts. We take particular interest in the **will** operator, as it allows smart contract authors to specify behaviour of the contract over multiple transactions.

2.6.3 Remarks

We discussed how holistic and functional specifications allow smart contract authors to model different aspects of a smart contract's behaviour. Specifically, we introduced pre and post-conditions and invariants, which can be used to write functional specifications for Flint contracts. We also introduced holistic operators, where we noted that the Will operator allows contract authors to express contract behaviour over multiple transactions.

We discuss how we add specification syntax to Flint in chapter 4, where we also discuss how we verify Flint contracts against these functional and holistic specifications. In the following section, we look at how we can translate Flint code into a verifiable form, where it can be verified against it's specification using the automatic formal verification technologies discussed in section 2.5.

2.7 Intermediate Verification Languages

To perform automatic formal verification of Flint, we must translate Flint code into a format which can be consumed by SMT solvers. This is required since we would like to automatically formally verify Flint contracts, and SMT solvers do this. Most SMT solvers consume a low-level encoding of logical propositions, so we would be required to translate Flint into a series of low-level SMT encoding.

However, this may not be necessary, as we can use an intermediate verification language (IVL). IVLs provide an intermediate-level logical representation of a program, which a high-level language translation system can target. This reduces the complexity of high-level language translation systems. The IVL is translated, using it's own verification system, into a low-level SMT encoding and is verified using an SMT solver. IVLs have the same correctness guarantees as using the SMT solver directly, as the IVL is also verified using an SMT solver. Therefore, we can translate Flint code into an IVL, then verify the IVL translation of the Flint code. If the IVL code verifies, then the responsible Flint contract must also verify, assuming that the translation scheme preserves the semantics of the original Flint code. However, using an IVL increases the complexity of the compiler's technology stack and can reduce the verification performance. At this time we think that the benefits offered by IVLs outweigh the reduced verification performance. We evaluate the result of this decision in section 6.5.

In this section we explore potential IVLs which we could translate Flint into, we look at the benefits and limitation of each and their suitability for our use case.

2.7.1 F* embedding

F* [33] is a project between Microsoft Research [34] and INRIA [35], a general-purpose functional programming language aimed at program verification, powered by Z3 [27].

K. Bhargavan et al propose a framework to analyse and verify the functional correctness of Solidity contracts, using F* [9]. In particular, they propose two tools Solidity* and EVM*. Solidity* translates Solidity programs to a representation in F* and verifies it. They propose how to perform a shallow translation from a subset of Solidity, and outline the how to translate from their subset of Solidity to F*. For EVM*, they propose a tool which decompiles EVM bytecode to analyse low-level properties, such as the amount of gas consumed by calls.

For verifying Flint, we could approach the problem in the same way as K. Bhargavan et al did and perform a translation from Flint to F*. However, since F* is a functional language, this would not be a straightforward translation as we not only translate constructs but also programming paradigms.

2.7.2 Scilla

Scilla, short for Smart Contract Intermediate-Level Language [8], is a proposed intermediate verification target language for Solidity. Scilla representations of Solidity contracts can be verified by placing the Scilla contract into Coq [24], and using the interactive proof assistant to prove that the contract implementation satisfies it's specification [8]. However, the paper only introduces how to represent a Solidity contract in Scilla and how to represent a Scilla contract in Coq. It introduces no public tool or concrete implementation of this work, therefore we cannot integrate Scilla as part of Flint's verification system, as it doesn't yet exist.

2.7.3 Boogie

The Boogie [11] language was created from within the RiSE[18] group in Microsoft Research [34], and was designed for verifying object oriented programs. The smart contract model that Ethereum uses is quite analogous to the object oriented model, where the contract is an object and each function is a method defined on it.

As shown in fig. 2.2, Boogie was also explicitly designed as an intermediate verification language and many languages currently compile down to it, such as Dafny [36], Java Modelling Language [37] and C [38]. Because of how many other languages use Boogie as their IVL, Boogie will benefit from the research and resources at Microsoft Research. Furthermore, due to Boogie's popularity, there are other analysis tools which have been built as part of the Boogie ecosystem, and consume Boogie code. Therefore, any tool which emits Boogie code can use and benefit from these tools, for example:

[18]More information about the RiSE group can be found here: `http://rise4fun.com/`

[19]`https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/`

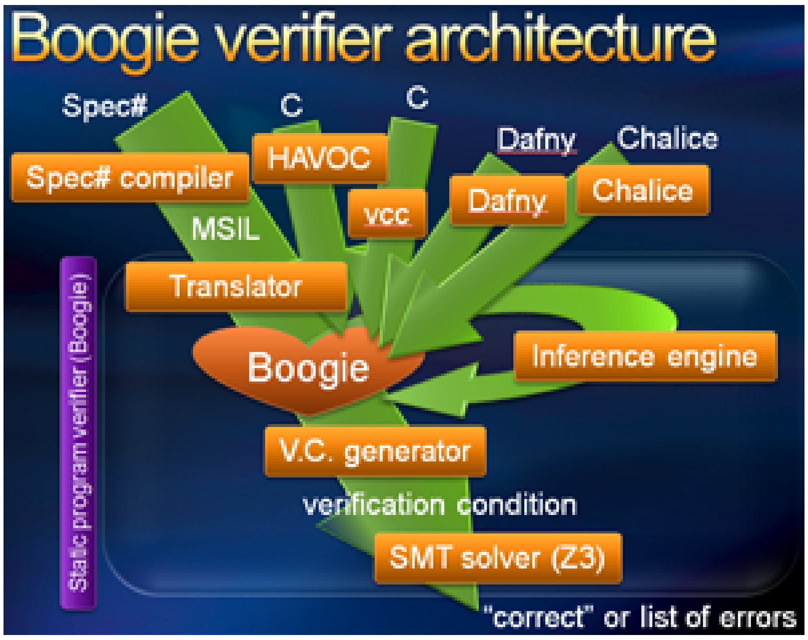


Figure 2.2: Boogie verifier architecture, taken from[19]

- **Boogaloo**. The run time assertion checker, which is able to find concrete examples for failing assertions [39].
- **Symbooglix**. A symbolic execution engine [12].
- **Corral**. A solver for the reachability modulo theories problem [40].

We discuss how to execute the Boogie verifier in the implementation chapter 5 of our work. As mentioned in section 2.5.2, we can use symbolic execution to verify multi-transaction contract specifications. The Symbooglix symbolic execution engine can consume and Boogie code. Therefore, if we use Boogie as Flint's IVL, we can verify functional specifications, but we can also verify multi-transaction specifications using Symbooglix.

2.7.4 Remarks

We discussed how translating to an IVL, rather than a low-level language, reduces the complexity of the verification translation system, whilst maintaining the same verification guarantees as using an SMT solver directly. We also acknowledge that the use of an IVL will increase contract verification time, as we introduce another layer of translation.

We also discussed how translating to an object-oriented IVL is very natural for translating smart contracts, where each contract can be thought of as an object. The Boogie IVL and surrounding ecosystem allows us to benefit from the verification of functional specifications, using Boogie's verification system, and also multi-transaction specifications, using Symbooglix. In the following section, we introduce the Boogie IVL syntax.

2.8 Boogie Intermediate Verification Language

Here we briefly introduce the syntax and semantics of the Boogie IVL, which we will translate Flint code into. We outline the grammar, types, variables, control flow and proof obligations that the language supports.

2.8.1 Boogie grammar

The high level structure of a Boogie program is as follows:

```
program ::= typedecl* symboldecl* axiom* vardeclstmt* proc* impl*
typedecl ::= type typename;
symboldecl ::= constdecl | functiondecl constdecl ::= const var : type;
arraytype ::= [ type, type ] type
functiondecl ::= function function ( type* ) returns ( type );
axiom ::= axiom expr;
```

Figure 2.3: Boogie grammar[20]

We give an example Boogie program in appendix A.3.

2.8.2 Types

Boogie supports 5 basic types, including the `map` type which can map from any type to any type.

```
var integers: int;
var reals: real;
var booleans: bool;
var bitvector: bv1;
var map: [int]bool;
```

Listing 2.14: Example type declarations in Boogie

You can also define your own types, such as

```
type Address;
const wallet: Address;
```

Listing 2.15: Example custom type declaration in Boogie

Which declares a type Address, and declares `wallet` is a fixed, but unspecified, value of type `Address`.

2.8.3 Procedures and Pre and Post-Conditions

Boogie procedures are analogous to functions or methods. They contain the statements which we would like to verify, and can be annotated with pre and post-conditions. The `requires` clause specifies a pre-condition for the procedure, which is assumed to hold on entry. The `ensures` clause specifies the procedure's post-condition, which the Boogie verifier attempts to prove by the end of the procedure.

```
var contractWei, receivedWei, sentWei: Wei;

procedure deposit_Bank(value: Wei)
    // No negative money
    requires (value ≥ Wei.New(0));
    requires (contractWei ≥ Wei.New(0) ∧ receevedWei ≥ Wei.New(0) ∧ sendWei ≥ Wei.New(0));
    requires (contractWei = receivedWei − sentWei);

    // Ensure no money is lost
    ensures (contractWei = receivedWei − sentWei);
{
    receivedWei := receivedWei + value;
    contractWei := contractWei + value;
}
```

Listing 2.16: Example Bank deposit Boogie program

For example, this is procedure illustrating what a deposit function for a Bank may look like when translated to Boogie. The post-condition tests that no money has been lost, during the deposit.

It is possible to call Boogie procedures:

```
procedure F(n: int) returns (r: int)
  ensures 100 < n ⟹ r = n − 10;
  ensures n ≤ 100 ⟹ r = 91;
{
  if (100 < n) {
    r := n − 10;
  } else {
    call r := F(n + 11);
```

[20] Taken from https://boogie-docs.readthedocs.io/en/latest/LangRef.html#grammar

```
    call r := F(r);
  }
}
```

Listing 2.17: Example calling Boogie procedure taken from Boogie documentation [21]

When calling a Boogie procedure the callee's pre-condition must hold at the call location, otherwise the verification will fail.

2.8.4 Global Variables and Modifying Them

Program state is made up of mutable local or global variables. Local variables are declared within procedure bodies and global variables are declared alongside procedure declarations. When a procedure modifies a global variable it must declare the global variable it modifies, this is done with the `modifies` clause. We give an example of a global variable and modifying procedure below.

```
var balance: int;

procedure increaseBalance(value: int)
    requires(value ≥ 0);

    modifies balance;
{
    balance := balance + value;
}
```

Listing 2.18: Modifying a global variable

2.8.5 Axioms and Functions

Function declarations introduce mathematical functions into Boogie programs and the properties of these functions are shown through axiom declarations. For example:

```
function Wei.New(value: int) returns (result: Wei);

axiom (∀ i, j: int • i > j ⟺ Wei.New(i) > Wei.New(j));

axiom (∀ i, j: int • i ≥ 0 ∧ j ≥ 0 ∧ i = j ⟺ Wei.New(i) = Wei.New(j));

axiom (∀ i, j, k: int • i + j = k ⟺ Wei.New(i) + Wei.New(j) = Wei.New(k));
```

Listing 2.19: Example function declaration

2.8.6 Control Flow and Loop Invariants

Boogie contains basic control flow constructs, such as `if-else` statements and `while` loops. Boogie is sometimes able to infer the loop invariant, of while loops, however it is possible to add `invariant` modifiers to the loop statement to make this explicit. On entry to the loop, the verifier asserts that the loop invariant holds. It then assumes the invariant and shows that the invariant holds by the end of the body. Below, we give an example using the `while` statement.

```
var n, N: int;
assume (N ≥ 0);
n := 0;
while (n < N)
    invariant n ≤ N;
{
    n := n + 1;
}
```

Listing 2.20: Example while loop with loop invariant

2.8.7 Assumptions and Assertions

Assumptions can be inserted into Boogie programs to aid the verifier. `assume` statements are assumed to be true by the verifier, and can be used by the verifier to fulfil any relevant proof obligations. Similarly, `assert` statements can be placed into Boogie programs which become extra proof obligations for the verifier to solve. Assertions can also be used to aid the verifier, by introducing lemmas, which the verifier can then combine to satisfy it's final property. Below, we give an example of an assertion and an assumption.

[21] https://rise4fun.com/Boogie/McCarthy-91

```
procedure mathsCheck()
{
    var i: int;
    assume (i == 2);
    assert ( 2 * i == 4);
}
```

Listing 2.21: Boogie assumption and assertion example

There is a special case for the `assume` statement. If the verifier comes across `assume (false);`, the verifier stops verifying the current execution path. This is because `assert (false);` allows the verifier to trivially prove any property[11]. For example, the following procedure successfully verifies.

```
procedure assumeFalse()
{
    assume (false);
    assert (1 == 2);
}
```

Listing 2.22: Boogie assume false example

2.8.8 Remarks

We see that the Boogie intermediate verification language is an expressive verification language. Boogie supports local and global variables; basic types; and basic control flow, such as if-statements, while-loops and function calls. Importantly for our purposes, we can also express assertions, assumptions, and pre and post-conditions. We discuss in chapter 4 how these can be used to express Flint functional specifications.

2.9 Remarks

Looking at the history of smart contract hacks and exploits, we see the importance of verifying smart contracts. Large amounts of money has been lost due to bugs in smart contracts. The Flint language was designed with smart contract safety in mind, this is seen with the introduction of key features such as protection blocks and the Wei asset type. In keeping with this theme, a verification system for Flint would further improve the safety offered by the language.

To prevent bugs and vulnerabilities, smart contract authors cannot rely solely on unit testing to guarantee the correctness of their smart contracts. The DAO and Parity Wallet hacks made an example of the risks that contract authors take when using only unit testing. Instead, by showing correctness for all execution paths, we see that formal verification provides the strong guarantees that smart contract authors require. Acknowledging this, the Ethereum Foundation is developing an automatic formal verification system within the Solidity compiler[22]. This system can verify a subset of Solidity's language constructs, using the Z3 SMT solver, against user-supplied functional specifications. We also see that the VerX system, developed by Chain Security, can automatically verify user-supplied holistic specifications for Solidity smart contracts using symbolic execution.

Looking at Solidity's verification method, we see that the existing approach used to automatically verify contracts against user-supplied specifications include translation of the smart contract into a verification language. The Solidity verifier's approach translates Solidity code into low-level logical predicates, which are fed directly into the SMT solver. However, this approach increases the complexity of the verification translation system. We argue that translating Flint to the Boogie IVL is less complex, and returns the same results. Furthermore, by translating Flint to Boogie we also benefit from other tools in Boogie's ecosystem, such as the Symbooglix symbolic execution engine, which we can be used to verify holistic specifications (see section 4.7.3).

In the next chapter, we introduce our translation scheme which translates Flint constructs into Boogie. In chapter 4 we discuss the new syntax we introduce to Flint, for user-supplied functional and holistic specifications. In chapters 5 and 6 we discuss our implementation and evaluation of the Flint verification system we implement.

[22]Solidity documentation for the SMT Checker `https://solidity.readthedocs.io/en/latest/layout-of-source-files.html#smt-checker`

Chapter 3

Translating Flint to Boogie

In this chapter, we introduce our translation scheme from each Flint construct to Boogie. For each Flint construct, we argue that the translation is semantic preserving.

We only propose translations for Flint constructs which are currently supported by the Flint compiler. Currently the compiler defines the semantics of each Flint construct. To verify as many contracts as possible, each Flint construct needs translating to Boogie. The translations must respect the semantics of the operation described by the construct. This ensures that any result we gain from the verification of the translation is meaningful to the original Flint source code. The translations outlined below are only defined for syntactically and semantically correct Flint contracts.

3.1 File Structure

Each Flint file contains top level declarations, such such as contract, struct, trait, external declarations; or enumerations. The translations for each of these declarations, outlined below, recursively done and appended together, forming the complete Boogie representation of the Flint file.

3.2 Types

Each Flint type must be converted into the corresponding Boogie type

3.2.1 Basic Types

Name	Flint Type (in code)	Boogie Type (in code)
Int (256-bit integer)	`Int`	`int`
Address (160-bit)	`Address`	`int`
Boolean	`Bool`	`Bool`
String	`String`	`(int, [int]int)`
Struct	`Struct`	`int`

Since Boogie has only Int's, Bool's and user-defined types. Each Flint type must be mapped to one of those. We discuss how struct types are translated in the section 3.12. Addresses are mapped into their base-10 integer representation. Strings are translated into an array of integers, representing each char value, and an integer which tracks the size of the string.

3.2.2 Dynamic Types

Flint supports the dynamic types: arrays; and dictionaries. Dictionaries can map from any type to any type, and arrays can be constructed of any type. Boogie contains only the map construct. We show how Flint types can be mapped to Boogie types below.

Name	Flint Type (in code)	Boogie Type (in code)
Dynamic-size list of type T	`[T]`	`[int]t`
Fixed-size list of type T and size n	`T[n]`	`[int]t`
Dictionary of keys K, and values V	`[K: V]`	`[k]v`

where t, k, v are the Boogie translations of Flint types T, K, V

When translating Flint arrays and dictionaries, it is important to translate the semantics of each. We cover how we maintain these semantics, in more detail below.

To maintain the semantics of the Flint constructs, we adopt *shadow variables*. These are external data structures to a data structure, which provide metadata about the data structure. As the data structure changes, it is important to update the shadow variables to maintain consistency with the data structure they are supporting.

Arrays

In Flint it is possible to access the current size of an array though the *size* field. Therefore we must track the current size of the array, and update it as the array grows in size.

```
// Create array
let array: [Int] = [0,1,2]

// Append a new value
array[3] = 3

// Query the array size
let size: Int = array.size
```

Listing 3.1: Flint access array size

We introduce a *size* shadow variable for each array. This shadow variable tracks the current size of the array. As the size of the array grows, its shadow variable is adjusted to reflect the actual size of the array. When the *.size* property of an array is accessed, we query the size of array's shadow variable.

```
// Create array
array[0] := 0;
array[1] := 1;
array[2] := 2;
size_array := 3;

// Append a new value
array[3] := 3;
size_array := size_array + 1;

// Query the array size
size := size_array;
```

Listing 3.2: Access array size Boogie translation

In listing 3.2, we can see how we increase the `size_array` shadow variable when a new value is appended to the end of the array. A naive translation of assigning to an array index, as illustrated above, always increments the size of the array. However, when assigning to an array, we might only be overwriting an existing value. In this case, it would be incorrect to increment the size of the shadow variable as the size of the array has not increased. To handle this case, a simple `if` check would suffice.

```
// Append a new value
array[3] := 3;
if (3 == size_array) {
    // Index of 3 is equal to size of array, so we are appending to the array
    size_array := size_array + 1;
}

// Query the array size
size := size_array;
```

Listing 3.3: Boogie array access to update size shadow variable

For fixed size arrays, the size of the array cannot change and is always known. The value of the size shadow variable is n, where n is the size of the array. For assigning into a fixed size array, the if check is not necessary, as the size of the array cannot increase.

Finally, when assigning an array we must also update it's corresponding size shadow variable.

```
let A: [Int] = [0]
let B: [Int] = [1, 2]

A = B
assert (A.size === B.size)
```

Listing 3.4: Flint array assignments

For example, in listing 3.4 we must make sure to set A's shadow variable to be equal to B's. We generalise this rule for any array assignment, the size shadow variable of the *lhs* of the assignment must be set equal to the corresponding size shadow variable of the *rhs*.

Dictionaries

For dictionaries, we need to track its size and we also need to track the dictionary's *keys*. The keys property contains all the keys currently within the dictionary.

We introduce the *size* and *keys* shadow variables. To keep the shadow variables consistent with the dictionary, we use a method similar to the one described above.

```
let d: [Address: Int] = [0x1: 1]

// Assigning into the dictionary
d[0x2] = 2
```

Listing 3.5: Flint assign into dictionary

Like assigning values to arrays, for dictionaries we need to determine whether a new key is being added to the dictionary. If a new key is being added we need to add this key to the *keys* shadow variable and increment the size shadow variable. We do this by iterating through the keys shadow variable, until the size of the dictionary, and checking if it contains the key. If we do not find the key we add it and increment the size shadow variable.

```
d[0x1] = 1;

// Assigning into the dictionary
d[0x2] = 2;
tmp_counter := 0;
contains_value := false;
while (tmp_counter < size_d) {
    if (keys_d[tmp_counter] == 0x2) {
        contains_value := true;
    }
    tmp_counter := tmp_counter + 1;
}
if (contains_value) {
    keys_d[size_d] = 0x2;
    size_d := size_d + 1;
}
```

Listing 3.6: Boogie translation of assigning into dictionary

Finally, when a dictionary is assigned to another, we must make sure that the shadow variable values are also transferred.

```
let A: [Address: Int] = [0x1: 0]
let B: [Address: Int] = [0x2: 1, 0x3: 2]

A = B
assert (A.size === B.size)
assert (A.keys === B.keys)
```

Listing 3.7: Flint dictionary assignments

For example, in listing 3.7 we must make sure to set A's shadow variables to be equal to B's. We generalise this rule for any dictionary assignment, the size and keys shadow variable of the *lhs* of the assignment must be set equal to the corresponding size and keys shadow variable of the *rhs*.

3.2.3 Range Types

The two range types `a...b` (open) and `a..<b` (half open), define ranges which run from **a** until (and, if half open, excluding) **b**. As the range construct can only be used in for-loops, we consider our translations in this context.

```
let start: Int = 3
let end: Int = 5
let sum: Int = 0
for let i: Int in (start...end) {
    // Loop body
    sum += i
}
```

Listing 3.8: Flint for-loop with range declaration

We translate ranges by counting the from *start* until *end* (or $end - 1$, if half-open), and the loop variable is assigned the value of the counter, as shown in listing 3.9.

```
start := 3;
end := 5;
sum := 0;

loop_counter := start;
while (loop_counter ≤ end) {
    i := loop_counter;

    // Loop body
    sum := sum + i;

    loop_counter := loop_counter + 1;
}
```

Listing 3.9: Boogie translation or for-loop with range

3.2.4 Solidity Types

Currently all the Solidity types which Flint supports, are integers. These are therefore mapped to the Boogie int type. Furthermore, since Flint integers are all 256 bits, we can safely map all Solidity integers into Flint integers without any overflow or truncation issues.

3.3 Constants and variables

Flint's variable declarations are translated directly to Boogie's variable declarations. The Flint **var** construct is semantically the same as Boogie's. A variable **let** declaration, requires that the variable is not changed after assignment. The Flint compiler's semantic analyser guarantees this property for us, and therefore it is safe to translate the **let** declaration into Boogie's **var** declaration.

```
var v: Int
let l: Int
```

Listing 3.10: Flint declarations

```
var v: int;
var l: int;
```

Listing 3.11: Boogie declarations

3.4 Naming

Boogie does not allow variable shadowing therefore, during translation, we name parameters and variables uniquely, to avoid clashes. We do this by adding corresponding suffixes, for each named object's surrounding scope. This allows the generation of unique variable and function names, even for shadowed variables, based on the variables scope. We give an example Flint contract in listing 3.12, and show how we translate each construct's name.

3.5 Functions

To translate Flint functions, we use Boogie procedures. Below we outline in more detail how we translate each component of a function declaration.

3.5.1 Signature

```
@payable
public func deposit(account: Address, amount: implicit Wei) -> Bool
    mutates (balances)
```

Listing 3.13: Flint function signature

```
contract C {
    // global1_C
    var global1: Int = 0
}

C :: (any) {
    // init_C
    public init() {
        // i_init_C
        let i: Int = 0
    }

    // functionInt_C
    func function(param1: Int) { //param1_functionInt_C
        // ...
    }
}
```

Listing 3.12: Flint variable name translation

```
procedure deposit_AddressWei(account_deposit_AddressWei: Int,
                             amount_deposit_AddressWei: Int) returns (returnValue: bool)
    modifies balances;
```

Listing 3.14: Boogie procedure signature

Flint functions, and special functions (initialisers and fallbacks), signatures are translated into Boogie procedure declarations. As shown in listing 3.14, a Boogie procedure declaration contains the procedure name, its argument list, and its return type (and associated return value). To process the function's parameters and return type, the type translation rules from section 3.2 are applied. The ordering of the parameters is maintained to keep the argument ordering of function calls similar to the original Flint source, see section 3.7.2. Function attributes are discarded and function visibility modifiers are discarded, as we do not use them to verify any functional correctness properties.

3.5.2 Naming

In listing 3.14. You may have noticed that the function's name is used verbatim as the procedure name, in particular the function parameter types are also concatenated to the function name. This is done because our translation scheme must be able to translate and distinguish between overloaded Flint functions.

Since Flint functions can be overloaded, unlike Boogie procedures, we must be able to uniquely identify the procedure being called, from only the procedure name. We propose doing this by combining the function name, and appending the types of the function's arguments. This ensures that each generated procedure name is unique, as it is not possible to declare a function with the same name and parameter list, within the same contract.

3.5.3 Mutates

The mutates clause is translated into Boogie modifies clauses. We introduced the `mutates` clause to Flint function declarations for two reasons: to make explicit the contract state that the function is modifying, another way for the contract developer to see the effects of a Flint function; and because Boogie procedures must identify which global variables are modified by the procedure. The translation of the mutates clause requires determining the translated identifier for the global variable which is modified, in listing 3.13 this would be the `balances` variable.

3.5.4 Body

A Flint function declaration is also immediately followed by its body, this is also the case for Boogie procedures. The body of the Flint function is translated, by applying the translation rules for each Flint statement, outlined below 3.11.

```
public func transfer(source: Address, destination: Address) -> Bool
    mutates (balances)
{
    let sourceBalance = balances[source]
    balances[source] = 0

    let destinationBalance = balances[destination]
    destinationBalance += sourceBalance
    balances[destination] = destinationBalance

    return true
}
```

Listing 3.15: Flint function body example

However, Flint local variables may be declared at any point in the function body (listing 3.15), Boogie local variables must be declared at the beginning of the procedure. This requires that our translation scheme collects all the variable declarations and places them at the beginning of the procedure body, as shown in listing 3.16.

Furthermore, we translate a Flint function **return** statement to a special assignment in Boogie. Boogie procedures do not use the **return** statement for procedures to return values, instead you assign the return value of the procedure to a special 'variable' declared in the procedure declaration. This special variable contains the value to be returned by the procedure, therefore when translating the return keyword we look up the declared return result identifier, from the translated function's procedure declaration, and assign the return value to it. This is shown in listing 3.16.

```
procedure transfer_AddressAddressInt(source_transfer_AddressAddressInt: Int,
    destination_transfer_AddressAddressInt: Int) returns (returnValue: bool)
    modifies balances;
{
    // Declaring all local variables
    var sourceBalance: int;
    var destinationBalance: int;

    sourceBalance := balances[source_transfer_AddressAddressInt];
    balances[source] = 0;
    // Omitting shadow variable consistency statements, for brevity

    destinationBalance := balances[destination_transfer_AddressAddressInt];
    destinationBalance := destinationBalance + sourceBalance;
    balances[destination_transfer_AddressAddressInt] := destinationBalance;
    // Omitting shadow variable consistency statements, for brevity

    // return
    returnValue := true;
}
```

Listing 3.16: Boogie procedure body translation example

3.6 Contracts

The contract declaration and protection blocks are the two components which make a Flint contract. In this section we discuss our translation for each component.

3.6.1 Declaration

Global variables

We translate contract global variables into Boogie global variables, using the type translations discussed in section 3.2. Flint global variables can be given initial values as part of their declaration, Boogie has no concept of this. We propose assigning the global initial values in the contract's initialiser function. The contract must be initialised before it is used, at which point the initial values of the contract are set. This requires the translation to take the initial values assigned to each global variable and do the assignment at the beginning of the contract's init function. An example of this is shown in listing 3.17 and 3.18

```
contract Oven {
    var timer: Int
    var power: Int = 180
}

TrafficLight :: (any) {
    public init(seconds: Int) {
        self.timer = seconds
    }
}
```

Listing 3.17: Flint contract with global variables

```
var timer_Oven: int;
var power_Oven: int;

procedure init_Oven(seconds: int)
    modifies time_Oven;
    modifies power_Oven;
{
    // Global variable initial value
    power_Oven := 180;

    timer_Oven := seconds;
}
```

Listing 3.18: Boogie translation of global variables

Type States

We propose translating type states by encoding them into an integer representation. For example, each type state is assigned a unique increasing value.

```
// Red   == 0
// Amber == 1
// Green == 2
contract TrafficLight (Red, Amber, Green) {
    // Variable / event declarations
}
```

Listing 3.19: Flint contract with type states

We also propose the introduction of a global *var stateVariable: int;* which tracks the current type state that the contract is in. This global variable is assigned only the values corresponding to the encoding of the declared type states of the contract. This is discussed in section 3.11.4.

Events

Events are not included as part of our translation scheme.

3.6.2 Protection blocks

As discussed in section 2.3.7, Flint caller protections limit who (and when) certain functions can be called. Caller groups check whether the address of the caller is authorised to call the function; type state protections check that the contract is in the appropriate state. In the following subsections we discuss how we translate these semantics to Boogie.

Caller group

A caller group asserts that the caller is part of a specified group. This is done via the address of the caller and checking whether they are authorised to call the function. Caller groups can be expressed in multiple ways, we outline them and their translations below. However, we first propose modelling the address of the caller with a global variable `caller: Address`. `caller`'s value can be any valid address, which as discussed in section 3.2 are represented as integers. We use the `caller` global variable later when we need to refer to the value of the caller.

Predicate functions have type `(Address) -> Bool`, where the caller's address is passed as the first parameter. The result of the function call is used to determine if the caller has authorisation to call the functions declared in the protection block. An example Flint contract using this functionality is shown in listing 3.20, below.

For the functions who need this authorisation, it would be nice to express this property using a pre-condition of the form shown in listing 3.21.

Intuitively this would require the `lucky_Lottery` procedure to return true, in order for `withdraw_Lottery` to be successfully called. However `lucky_Lottery` is a procedure, the Boogie syntax does not allow procedures to be used in a Boogie `requires` clause, only Boogie functions and expressions.

[1] https://docs.flintlang.org/docs/language_guide#protection-blocks

```
contract Lottery {}

Lottery :: (lucky) {
    func lucky(address: Address) -> Bool {
    // return true or false
    }

    public func withdraw() {
        // Withdraw winnings
    }
}
```

Listing 3.20: Flint predicate function caller group example taken from [1]

```
var caller: int;

procedure lucky_Lottery(address: int) returns (returnValue: bool)
{
    // return true or false
}

procedure withdraw_Lottery()
    requires (lucky_Lottery(caller));
{
    // Withdraw winnings
}
```

Listing 3.21: Ideally use Boogie procedure as pre-condition

Therefore we must find an equivalent approach, which does not use the `requires` clause.

We propose calling the predicate function as a check at the start of the called function. We call the predicate function with value of caller, and test the value the function call returns. If the predicate function returns $false$, we $assume(false)$; to indicate that the transaction would be reverted as the caller was unauthorised. If the predicate function returns true, the execution of the called function continues as normal. An example is shown in listing 3.22.

```
var caller: int;

procedure withdraw_Lottery()
{
    var authorised: bool;
    call authorised := lucky_Lottery(caller);
    if (authorised) {
        // Authorised caller, continue as normal
    } else {
        assume (false);
    }

    // Withdraw winnings
}
```

Listing 3.22: Boogie predicate function translation

0-ary functions have the type `() -> Address`. 0-ary functions have similar semantics to predicate functions, except that the caller's address must match the address that they return. If the caller's address and the address returned by the 0-ary function do not match, the caller is not authorised to call the function and the transaction reverts. Our proposal follows the same line of reasoning that was used for translating predicate functions. Therefore we propose calling the 0-ary function as a check at the start of the called function. We call the 0-ary function and test if the value the function call returns the caller's address. If the test fails, we $assume(false)$; to indicate that the transaction would be reverted as the caller was unauthorised. Otherwise, the execution of the called function continues as normal. An example is shown in listing 3.24.

```
contract Bank {}

Bank :: (onCall) {
    func onCall() -> Address {
        // The person on call
    }

    public func freezeWithdrawals() {
        // Freeze account withdrawals
    }
}
```

Listing 3.23: Flint 0-ary function example

```
var caller: int;

procedure onCall_Bank()
    returns (returnValue: int)
{
    // The person on call
}

procedure freezeWithdrawals_Bank()
{
    var authorised_address: int;
    call authorised_address :=
        freezeWithdrawals_Bank();
    if (authorised_address = caller) {
        // Authorised caller
        // continue as normal
    } else {
        assume (false);
    }

    // Freeze account withdrawals
}
```

Listing 3.24: Boogie 0-ary function translation

State property (single address) requires that the caller's address is equal to the given property. This can be expressed using Boogie `requires` clauses on the translated functions. This approach ensures that the caller's address must match the state property, otherwise the function call will fail.

```
contract Bank {
  let owner: Address
}

Bank :: (owner) {
  public func clearDeposits() {
    // ...
  }
}
```

Listing 3.25: Flint single address state property example inspired by [2]

```
var caller: int;
var owner_Bank: int;

procedure clearDepostits_Bank()
    requires (caller = owner_Bank);
{
    // ...
}
```

Listing 3.26: Boogie single address state property translation

State property (array of addresses) requires that the caller's address is contained as an element within the given property. This can be expressed using Boogie `requires` clauses on the translated functions. This approach ensures that the caller's address is contained within the state property, otherwise the function call will fail.

```
contract Bank {
    var managers: [Address]
}

Bank :: (managers) {
  public func payEmployees() {
    // ...
  }
}
```

Listing 3.27: Flint array of addresses state property example inspired by [3]

```
var caller: int;
var managers_Bank: [int]int;

procedure payEmployees_Bank()
    requires (∃ i: int •
    managers_Bank[i] = caller);
{
    // ...
}
```

Listing 3.28: Boogie array of addresses state property translation

State property (dictionary of addresses) requires that the caller's address is contained as an element within the given property. This can be expressed using Boogie `requires` clauses on the translated functions. This approach ensures that the caller's address is contained within the state property, otherwise the function call will fail.

[2]https://docs.flintlang.org/docs/language_guide#protection-blocks
[3]https://docs.flintlang.org/docs/language_guide#protection-blocks

```
contract Bank {
    var customers: [String: Address]
}

Bank :: (customers) {
  public func withdraw() {
    // ...
  }
}
```

Listing 3.29: Flint dictionary of addresses state property example inspired by [4]

```
var caller: int;
type String;
var customers_Bank: [String]int;

procedure withdraw_Bank()
    requires (∃ c: String •
    customers_Bank[c] = caller);
{
    // ...
}
```

Listing 3.30: Boogie dictionary of addresses state property translation

any as a caller group places no constraints on which addresses are authorised to call the functions. Every address is authorised, therefore we do not place any pre-conditions on the affected translated Flint function.

Multiple caller groups

We must also decide how to translate protection blocks when multiple caller groups are specified. The Flint semantics of this are that if the caller satisfies any one of the listed caller groups then they are authorised to call their function, otherwise the transaction is reverted.

When all of the caller groups are of state properties: single address, array or dictionary of addresses; we propose introducing one pre-condition in a similar fashion to how we translate the single state property caller groups. The difference with multiple, is that we propose combining the pre-conditions with logical OR. This forms a pre-condition where only one of the caller groups must be satisfied for the function to be successfully called, following the Flint semantics.

```
contract Bank {
    let manager: Address
    var accounts: [Address]
}
Bank :: (manager, accounts) {
  func forManagerOrCustomers() {}
}
```

Listing 3.31: Flint multiple state property caller groups example taken from [5]

```
var caller: int;

var manager_Bank: int;
var accounts_Bank: [int]int;

procedure forManagerOrCustomers()
    requires (manager_Bank = caller
∨ ∃ i: int • accounts_Bank[i] = caller);
{
}
```

Listing 3.32: Boogie multiple state property caller groups translation

However, then one or more of the caller groups is a 0-ary function or function predicate, we propose a different approach. As we need to test all of the given caller groups to see if any are satisfied, we must perform these checks in the procedure body not as a procedure pre-condition. This is because the function predicate and 0-ary functions must be tested in the procedure body. Therefore, we propose testing the caller groups in the procedure body and combining the results with logical OR and testing the result of that with an `if` statement to see if the caller satisfies the caller groups. If the if-condition fails, we `assume (false);` as before to model the transaction failing and being reverted.

[4]https://docs.flintlang.org/docs/language_guide#protection-blocks
[5]https://docs.flintlang.org/docs/language_guide#protection-blocks

```
contract Bank {
    let manager: Address
}

Bank :: (manager, onCall) {
    func onCall() -> Address{
        // The person on call
    }

    func resolveIncident() {
        // ...
    }
}
```

Listing 3.33: Flint multiple caller groups example inspired by [6]

```
var caller: int;

var manager_Bank: int;

procedure onCall ()
    returns (returnValue: int)
{
    // The person on call
}

procedure resolveIncident()
{
    var authorised_address: int;
    call authorised_address := onCall();
    if (caller == manager ∨
    authorised_address)
    {
        // Address is authorised
        // continue execution as normal
    } else {
        assume (false);
    }

    // ...
}
```

Listing 3.34: Boogie multiple caller groups translation

Caller group variable

The caller's address can be bound to a variable. We propose the introduction of a local variable, named the same as the caller group variable, and set it's value to *caller*.

```
contract AddressBook {
  var book: [Address: String] = [:]
}

AddressBook :: address <- (any) {
  public func remember(name: String)
  {
    book[address] = name
  }
}
```

Listing 3.35: Flint caller group variable example inspired by [7]

```
var caller: int;
type String;
var book_AddressBook: [int]String;

procedure remember_AddressBook(name: String)
{
    var address: int;
    // Assign caller group variable
    address := caller;

    book_AddressBook[address] := name;
}
```

Listing 3.36: Boogie caller group variable translation

Type state protection

This requires that the contract be in the specified type state, for the function to be successfully called. In a similar fashion to caller groups, we introduce a global variable `stateVariable` which tracks the current type state of the contract. This is discussed further in section 3.6.1. To verify that the contract is in the correct state for the contract to be called, we propose introducing preconditions to the translated Boogie procedure. We require that `stateVariable` is equal to the encoding of the provided type state. If multiple type states are provided, instead we require that `stateVariable` is equal to the encoding of one or the other.

[6] https://docs.flintlang.org/docs/language_guide#protection-blocks
[7] https://docs.flintlang.org/docs/language_guide#protection-blocks

```
contract Poll(Open, CountingVotes,
    Result) {
  // ...
}

Poll @(Open) :: (any) {
  public func voteFor(option:
    String) {
    // ...
  }
}
```

Listing 3.37: Flint type state example taken from [8]

```
// Open = 0
// CountingVotes = 1
// Result = 2
var stateVariable_Pool: int;
type String;

procedure voteFor(option: String)
    requires (stateVariable = 0);
{
    // ...
}
```

Listing 3.38: Boogie type state translation

3.7 Expressions

In this section we outline how we translate Flint expressions to Boogie expressions.

3.7.1 Binary Expression

Flint has many binary operators. We propose translating binary expressions by translating each side of the binary expression into Boogie, using the appropriate rule, and translating the operator using the rules we discuss in section 3.9. This process generates the corresponding Boogie expression.

However, we note that the equals operator (=) is treated slightly differently as it results in a Boogie assignment statement, rather than a Boogie expression. Therefore we must handle the translation of nested assignments, as this is not possible in Boogie. We give an example in listing 3.40. We can see that we must first increment `j`, and then use `j`'s new value to assign to `i`.

```
var i: Int = 0
var j: Int = 2

i = j += 1
```

Listing 3.39: Flint nested example

```
i := 0;
j := 2;

j := j + 1;
i := j;
```

Listing 3.40: Boogie nested example translation

3.7.2 Function calls

We propose translating Flint function calls into Boogie procedure calls. Like Flint function calls, procedure calls transfer the current flow of execution to the body of the called function. Like a Flint function, a Boogie procedure can also return a value which is received by the caller. An example of this is shown in listing 3.41. However, if a Boogie procedure returns a value, it must always be received by the caller. Therefore, even for function calls which do not capture the return value of the function, we must introduce a local temporary variable to capture and discard the result of the function call. An example of this can be seen in listing 3.42.

```
// Call function and capture result
let result: Int = someFunction(10,
    0x0)

// Call function and ignore result
someFunction(10, 0x0)

// Call funcion, which returns
    nothing
someFunctionNoValue(10)
```

Listing 3.41: Flint function call examples

```
var result, temp: int;

// Call function and capture result
call result := someFunctionReturns(10, 0)

// Call function and ignore result
call temp := someFunctionReturns(10, 0)

// Call funcion, which returns nothing
call someFunctionNoValue(10)
```

Listing 3.42: Boogie translation of function calls

[8]https://docs.flintlang.org/docs/language_guide#protection-blocks

3.7.3 Dot access

In this subsection we only talk about how dot access is used to access contract state variables and array or dictionary properties. Dot accesses for structs are discussed here 3.12.2. To maintain the semantics of the dot operator accessing state properties, we propose translating `self.a` into the Boogie global variable which represents the `a` state property. This is shown in listing 3.44.

```
contract Bank {
    var manager: Address;
}

func work() {
    send(self.manager, 100)
}
```

Listing 3.43: Flint state property access

```
var manager_Bank: int;

procedure work_Bank() {
    call send(manager_Bank, 100);
}
```

Listing 3.44: Boogie translation of state property access

When used on arrays and dictionaries, the dot operator can be used to access their size and keys properties. We propose translating these accesses into their returning their shadow variables. An example is given in listing 3.46.

```
let d: [Int: Int]
let a: [Int]

let dKeys: [Int] = d.keys
let aSize: Int = a.size
```

Listing 3.45: Flint size/keys collection access

```
var d, a: [int]int;
// Shadow variables
var size_d, size_a: int;
var keys_d: [int]int;
// Remainder of declarations
var dKeys: [int]int;
var aSize: int;

dKeys := keys_d;
aSize := size_a;
```

Listing 3.46: Boogie translation of size/keys collection access

3.7.4 Subscript Expression

It is possible to index into Flint arrays and dictionaries to access their elements. The Boogie language has the same syntax and semantics, for their subscript expression, as Flint. Therefore we propose translating the subscript expression by first translating its base first, the base should have the type of array or dictionary. We then translate the index expression, and combine them using Boogie's subscript expression syntax.

```
let js: [[Int]] = [[0, 1]]

let j: Int = js[0][1]
```

Listing 3.47: Flint subscript expression example

```
var js: [int]int;
var j: int;

js[0][0] := 0;
js[0][1] := 1;

j := js[0][1];
```

Listing 3.48: Boogie subscript expression translation

3.7.5 Attempt

The attempt expression is used to perform dynamic checking of caller protections at run-time. There are two versions of the attempt expression: `try?`, which returns a boolean indicating whether the function call was successful; and `try!` which reverts the transaction if the call fails. Whether a call is successful is determined by whether the function's protections are satisfied. We propose testing the protections of the called function before deciding to perform the function call, as part of our translation. In particular, we take the target function's procedure translation and extract it's pre-conditions. These pre-conditions include the protections given to the function and so if the caller can satisfy them the function call proceeds, otherwise the function call fails. If the function call fails, either the expression returns false or we model the transaction failing by inserting `assume (false);`, based on whether `try?` or `try!` was used.

```
Bank :: (any) {
  func foo() {
    let result: Bool = try? bar()

    try! bar()
  }
}

Bank :: (manager) {
  func bar() {}
}
```

Listing 3.49: Flint try expression example taken from [9]

```
var caller: int;

procedure foo_Bank()
{
    var result: bool;

    // try?
    // Test if function protections are
    satisfied
    if (caller == manager) {
        result := true;
        call bar_Bank();
    } else {
        result := false;
    }

    // Test if function protections are
    satisfied
    if (caller == manager) {
        call bar_Bank();
    } else {
        // Call fails
        assume (false);
    }
}

procedure bar_Bank()
    requires (caller == manager);
{
    // ...
}
```

Listing 3.50: Boogie try expression translation

3.8 Literals

Basic types

Literal	Flint Representation	Boogie Representation	Explanation
Integer	10	10	Use the same representation
Address	0x10	16	Convert hex into base 10
Boolean	true	true	Use the same representation
String	-	-	Not supported

Dynamic types

From section 3.2 our translation scheme translates dictionaries and arrays to Boogie maps. Map literals do not exist in Boogie, therefore we must split the Flint literal into individual assignments into a temporary map, which represents the Flint array or dictionary literal. The temporary map must also have size and/or keys shadow variables depending on the type of literal, as discussed in section 3.2.

```
//[0, 1, 2]
tmp_array_literal[0] = 0
tmp_array_literal[1] = 1
tmp_array_literal[2] = 2
size_tmp_array_literal = 3
```

Listing 3.51: Array literal translation

```
//[0x0: 0, 0x10: 10, 0x20: 20]
tmp_dictionary_literal[0] = 0
tmp_dictionary_literal[16] = 10
tmp_dictionary_literal[32] = 20
size_tmp_dictionary_literal = 3
keys_tmp_dictionary_literal[0] = 0
keys_tmp_dictionary_literal[1] = 10
keys_tmp_dictionary_literal[2] = 20
```

Listing 3.52: Dictionary literal translation

Self

We do not include the Self literal in our translation scheme, as the Self literal (which is part of the trait construct) does not exist in the Flint representation of the smart contract, that our translator operates on. This is discussed in our translation of traits, section 3.13.

[9]https://docs.flintlang.org/docs/language_guide#dynamic-checking

3.9 Operators

We translate the Flint operators into the equivalent Boogie operators. We summarise our proposed translations below:

3.9.1 Arithmetic Operators

The tables below show the translation of each Flint operator, into Boogie. The `**` and unsafe operators are not natively supported by Boogie. In listing 3.53, we outline the function we implement to translate the Flint power operator. Using this Boogie helper function, we can replace every every occurrence of `a ** b`, with the call `power(a, b)`. As shown below, the unsafe operators are translated into the equivalent safe operator translation and their result is modulo 2^{255}, which is Ethereum's max integer size. The safe operators guarantee not to cause integer over or underflows, due to run-time checks.

Name	Flint	Boogie
Addition	$+$	$+$
Subtraction	$-$	$-$
Multiplication	$*$	$*$
Division	$\backslash$	**div**
Modulo	%	**mod**
Exponentiation	$**$	`power(n, e)`

Table 3.1: Safe operators

Name	Flint	Boogie
Addition	`a &+ b`	`(a + b) mod power(2, 255)`
Subtraction	`b &- b`	`(a - b) mod power(2, 255)`
Multiplication	`a &* b`	`(a * b) mod power(2, 255)`

Table 3.2: Unsafe operators

```
function power(n: int, e: int) returns (i: int);
axiom (∀ n: int • power(n, 0) = 1); // Base case

// Recursive cases
axiom (∀ n, e: int • (e mod 2 = 0 ∧ e > 0) ⟹
    (power(n, e) = power(n, e div 2)*power(n, e div 2)));

axiom (∀ n, e: int • (e mod 2 = 1 ∧ e > 0) ⟹
    (power(n, e) = n*power(n, (e−1) div 2)*power(n, (e−1) div 2)));
```

Listing 3.53: Boogie representation of the power operator

3.9.2 Boolean operators

The Flint and Boogie boolean operators share the same semantics as each other, therefore we propose the trivial translations from Flint to Boogie.

Name	Flint	Boogie
Equal to	$==$	$==$
Not equal to	$!=$	$!=$
Logical or	$\|\|$	$\|\|$
Logical and	&&	&&
Less than	$<$	$<$
Less than or equal to	$<=$	$<=$
Greater than	$>$	$>$
Greater than or equal to	$>=$	$>=$

Table 3.3: Boolean operators

3.10 Enumerations

Flint enumeration declarations are translated into Boogie `const unique` declarations, each case is separated into a unique const declaration. The semantics of the unique const declarations are such that every `const unique` is guaranteed to have a value different to all the other unique const declarations. We use these declared constants to replace any occurrence of the an enumeration's case.

```
enum CompassPoint: Int {
  case north = 1
  case south = 2
  case east = 3
  case west = 4
}
```

Listing 3.54: Flint enumeration declaration taken from Flint documentation [10]

```
type CompassPoint = int;
const unique north_CompassPoint:
    CompassPoint;
const unique south_CompassPoint:
    CompassPoint;
const unique east_CompassPoint:
    CompassPoint;
const unique west_CompassPoint:
    CompassPoint;
```

Listing 3.56: Enumeration translation

```
let direction: CompassPoint =
    CompassPoint.north
```

Listing 3.55: Flint enumeration usage taken from Flint documentation [11]

```
var direction: CompassPoint;
direction := north_CompassPoint;
```

Listing 3.57: Enumeration usage translation

3.10.1 Associated values

Flint enumerations can also have associated values, with which the enumeration case can be used to represent that value. Our translation scheme also provide these semantics for associated values, only of type int. This is achieved through Boogie axioms, which express the value of the constant, we give an example in listing 3.58.

```
axiom (north_CompassPoint = 1);
axiom (south_CompassPoint = 2);
axiom (east_CompassPoint = 3);
axiom (west_CompassPoint = 4);
```

Listing 3.58: Enumeration associated value

3.11 Statements

3.11.1 Assignment

Flint permits the assignment of expressions to variables. Our translation scheme uses the Boogie assignment statement to perform the same function.

```
let counter: Int = 10
```

```
var counter: int;
counter := 10;
```

Therefore our general translation scheme for assignments is to translate the expression and then assign it's value to the variable.

Compound assignment

However it may not always be possible to translate from one Flint statement to one Boogie statement, take for example the compound operators: *+=*, -=, **=*, */=*.
These operators increment, decrement, multiply and divide their variable during assignment; but they also return the new value of the variable. Take the example in listing 3.59 below.

```
1 let i: Int = 10
2 let j: Int = (i += 10)
3
4 i = (i += 1) + (j  /= 2) + 3
```

Listing 3.59: Flint compound assignment

The semantics of line 2 require that i first be incremented by 10 and then assigned to j. This type of assignment cannot be expressed in one statement in Boogie, therefore we propose splitting the assignment into multiple stages. First we increment i, in one statement, and then we assign it's value to j. See listing 3.60 We generalise this approach such that any expression which requires pre-statements are placed before the assignment.

[11] https://docs.flintlang.org/docs/language_guide#enumerations

```
var i: int;
var j: int;

// j = (i += 10)
i := 10;
i := i + 10;
j := i;

// i = (i += 1) + (j  /= 2) + 3
i := i + 1;
j := j div 2;
i := i + j + 3;
```

Listing 3.60: Boogie compound assignment translation

3.11.2 Loops

The only looping construct Flint has is the for-loop, it can loop through ranges, arrays and dictionaries. We discussed how we translate looping through ranges in section 3.2.3. Here we discuss looping through arrays and dictionaries.

```
let is: [Int] = [0, 1, 2]
let count: Int = 0
for let i: Int in is {
    count += i
}
```

Listing 3.61: Flint looping through array example

```
let ds: [Address: Int] = [0x0: 0,
    0x1: 1, 0x2: 2]
let count: Int = 0
for let a: Address in is {
    count += ds[a]
}
```

Listing 3.62: Flint looping through dictionary example

Boogie's looping construct is the while loop, it is possible to translate any for loop to a while loop of the form:

```
counter := initial_value
while (counter < final_value) {
    iteration_variable = iterator_value(counter)

    // Loop body

    counter := update(counter)
}
```

We use this pattern to translate Flint for-loops into Boogie while-loops, changing only the *iterator_value* and *update* functions, depending on whether we're iterating through arrays or dictionaries.

To iterate through arrays, the *initial_value* of the loop counter is zero, as we start iterating from the beginning of the array, and the *final_value* is the size of the array. The size of the array is tracked by the size shadow variable (see section 3.2.2).

We update the counter by incrementing it by 1. The *iteration_variable* value is found by using the counter to index into the array. This is shown is listing 3.63.

```
count := 0;
counter := 0;
while (counter < size_is) {
    i := is[counter];

    // Loop body
    count := count + i;

    counter := counter + 1;
}
```

Listing 3.63: Boogie translation of iterating through array

For dictionaries, we cannot iterate through the dictionary directly, we don't know what the 'next' key for an arbitrary type should be. Therefore we use the size shadow variable to iterate through the dictionary's keys shadow variable. Shown in listing 3.64.

```
count := 0;
counter := 0;
while (counter < size_ds) {
    a := keys_ds[counter];

    // Loop body
    count := count + ds[a];

    counter := counter + 1;
}
```

Listing 3.64: Boogie translation of iterating through dictionary

3.11.3 Conditionals

Flint if-statements are translated to the Boogie if statements. The semantics of both constructs are the same. Therefore we translate the if-condition, using the rules discussed in section 3.7, and we translate bodies of the true and false cases, using the rules discussed in section 3.11.

```
if condition {
    // True case
} else {
    // False case
}
```

Listing 3.65: Flint if-statement

```
if (condition) {
    // True case
} else {
    // False case
}
```

Listing 3.66: Boogie if-statement translation

3.11.4 Become statements

Become statements modify the type state of Flint smart contracts. As discussed in section 3.6.1, we propose modelling types states through the use of a global state variable. Therefore the become statement is translated to an update of the global state variable.

The new value the state variable is assigned is found by translating the new type state. For example, listings 3.67 and 3.68 show the translation of a `become` statement, within a traffic light smart contract. As part of translating the type state declaration of the contract, we assume that the `Green` state has been encoded to a value of 0.

```
become Green
```

Listing 3.67: Flint become statement

```
// State value of 0 corresponds to Green
stateVariable_TrafficLight := 0;
```

Listing 3.68: Boogie become-statement translation

3.11.5 Return statements

Boogie procedures do not use the return statement as a way of specifying the value the procedure returns, instead it only indicates the end of the execution path. To specify the value that the function returns, instead you must assign the return value into the `returnVariable`, which is part of the procedure declaration.
Therefore `return` statements are translated into assignments of the `returnVariable`, as shown in listing 3.70.

```
func ten() -> Int {
    return 10
}
```

Listing 3.69: Flint return statement

```
procedure ten() returns (returnVariable: int) {
    returnVariable := 10;
    return;
}
```

Listing 3.70: Boogie return statement translation

3.11.6 Do-Catch statements

Flint Do-Catch statements catch exceptions, currently only external calls can throw exceptions [12]. Do-catch blocks attempt to execute each statement inside their *do*-block, if a statement throws an exception then the series of *catch*-block statements are executed.

Boogie has no concept of throwing or catching exceptions. Therefore we model the behaviour of the catch-block directly. We propose first translating the catch-block of the statement, and temporarily holding it, then for every statement within the do-block we test to see if it has failed. If the statement fails we execute the catch-block, else we continue executing statements in the do-block. This scheme ensures that only the successful statements are executed until the exception is thrown, after this point only the catch-block statements are executed and execution continues from after the do-catch statement. We model whether the potentially exception-throwing statement throws an exception by introducing a local boolean variable and `havoc`'ing it's value.

Listings 3.71, 3.72 show an example translation of the do-catch block. We discussed the translation of external calls in section 3.14.

```
let callFailed: Bool = false
do {
  call extContract.someFunction()
} catch is ExternalCallError {
    callFailed = true
}
```

Listing 3.71: Flint do-catch with external call example

```
// translation of exception throwing statement
var callFailed, exception_thrown: bool;
callFailed := false;

// Translation of call function
havoc global_state;

havoc exception_thrown;
if (exception_thrown) {
    // Exception was thrown, execute catch-block
    callFailed := true;
} else {
    // No exception thrown, continue executing remainder of do-block
}
```

Listing 3.72: Boogie translation of do-catch block

3.12 Structs

In the following subsections we discuss how we translate Flint structs, and their uses, into Boogie.

3.12.1 Declaration

Struct declarations themselves contains two further declarations, variable declarations and function declarations. The variable declarations specify the state which is associated with the struct, and the function declarations specify the methods which can be called on the state. Structs are pass-by-reference in Flint.

Struct fields

To represent structs in Flint, we use an approach not unlike modelling the heap, in an object oriented language. We use Boogie's map constructs to represent the address space used for struct objects. We do not use one map to represent the entire struct memory space, as this would only allow us to represent structs containing fields of only the same type. Boogie maps must map to values of one type. Therefore representing a struct containing two fields, one integer and one boolean value, would require encoding the boolean as an integer. This theme would continue for all fields in a struct, so they could be represented in the struct memory space. This is an approach we could have taken, it would have required us to introduce a struct layout scheme and axioms and checks to make sure that the correct type is inserted into the location of each struct field.

[12] https://docs.flintlang.org/docs/language_guide#do-catch-blocks

We instead opted for a simpler approach which did not require the encoding of Boogie to Boogie types, or an explicit struct layout scheme. We propose representing the memory space of each struct field separately. That is to say, for each field in a struct, we create a memory space which contains the value of that field for every struct instance. A struct instance is able to look-up the values of their fields, by searching in the global variables which represent the memory spaces for each of their fields.

```
struct S {
    var i: Int = 10
    var j: Bool = false
}
```

Listing 3.73: Flint struct example

```
var i_S: [int]int;
var j_S: [int]bool;

var nextInstance_S: int;
```

Listing 3.74: Boogie translation of struct

In order to determine the values of a struct instance's fields, a struct instance identifier is required. The instance identifier allows for unique look-up of the struct instance's fields. The instance identifier is analogous to pointers from languages such as C++. The pointer is used to access the locations in memory which contain the object's fields. We discuss how we create a struct instance identifier in the following subsubsection. In our case, to represent Flint structs we introduce global maps for each struct field to represent the memory spaces for each struct. We also introduce a global variable *nextInstance* which tracks the size of the struct memory space, or equivalently the number of created struct instances. Therefore the struct instance identifier is directly used to index into the global variables which represent the memory spaces for each of the struct's fields.

Struct methods

When it comes to translating the struct methods, we need to be able to access the struct fields corresponding to the particular instance of the struct that we are calling the method on. To do this we add an extra parameter to the function, *structInstance*, which is used to look-up the fields for that struct. Therefore, struct method calls must pass this extra argument to indicate which struct instance they intend to call the method on.

Furthermore, when translating a struct's method body, whenever a struct field is referenced, we must return the value of the field corresponding to the struct instance that the method was called on. Therefore we propose using the procedure's struct instance argument to look up the field's value in the corresponding field's memory space. This ensures we maintain the semantics of Flint struct field accesses.

```
struct A {
    var b: B

    func bValue() -> Int {
        return self.b.i
    }
}

struct B {
    var i: Int
}
```

Listing 3.75: Flint example of struct method

```
// Field memory space
var b_A: [int]int;
var i_B: [int]int;

// Struct methods
procedure bValue_A(structInstance: int) returns
    (returnValue: int)
{
    returnValue := i_B[b_A[structInstance]];
    return;
}
```

Listing 3.76: Boogie translation of a struct method

We must also propose a translation for struct constructors, which are slightly different to other struct methods. Struct constructors do not operate on a struct instance, a new instance must be created, to maintain the semantics of the struct constructor. We propose using the Boogie global variable *nextInstance* to allocate a new struct instance. Since *nextInstance* tracks the current number of instantiated structs, we can guarantee that it will always point to a struct instance value which isn't being used by another struct instance. Therefore, we propose that during struct constructor translation a new struct instance is reserved, using *nextInstance*, and at the end of the constructor call, the value of *nextInstance* is incremented to reflect that a new struct has been created. The struct's values are initialised as per the constructor's body and the new struct's instance variable is returned by the constructor procedure to it's caller.

```
struct A {
    var i: Int

    init() {
        self.i = 0
    }
}
```

Listing 3.77: Flint example struct init method

```
// Field memory space
var i_A: [int]int;
var nextInstance_A: int;

// Struct methods
procedure init_A() returns (newStruct:
    int)
    modifies nextInstance_A;
{
    var newInstance: int;
    newInstance := nextInstance_A;

    i_A[newInstance] := 0;

    nextInstance_A := nextInstance_A + 1;

    newStruct := newInstance;
    return;
}
```

Listing 3.78: Boogie translation of struct init method

3.12.2 Instances

Here we discuss further the struct instance identifier, and how it is used to maintain the semantics of calling struct methods and accessing struct fields.

In Flint, with any instance of a struct, you can call it's declared methods or access its fields. For us to translate these behaviours to Boogie, we must first know the particular struct instance whose fields we are accessing, or methods we are calling. Therefore, we first begin by proposing a translation of struct types. Struct variables are translated into Boogie `int` types, as shown in listing 3.80, where the variable is used to contain the instance value of that particular instantiation. In the Boogie translation of struct variables, rather than the variable containing the struct directly, the variable contains the instance value of the struct. This instance value, which as discussed in section 3.12.1 is generated by the constructor, can then be used to call methods on the struct instance or access it's fields.

```
let w: Wei = Wei(0)
```

Listing 3.79: Flint example of a struct variable

```
var w: int;
call w := init_Wei(0);
```

Listing 3.80: Boogie translation of struct variable

Accessing properties

We propose accessing struct properties as follows. The struct instance variable can directly be used to index into the global mapping used to represent the field's memory space.

```
let w: Wei = Wei(0)
let v: Int = w.rawValue

// ...

struct A {
    var b: B
}
struct B {
    var i: Int
}
// ‘a’, an instantiation of struct A
let i: Int = a.b.i
```

Listing 3.81: Flint example of accessing struct fields

```
// Field memory space
var rawValue_Wei: [int]int;
var b_A: [int]int;
var i_B: [int]int;

// Accessing fields
var w, v, a, i: int;
call w := init_Wei(0);
v := rawValue_Wei[w];

i := i_B[b_A[a]] ;
```

Listing 3.82: Boogie translation of accessing struct fields

Calling struct methods

As discussed in section 3.12.1, we proposed the addition of the struct instance argument to the generated procedures so the procedure is modifying the fields of the particular struct instance that the method was called on. Here we provide a concrete example of how this.

```
struct A {
    var b: B

    func bValue() -> Int {
        return self.b.i
    }
}

struct B {
    var i: Int
}
// ‘a’, an instantiation of struct A
let i: Int = a.bValue()
```

Listing 3.83: Flint example of calling a struct method

```
// Field memory space
var b_A: [int]int;
var i_B: [int]int;

// Struct methods
procedure bValue_A(structInstance: int)
    returns (returnValue: int)
{
    returnValue := i_B[b_A[structInstance]];
    return;
}

// Accessing fields
var a, i: int;

call i := bValue(a);
```

Listing 3.84: Boogie translation of calling a struct method

3.12.3 Structs as function arguments

Finally, Flint structs can be passed as function arguments. The & operator is used to indicate that a reference to the struct is being passed, the receiving function requires an **inout** parameter to accept the struct reference. The struct instance scheme, described above, fits this situation very nicely as it is a pass-by-reference scheme by construction. To translate a struct being passed by reference into a function we, simply pass the struct's instance variable as it's argument instead. The receiving function, accepts the struct's instance variable and can use it to make in-place changes to the struct, as Flint's semantics require. An example is shown in listing 3.86.

```
struct A {
    var i: Int

    init() {
        self.i = 0
    }
}

func increment(a: inout A)
    mutates (A.i)
{
    a.i += 1
}

let a: A = A()
increment(&a)
assert (a.i == 1)
```

Listing 3.85: Flint example of passing a struct by reference

```
// Field memory space
var i_A: [int]int;
var nextInstance_A: int;

// Struct methods
procedure init_A()
    modifies nextInstance_A;
{
    var newInstance: int;
    newInstance := nextInstance_A;

    i_A[newInstance] := 0;

    nextInstance_A := nextInstance_A + 1;
}

// Contract methods
procedure increment(a: int)
    modifies i_A;
{
    i_A[a] := i_A[a] + 1;
}

// Passing by referece
var a: int;
call a := init_A();
call increment(a);
assert (i_A[a] = 1);
```

Listing 3.86: Boogie translation of passing a struct by reference

3.13 Traits

We do not propose a translation scheme for the trait construct. As discussed in section 2.3.2, traits are not stand-alone constructs themselves as they are implemented by contracts and structs. Therefore we propose a translation for the resultant contracts and structs. This is discussed in sections 3.6 and 3.12. The Flint compiler ‘resolves’ traits into the conforming structs and contracts, as part of the compilation process. Therefore, traits can be transparent to our translation scheme. We discuss further how we implement the translation of traits in a way which is transparent to the verifier, in section 5.2.4.

As traits can contain contract functions and struct methods, these constructs still benefit from pre and post-conditions even though they are in traits. The only limitation is that the pre and post-conditions can express no properties about global state, as traits contain no global state. This is the same reason we do not have trait invariants. The pre and post-conditions are preserved when adding the trait property to to the struct or contract implementation, thus these pre and post-conditions are also transparent to the verifier.

```
external trait ExternalBank {
  @payable
  func pay() -> int256
  func withdraw(amount: int256) -> int256
    pre (amount > 0)
}
```

Listing 3.87: Example external trait with pre-conditions[13]

As shown above, pre and post-conditions which are added to external contracts are verified on entry and assumed on the return of external function calls. In a similar method to what we discuss below, the verifier asserts that the pre-conditions to any external functions must hold, and is free to assume that the post-condition of the function holds also. The compiler performs a run-time check to make sure that the function's post-condition holds, otherwise it reverts the transaction.

3.14 External Calls

As introduced in section 2.3.4, Flint external calls allow for the smart contract to invoke functions on other smart contracts deployed to the Ethereum blockchain. It is possible for a called external contract to call the public functions of the callee smart contract. This allows an external contract to change the state of the called contract. Therefore our translation scheme models this behaviour by calling `havoc` on each of the global contract variables. Local variables cannot be affected as they are stored in the stack frame of the currently executing transaction, therefore we don't call `havoc` on these variables. `havoc`'ing the global variables tells the verifier to assign fresh unknown values to the variables, and models external calls as though they modify all of the contract's global state. To make sure that the new variables are valid values, we immediately assume the contract's invariant after the `havoc`'ing the contract's global state to restrict their values to only valid ones. We can do this, because we can guarantee that the contract's invariant will hold after the external call, if the contract's invariant holds immediately before the external call (see an example of this in listing 3.89). This is because every Flint function guarantees the contract invariant will hold as a post-condition, assuming it holds on function entry. We make sure to satisfy this requirement by asserting that the contract's invariant holds at the external call location. We introduce and discuss contract invariants in detail, in section 4.1.3.

Furthermore, it is possible that external functions can fail. When called, the external function could throw an exception and the call to the external contract will be reverted. Depending on the type of external call, Flint will handle external call failure differently. If the external call is done using `call`, it must be contained within a `do-catch` block and the catch clause will be executed, if the external call fails. Failure is modelled by introducing a local boolean variable which is assigned a non-deterministic value, via `havoc`. Handling failure within a `do-catch` block is discussed in more detail in section 3.11.6. However, if the call is attempted using `call!` and the call fails, the calling transaction also fails. We model this behaviour by testing for failure and inserting `assume (false)`, to stop the verifier considering this execution path as the transaction would have been reverted.

Finally, as with any other Flint function call, it is possible to call the external function with arguments and to capture the function's return value. We propose that this is translated in the same way as non-external function calls in Flint, 3.7.2. Listing 3.89 gives a concrete example of how we propose translating Flint external calls.

```
external trait ExternalContract {}
contract C {
    var count: Int = 10
```

[13]Inspired by https://docs.flintlang.org/docs/language_guide#specifying-the-interface

```

    invariant (count == 10)
}

C :: (any) {
    func increment() {
        let externalContract = ExternalContract(address: 0x0)
        call! externalContract.someFunction()
    }
}
```

Listing 3.88: Flint external function call

```
var count_C: int;

procedure increment_C()
  // Invariant translation
  requires (count_C = 10);
  ensures (count_C = 10);
{
  var call_succeeded: bool;

  // Make sure contract invariant holds at external call location
  assert (count_C = 10);
  havoc call_succeeded;
  if (¬call_succeeded) {
      // Caller transaction is reverted
      assume (false);
  }
  // All global variables could have been modified during the external call, via contract
    rentrancy
  havoc count_C;

  // Can guarantee that that after the external call, the contract invariant will still
    hold
  assume (count_C = 10);
}
```

Listing 3.89: Boogie translation of external function call

3.14.1 Casting to and from Solidity types

External calls with external Solidity contracts, these contracts use Solidity integers which are smaller than the 256-bit integers that Solidity uses. Therefore these type conversions must be made explicit. As we are always casting between integers, and Boogie represents the Flint and Solidity integers in the same way, no special handling is required for this type casting and we are free to ignore it.

3.15 Standard Library

Flint comes with a standard library, of public functions and structs which can be used in a Flint smart contract. In this section we discuss how we translate each of the standard library functionalities.

3.15.1 Assets

Currently assets are implemented in the standard library using traits and structs. Our translation scheme does not distinguish between assets and any other use of structs, see section 3.13, therefore no special translation is required for the Asset trait.

3.15.2 Global functions

Assertions

`assert` is translated into Boogie's `assert`. The verifier will attempt to prove the property used as an argument. Assert is called with the Boogie translation of the Flint expression as it's argument.

Fatal error

We translate the Flint `fatalError()` function call into `assume (false)`. In Boogie, this indicates to the verifier that it should stop exploring this path. If the contract reaches this line of code, the contract execution is reverted to its previous state. We see how having the verifier stop exploring this path preserves these semantics, by emulating how the execution of the actual smart contract is reverted.

Send

We implement our own version of `send` which receives the Wei struct and the account the money should be sent to, and empties the money from the Wei struct.

```
procedure send(account: int, source: int)
    modifies (rawValue_Wei);

    ensures (rawValue_Wei[source] = 0);
{
    rawValue_Wei[source] := 0;
}
```

Listing 3.90: Boogie representation of send function

Note that `send` is an external call by another name. Therefore for any functions which call `send`, we apply the same approach as we describe in section 3.14. For example, we assert that the contract invariant holds before and we assume that it holds after the call.

3.16 Remarks

We discussed how we can translate each Flint construct into a semantically equivalent Boogie representation. By recursively applying each of the rules we described in this section we can theoretically translate any Flint smart contract into Boogie. In appendix E find a summary of each Flint construct and its translation.

In the next section we discuss the specification syntax we introduce to Flint. We show how we can write specifications of the Boogie representation of Flint contracts, using the translation scheme described here.

Chapter 4

Mitigating contract immutability

As discussed in section 2.2.1, smart contracts deployed on the Ethereum blockchain are immutable, which makes it important to ensure the correctness of your smart contract before deployment. In this section, we introduce functional and holistic specifications to Flint. We also introduce other methods which can be used to detect and prevent immutability bugs.

4.1 Functional Specifications

In this section, we introduce functional specification syntax to Flint, and discuss how functional specifications are translated into Boogie, so they Flint contracts can be verified against them. Specifically, we introduce pre and post-conditions for Flint functions, as well as struct and contract invariants.

4.1.1 Pre/Post-Conditions

We introduce pre and post-conditions to Flint functions by adding pre and post clauses to function declarations. Each pre/post-condition can express properties about either the global contract state or the function parameters, using the normal Flint expression syntax. We give an example in listing 4.1.

```
func fibonacci(n: Int) -> Int
    pre (n >= 0)
{
    if n < 2 {
        return n
    }
    return fibonacci(n-1)
         + fibonacci(n-2)
}
```

Listing 4.1: Flint pre condition example

```
procedure fibonacci(n: int)
    returns (return_value: int)

    // Translated pre-clause
    requires (n ≥ 0);
{
    var fibn1, fibn2: int;
    if (n < 2) {
        return_value := n;
        return;
    }
    call fibn1 := fibonacci(n−1);
    call fibn2 := fibonacci(n−2);
    return_value := fibn1 + fibn2;
    return;
}
```

Listing 4.2: Boogie pre condition translation

Flint `pre` and `post` clauses can be directly translated into the equivalent Boogie pre and post-condition clauses, `requires` and `ensures`. This is shown in listing 4.2.

Structs

When translating pre/post-conditions for struct methods we propose a very similar approach to contract functions. However, we must handle references to struct fields in the pre and post-conditions. The struct field references must be translated to the fields of the struct instance that the struct method is being called on. We do this by translating the field references using the approach discussed in section 3.12.2. An example is shown in listing 4.4.

```
struct A {
    var tens: Int = 10

    func add(i: Int)
        mutates (tens)
        pre (i % 10 == 0)
        post (tens % 10 == 0)
    {
        tens += i
    }
}
```

Listing 4.3: Flint struct pre/post-condition example

```
var tens_A: [int]int;

procedure add_A(structInstance: int, i: int)
    modifies tens_A;
    requires(i mod 10 = 0);
    ensures (tens_A[structInstance] mod 10 =
    0);
{
    tens_A[structInstance] :=
    tens_A[structInstance] + i;
}
```

Listing 4.4: Boogie struct pre/post-condition translation

4.1.2 Specification Predicates

We don't only introduce pre and post-condition syntax to Flint, we also introduce specification predicates. These predicates can be used to express logical properties, which can't otherwise be expressed using only Flint's expression syntax.

returns keyword

The `returns` keyword can be used only in the context of a Flint post-condition and is used to describe a property of the return value.

In particular we introduce `returns` to allow the function specification to logically express the value that the function returns. For example, listing 4.5 shows how we assert the result of a game of Rock Paper Scissors.

```
func RPS(s1: Int, s2: Int) -> Bool
  // ROCK == 0
  // PAPER == 1
  // SCISSORS == 2
  pre (s1 >= 0 && s1 <= 2)
  pre (s2 >= 0 && s2 <= 2)
  post (returns ((s1 == 0 && s2 == 2)
             || (s1 == 1 && s2 == 0)
             || (s1 == 2 && s2 == 1)))
{
  var outcome: Int = ((3 + s1 - s2))
  return (outcome % 3 == 1) // S1 win?
}
```

Listing 4.5: Flint returns example[1]

```
procedure RPS(s1: int, s2: int)
  returns (result: bool)
  // ROCK = 0
  // PAPER = 1
  // SCISSORS = 2
  requires (s1 ≥ 0 ∧ s1 ≤ 2);
  requires (s2 ≥ 0 ∧ s2 ≤ 2);
  ensures (result = ((s1 = 0 ∧ s2 = 2)
          ∨ (s1 = 1 ∧ s2 = 0)
          ∨ (s1 = 2 ∧ s2 = 1)));
{
  var outcome: int;
  outcome := ((3 + s1 − s2));
  result := (outcome % 3 = 1);
  return;
}
```

Listing 4.6: Boogie returns translation

We translate the returns keyword into Boogie by testing if the procedures result variable (namely `result` in listing 4.6) is equal to the `returns` value provided. This is shown in listing 4.6.

returning predicate

The `returning()` predicate also allows the contract author to express more complex properties about the return value of a function. The `returning(returnValue, expression)` predicate takes two arguments, a variable representing the return value of the function, and a Flint expression expressing a property of the return value. We give a simple example below.

```
func positiveValue() -> Int
  post (returning(r, r > 0))
{
    return 10
}
```

Listing 4.7: Flint returning example

```
procedure positiveValue() returns
    (result_value: int)
  ensures (result_value > 0);
{
  result_value := 10;
  return;
}
```

Listing 4.8: Boogie returning translation

[1]Inspired by: https://github.com/YianniG/flint/blob/master/examples/casestudies/RockPapersScissors.flint

Since the `returning` predicate expresses a property about the return value of a function, it can only be used within a post-condition. Furthermore, when translating the predicate into Boogie, we translate the predicate's expression argument and simply replace the provided return value argument, with the Boogie procedure's return variable name.

arrayContains / dictContains predicate

`arrayContains` and `dictContains` predicates, allow smart contract authors to express that an array, or dictionary, contains a value. `arrayContains(array, value)` and `dictContains(dict, value)` predicates, both take two arguments, the array/dictionary in question, and the value we wish to assert is contained within them. `dictContains` asserts that the dictionary contains a key equal to the value, and `arrayContains` asserts that the array contains an element with the value. We give an example below.

```
func getBalance() -> Int
pre (dictContains(balances, account))
{
  return balances[account]
}
```

Listing 4.9: Flint contains predicate example

```
procedure getBalance()
  returns (raw_value: int)

requires (∃ i:int • 0≤i ∧ i<size_balances
      ⟹ balances[i] = account);
{
  raw_value := balances[account]
}
```

Listing 4.10: Boogie contains predicate translation

For `arrayContains`, we translate the predicate into the following:
`(exists i: int :: 0 <= i && i < size_array ==> array[i] == value).`

Similarly for `dictContains`:
`(exists i: int :: 0 <= i && i < size_dict ==> keys_dict[i] == value).`

arrayEach predicate

The `arrayEach` predicate, allows the contract author to specify a property which must hold for each element within the array. The `arrayEach(element, array, expression)` predicate takes three arguments, name of the variable used to represent the value of each element in the array; the array whose elements we are expressing the property over; and the expression which must hold over each element of the array. We give an example below.

```
func positiveInput(array: [Int])
pre (arrayEach(e, array, e > 0))
{
  // ...
}
```

Listing 4.11: Flint arrayEach predicate example

```
procedure positiveInput(array: [int]int,
                        size_array: int)
  requires (∀ i: int • 0 ≤ i ∧ i < size_array
      ⟹ array[i] > 0);
{
  // ...
}
```

Listing 4.12: Boogie contains predicate translation

For `arrayEach`, we can see that we translate the predicate into a formula of the form:
`forall i: int :: 0 <= i && i < size_array ==> property with respect to {array[i]}.`

4.1.3 Invariants

Pre and post-conditions, provide a method of specifying the behaviour of individual functions. We introduce invariants as a way of specifying behaviour which must hold after the invocation of any function in the smart contract.

Contract invariants are specified in contract declarations, as shown in listing 4.13, and can express properties over the global contract state.

```
contract Counter {
    var count: Int = 0

    invariant (count >= 0)
}

Counter :: (any){
    public init() {}

    public func increment()
        mutates (count)
    {
        self.count += 1
    }

    public func count() -> Int {
        return count
    }
}
```

Listing 4.13: Flint contract invariant example

```
var count_Counter: int;

procedure init_Counter()
    ensures (count ≥ 0);
{
    count_Counter := 0;
}

procedure increment_Counter()
    modifies count_Counter;

    requires (count ≥ 0);
    ensures (count ≥ 0);
{
    count_Counter := count_Counter + 1;
}

procedure count_Counter()
    returns (result: int)

    requires (count ≥ 0);
    ensures (count ≥ 0);
{
    result := count;
}
```

Listing 4.14: Boogie translation of contract invariant

Our translation scheme takes the invariant in the contract declaration and splits it into pre and post-conditions. As illustrated in listing 4.14, each procedure has the contract invariant as a pre and post-condition. This ensures that at the end of each function invocation, the contract invariant must hold. You may notice that the only procedure which does not have the contract invariant as a pre-condition is Counter's *init* function. We make no assumptions about the state of the contract before it has been initialised, therefore we only require that after the contract has been initialised it's invariant holds.

Struct invariants

We also introduce struct invariants, which are very similar to contract invariants. We define the semantics of struct invariants to be such that the struct invariant must hold over every struct instance. This requires that every Boogie procedure has knowledge of the struct invariants, therefore struct invariants must be held by all contract and struct procedures. The struct invariant is also split into pre and post-conditions, as shown in listing 4.16.

```
struct A {
    var tens: Int = 10

    invariant (tens mod 10 == 0)

    func add(i: Int)
        mutates (tens)
    {
        tens += i
    }
}
```

Listing 4.15: Flint struct invariant example

```
var tens_A: [int]int;
var nextInstance_A: int;

procedure add_A(structInstance: int, i: int)
    modifies tens_A;
    requires(∀ i: int • i < nextInstance_A ⟹ tens_A[i] mod 10 = 0);
    ensures (∀ i: int • i < nextInstance_A ⟹ tens_A[i] mod 10 = 0);
{
    tens_A[structInstance] := tens_A[structInstance] + i;
}
```

Listing 4.16: Boogie struct invariant

On a final note for struct invariants, in a very similar fashion to external calls, if a struct calls another struct's method it's struct invariant must hold. This is required because the callee may have a reference to the caller's instance, which it could call a method on. If this is the case, the callee will assume that the caller's struct invariants hold, therefore the caller must ensure that this is the case.

4.1.4 Remarks

We introduced function pre and post-conditions and invariants, for contracts and structs. We also introduced some specification predicates, which allow contract authors to specify behaviours such as the return values of their functions. These types of predicates allow for expressive specifications, which allow contract authors to specify the complete behaviour of their contracts.

In appendix B, we provide example Flint contracts which demonstrate the functional specification syntax we introduced to Flint.

4.2 Array Out-Of-Bounds Detection

Using size shadow variables, we can detect out-of-bounds accesses of arrays. This is done by inserting an assertion before the array access, checking that the index specified is less than the size of the array. This forces the verifier to determine whether the attempted index would cause an out-of-bounds array access.

```
let is: Int = [0, 1]
let i: Int = is[2] // Out-of-bounds
```

```
is[0] := 0;
is[1] := 1;
size_is := 2;

// Test index
assert (0 ≤ 2 ∧ 2 < size_is);
i := is[2];
```

4.3 Divide By Zero Detection

We are able to use the translated contract and Boogie verifier to determine if the denominator of a division can be zero. This is achievable by asserting that the Boogie translation of the denominator is not zero. If the assertion verifies, the verifier can prove that the denominator cannot be zero. The code listings below, give an example:

```
.. previous statements ..

percent = agreed / total
```

```
.. previous statements ..

// Test denominator
assert (total ≠ 0);
percent: = agreed div total;
```

4.4 Inconsistent assumptions

When it becomes possible to specify invariants and pre-conditions, inconsistent assumptions also become possible. Inconsistent assumptions occur when the combination of a Boogie procedure's pre-conditions are equivalent to *false*. This allows for the trivial verification of the procedure's implementation. We give an example below.

```
procedure inconsistent(i: int)
    // Conflicting pre-conditions
    requires (i > 0);
    requires (i < 0);
{
    assert (false); // Verifies
}
```

Listing 4.17: Example Boogie procedure with inconsistent assumptions

We argue that inconsistent assumptions are typically unintentional; caused by invariants and pre-conditions unexpectedly conflicting. We propose a scheme to detect Flint functions with inconsistent assumptions to warn the contract author.

Inconsistent assumptions can be detected if a procedure's body is replaced with `assert (false);` and the procedure successfully verifies. We use this technique and apply it to the result of our translated Flint functions, to detect which resultant Boogie procedures have inconsistent assumptions. We take our translated Flint functions and replace their procedure implementations with `assert (false);`. For every successfully verified procedure, we can determine the corresponding Flint function which has inconsistent assumptions.

```
contract Counter {
  var count: Int

  invariant (count >= 0)
}

Counter :: (any){
  public func increment()
    mutates (count)

    pre (count < 0)
  {
    self.count += 1
  }
}
```

Listing 4.18: Flint inconsistent assumptions example

```
var count_Counter: int;

procedure init_Counter()
  ensures (count ≥ 0);
{
  assert (false);
}

procedure increment_Counter()
  modifies count_Counter;

  // Inconsistent
  requires (count < 0);
  requires (count ≥ 0);
  ensures (count ≥ 0);
{
  // succeeds -> inconsistent assumptions
  assert (false);
}
```

Listing 4.19: Boogie translation to detect inconsistent assumptions

4.5 Unreachable Code

We are able to use the translated contract and Boogie verifier to determine if code paths can never happen. In particular, we are able to detect when if-statement conditions are verifiably true or false. This is achievable by asserting that the if-condition does, or does not, hold. If one of the assertions verifies, the verifier can prove that one of the if-statement cases cannot happen. This approach requires the verifier be run multiple times, as it is not possible to simultaneously ask the verifier to prove whether a condition is true or false, without it trivially verifying. The examples below show how this can be done.

```
.. previous statements ..

if (a > 10) {
  .. do something ..
} else {
  .. do something ..
}
```

```
.. previous statements ..

// Test condition true
assert (a > 10);
```

```
.. previous statements ..

// Test condition false
assert (¬(a > 10));
```

Typically, having unreachable blocks of code is an unintentional side effect, therefore it is useful to detect and report these occurrences to the user. This scheme can be used to detect, not only code blocks which the if-condition never satisfies, but also code which the if-condition always satisfies.

4.6 Asset accounting

Smart contracts receiving and sending money generally have to track how much money they have, for example *Bank.flint* (see appendix D.1). Bank.flint is a smart contract which allows users to deposit and withdraw money. In this case, the smart contract must track how much money each user has deposited into their account, so that the user can withdraw, at most, the amount of money that they put in. It is the responsibility of the smart contract author to make sure that this is implemented correctly, to prevent user's losing money.

If there is a bug in how the smart contract tracks value in user's accounts, it could lead to a situation where the balance of the smart contract is different to the sum of its user accounts. This would be a disparity between the actual amount of money held in the contract, and the amount of money represented by the fields and logic of the smart contract. We argue that it is dangerous if a smart contract is able to achieve this, as it means that the smart contract can 'think' it has more or less money that it actually has. In the rest of this section: we describe in further detail how this can happen; explore an approach which would verify that the contract never reaches this state; and conclude by updating Flint's Wei semantics to reduce the number of ways that a contract can enter this state.

```
public func clear(account: Address)
  mutates (balances, Wei.rawValue)
{
  balances[account] = Wei(0) // This doesn't change the contract's balance
}
```

Listing 4.20: Example of a dangerous function

4.6.1 Wei

Flint's Wei struct encourages the safe manipulation of Wei in a smart contract. For example, the struct enables the atomic transfer of Wei; and makes all the uses of Wei explicit. However, it is still possible for the balance of the contract to fall out-of-sync with the Wei represented in it.

Constructor

When calling the Wei constructor, you can specify an amount (greater than zero) of Wei you would like to create. Therefore, the amount of Wei represented in the contract, can be trivially greater than the balance of the contract, we simply initialise a Wei instance with as much Wei as we'd like.

```
// Succeeds even if the contract balance doesn't have 1000 Wei
let w: Wei = Wei(1000)
```

This is a dangerous pattern, therefore we propose requiring that the Wei struct must always be initialised with 0 Wei. This approach prevents the amount of Wei represented in the contract, being greater than the balance of the contract. To introduce Wei into the smart contract, the constructor does not need to be called with a value greater than zero, as `@payable` functions with `implicit` Wei parameters perform this function.

```
@payable
public func receive(implicit w: Wei) { // Increase balance amount
    balance.transfer(&w)
}
```

Assignment

It is also possible for the amount of Wei represented in the contract to be less than the balance of the contract. This is possible because struct instances can be assigned between variables. Therefore, we can reassign any Wei variable with a fresh Wei instance containing 0 Wei. This reduces the amount of Wei represented in the contract.

```
Bank :: (manager) {
    public func clear(account: Address)
    {
        // Wei previously in user's account is lost
        deposits[account] = Wei(0)
    }
}
```

Listing 4.21: Destroying Wei - assignment

This is a dangerous pattern, the overridden Wei struct cannot be recovered after the assignment. Instead, to implement this functionality, the Wei should be transferred to another variable to keep track of the balance of the contract.

Local variable

To avoid the pattern above, we might be tempted to transfer the Wei to a local variable, however this could lead to another dangerous pattern. Wei contained within local variables is lost when the function returns.

```
Bank :: (manager) {
    public func clear(account: Address)
    {
        let value: Wei = Wei(0)
        value.transfer(&deposits[account]) // Empty account

        // Wei in ‘value’ is lost
    }
}
```

Listing 4.22: Destroying Wei - function completion

This requires that we must transfer all Wei into global variables, by the end of the function. Listing 4.23 shows a safe use of Wei.

```
contract Bank {
    let deposits: [Address: Wei] = [:]
    let manager: Address

    let managerBonus: Wei
}

Bank :: (manager) {
    public func clear(account: Address)
    {
        let value: Wei = Wei(0)
        value.transfer(&deposits[account]) // Empty account

        managerBonus.transfer(&value) // Store Wei in global variable
    }
}
```

Listing 4.23: Safe use of Wei

4.6.2 Verifying safety

Here we explore whether the verifier can be used to detect and prevent Wei being destroyed. The problem we are trying to solve, is when the balance of the smart contract is not longer equal to the amount of Wei represented within the contract. This can be expressed as the following specification:

$$balance = representedValue \text{ where } balance = receivedWei - sentWei$$

Therefore the specification we propose verifying contracts, is:

$$representedValue = receivedWei - sentWei$$

If we can verify that a smart contract satisfies this specification, then we can guarantee that the contract accounts for all the Wei contained within its balance. To do this, we propose the following scheme.

Shadow variables

To perform this verification, the verifier must track the Wei which is sent, received and represented in the contract. Therefore, we propose the introduction of three shadow variables: `totalWei`, `sentWei` and `receivedWei`. We propose using these shadow variables to check that the invariant `totalWei = receivedWei - sentWei` holds for every Flint function. Listing 4.24, gives an example.

```
var totalWei: int;
var sentWei: int;
var receivedWei: int;

procedure doSomething()
    requires (totalWei == receivedWei - sentWei);
    ensures (totalWei == receivedWei - sentWei);
{
    //... do something ...
}
```

Listing 4.24: Asset accounting shadow variables example

The remainder of this subsection outlines our verification proposal, which tracks the change in the amount of Wei received, sent and represented in the contract.

Wei Constructor

As discussed in section 4.6, the Wei struct can be initialised with arbitrary Wei, `let w: Wei = Wei(n)`. In this case, we propose increasing the value of `totalWei`, by `n`. This indicates that the amount of Wei represented in the contract has increased.

```
let w: Wei = Wei(n)
```

```
//let w: Wei = Wei(n)
totalWei := totalWei + n;
```

Wei Assignment

When a Wei struct is assigned to another Wei variable, the old Wei is lost. Therefore the `totalWei` variable must be decremented by the amount of Wei which was in the variable, before it is overwritten.

```
let w1: Wei = Wei(n)
let w2: Wei = Wei(m)

w1 = w2
```

```
// let w1: Wei = Wei(n)
totalWei := totalWei + n;
// let w2: Wei = Wei(m)
totalWei := totalWei + m;

// w1 = w2
totalWei := totalWei - rawValue[w1];
```

Wei Transfer

Wei can be transferred from one Wei instance to another. If the transfer occurs between two local variables or two global variables, the transfer has no affect on the amount of Wei represented in the contract. However, if the Wei is transferred from a local variable to a global variable, the amount of Wei represented in the contract increases. Conversely, if the Wei is transferred from a global variable to a local variable, the amount of Wei represented in the contract decreases. Therefore, we propose incrementing or decrementing `totalWei` if Wei is transferred between local and global variables.

```
let w: Wei = Wei(0)

w.transfer(&balance)

balance.transfer(&w)
```

```
// let w: Wei = Wei(0)
totalWei := totalWei + 0;

// w.transfer(&balance)
totalWei := totalWei - rawValue[balance];

// balance.transfer(&w)
totalWei := totalWei + rawValue[w];
```

Sent Wei

The `send` global function allows Wei to be sent to other Ethereum addresses. In this case, the amount of Wei that the contract has sent must be incremented. Therefore we propose incrementing `sentWei`, by the amount of Wei sent.

```
let w: Wei = Wei(0)

send(0x1, &w)
```

```
// let w: Wei = Wei(0)
totalWei := totalWei + 0;

// send(0x1, &w)
sentWei := sentWei + rawValue[w];
```

Received Wei

Wei can be received by the contract with public `@payable` functions. In this case, the amount of Wei that the contract has received must be incremented. Therefore we propose incrementing `receivedWei`, by the amount of Wei received.

```
@payable
public func receive(implicit w: Wei)
{
    //... do stuff
}
```

```
procedure receive(w_value: int)
{
    var w: int;
    call w := init_Wei(w_value);

    // implicit w: Wei
    receivedWei := receivedWei + w_value;

    //... do stuff
}
```

4.6.3 Updating Flint's semantics

In the previous subsection, we explored a method which uses the verifier to prove that all the Wei in the contract balance is accounted for. However this approach has limitations, including that it allows for dangerous patterns. In particular, users can assign Wei to other Wei type variables, overwriting their value. We argue that this pattern should not be possible in Flint, and explore how we can prevent it in the first place, by updating the semantics of Flint. Listing 4.25 gives an example of a dangerous Wei pattern.

```
var w: Wei = Wei(0)
// should throw error -> potentially overwriting Wei
w = Wei(0)
```

Listing 4.25: Dangerous assignment of Wei

We propose not allowing Wei to be assigned to a variable, more than once. In particular, we propose requiring that variables of type Wei must be declared using the `let` construct, to prevent further assignment to the variable. Similarly, we propose that dictionaries and arrays which contains Wei as elements, must be declared using the `let` construct.

```
var w1: Wei = Wei(0) // Not allowed
let w2: Wei = Wei(0) // Allowed

let ws: [Wei] = [Wei(0)]
ws[0] = Wei(0) // Not allowed
ws = [Wei(0)] // Not allowed
ws[0].transfer(source: &Wei(0)) // Allowed
```

Listing 4.26: Proposed updates to Flint's semantics

Furthermore, we must make sure that any Wei received via a `payable` function, or extracted from global variables, is either sent out of the contract or placed back into the contract's global state. Otherwise, the Wei represented in the contract's global state may not reflect the true amount of Wei held in the contract. We propose requiring that every locally declared Wei, or Wei parameter, must execute the `transfer` method or global `send` function at least once. We can see how this requires that any Wei held in local variables, must be transferred back into global variables.

```
let deposits: Wei = Wei(0)

@payable
func deposit(implicit w: Wei) {
    let temp: Wei = Wei(0)
    temp.transfer(source: &w) // W must be transferred
    deposits.transfer(source: &temp) // temp must be transferred

    // All Wei held in local variables has been transferred to global variables
}
```

Listing 4.27: All Wei must be transferred or sent

However, this approach is limited only in how we handle partial transfers. If a contract author wishes to only send half of the Wei contained within a local variable, this approach considers all of the Wei as moved and wouldn't raise a warning. When in fact, half of the Wei is still contained in the variable, which would lead to the represented value and true value of the contract falling out of sync. We posture that this problem might be solved through formal verification, perhaps by proving that all of the Wei in a local variable has been transferred into global variables or out of the contract. We see how modelling the transfer of Wei through a smart contract is complex, and at this point we leave this as an open problem.

4.7 Holistic Specifications

As discussed in section 2.6.2, holistic specifications are concerned with describing the overall behaviour of software components. Holistic and functional specifications together create an expressive language, which can be used to check and understand the behaviour of Flint smart contracts. In this section, we describe the semantics of our holistic operators, their implementation, and how we use symbolic execution to verify Flint smart contracts against them.

4.7.1 Semantics

We introduce the holistic operator `will` to Flint's specification language. `Will(A)`, for the Flint expression A, asks if all execution paths from the initial state contains a state where A is true.

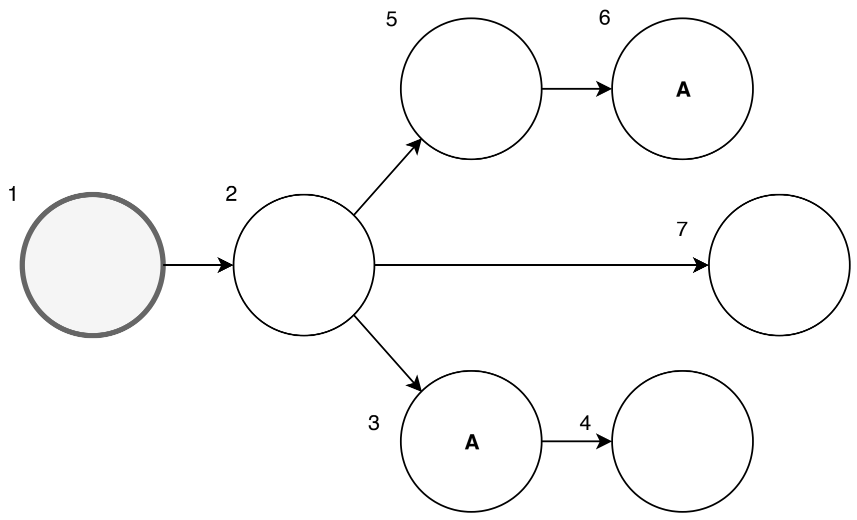


Figure 4.1: Example smart contract states and transitions

Fig 4.1 gives an example of a state diagram of a theoretical smart contract. State 1 is the initial state of the contract. Each arrow represents a transition from one state to another, which is caused by a transaction calling a mutating function on a smart contract. The logical proposition **A** holds, in states 3 and 6, and the possible execution paths for this state space are as follows: $1 \rightarrow 2 \rightarrow 5 \rightarrow 6$; $1 \rightarrow 2 \rightarrow 7$; and $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. Henceforth referred to as paths a, b and c respectively.

For the state diagram above, `Will(A)` does not hold as there exists a path, namely path b, which does not contain a state where **A** holds. Our semantics require that **A** must be true at some point, for all execution paths.

The semantics of the `will` operator we propose are inspired by the semantics proposed by S. Drossopoulou et al [32]. The difference between the two is whether we require the property to be true for all execution paths, or for at least one. The *will* property proposed by S. Drossopoulou et al requires that **A** must hold at some point, for at least one execution path. Whereas our semantics require that it must hold, at some point, for all execution paths. We discuss further the strengths and limitations of the semantics we propose in the evaluation section 6.3.

4.7.2 Syntax

Here we outline the syntax we introduce to Flint, to express the holistic specifications, we describe above. Since our holistic operators are describing properties about the state of the contract, we propose the addition of a holistic specification declaration in a contract declaration. For example, listing 4.28 gives an example of how the Will operator is used to specify properties of the contract.

```
contract Hol {
    var i: Int = 0

    will(i >= 5)
}
```

Listing 4.28: Flint contract with holistic specification

Here we specify that the global variable ***i*** will, in every execution path of the smart contract, satisfy $i >= 5$ at some state.

4.7.3 Encoding holistic specifications

We discuss our implementation of the verification system, for holistic specifications, for Flint smart contracts. In order for the specifications to be automatically verified, we translate them into a form which can be consumed by a tool to perform the verification. In this section, we propose our translation of the specification into Boogie and how we use the symbolic execution engine Symbooglix [12] to perform the verification.

Will

Following the discussion of the semantics we propose for the `will` operator 4.7.1, we can see that they cannot be expressed using the Hoare logic primitives that Boogie provides. Holistic specifications describe how state should change over multiple transactions, however functional specifications can only describe state changes during one transaction. Therefore, we propose encoding the semantics of the `will` operator in a Boogie procedure. We then symbolically execute the procedure to verify whether the smart contracts satisfies the specification. Listing 4.29 gives a template for the Boogie procedure code we use to represent `will(A)`, where ***A*** is a Boogie translation of a Flint expression such as $i >= 5$.

```
procedure Will() {
    call init_Contract();
    while(¬A ∧ termination) {
        /* If no possible functions to execute, break out of loop */
        ...
        /* Randomly select a function to execute */
    }
    assert (A);
}
```

Listing 4.29: Boogie example of *Will(**A**)* operator

The procedure first initialises the contract and then searches the state space of the contract until ***A*** is true, or no more states can be reached. The symbolic execution engine, randomly selects a Flint function to call, which allows for each possible state of the program to be explored. This follows the semantics of the *Will* operator, we defined. Each execution path is explored by the symbolic execution engine, which is then checked to see if A holds true at some point.

To implement the random selection of the Flint function to call, we create two Boogie helper procedures: `callable_functions` and `select_function`. `callable_functions` returns an array of the callable functions from the current state. The array contains a unique identifier corresponding to each Flint function. `select_function` receives the array of function identifiers and randomly calls on the functions.

Since a transaction targeting a Flint smart contract can only call public functions we only consider public functions. Furthermore, an optimisation we immediately apply is that we only consider public mutating functions. These are the only callable functions which can change the state of the program, thus reducing the number of transitions that the symbolic execution engine must explore.

To determine which of the public mutating functions can be called, we symbolically test whether each of their pre-conditions have been satisfied. This includes implicit pre-conditions, such as the current caller capabilities and the type state of the contract. Listing 4.30 demonstrates the pattern used to construct the `callable_functions` procedure.

```
procedure callable_functions() returns (functions: int, callable_functions: [int]int)
{
    var count: int;
    var tmp_callable_functions: [int]int;

    count := 0;
    // Testing if function 1's pre-conditions are currently satisfied
    if (FUNC1_PRE_CONDITIONS) {
        tmp_callable_functions[count] = FUNC1_GLOBAL_ID;
        count := count + 1;
    }
    if (FUNC2_PRE_CONDITIONS) {
        tmp_callable_functions[count] = FUNC2_GLOBAL_ID;
        count := count + 1;
    } ...

    functions := count;
    callable_functions := tmp_callable_functions;
    return;
}
```

Listing 4.30: Boogie *callable_functions* procedure

Once all the callable functions have been determined, we must then randomly choose one function to call. `select_function` is called, using the output from `callable_functions`, and it calls one of the functions. We implement this by introducing a '`selector`' local procedure variable, which provides a random index into the list of callable functions. We do this by ***havoc***'ing the variable, which tells the symbolic execution engine to assign a fresh symbolic value to the variable. Listing 4.31 demonstrates the pattern used to construct the `select_function` procedure.

```
procedure select_function(functions: int, callable_functions: [int]int)
{
    var selector_index, selected_function: int;

    // Function arguments + return values
    var arg1, arg2, ... result_value: int;

    havoc selector_index;
    // Make sure selector_index has a valid range
    assume (0 ≤ selector_index ∧ selector_index < functions);
    selected_function = callable_functions[selector_index];

    if (selected_function = FUNC1_GLOBAL_ID) {
        call result_value := proc_1();
    } else if (selected_function = FUNC2_GLOBAL_ID) {
        havoc arg1, arg2 ...;
        call proc_2(arg1, arg2...);
    } ...
}
```

Listing 4.31: Boogie *select_function* procedure

As you can see in listing 4.31, the functions we can call may require arguments. The symbolic executor allows us to call the functions with symbolic values as arguments, therefore we are able call the function with all possible argument values at once.

We combine the two helper functions into a procedure which is the Boogie encoding of our proposed semantics of `will`. Listing 4.32 gives an example of what this looks like. We perform a check, after we call `callable_functions`, to see if any function can be called. If not, we have reached a terminal state and the while loop is exited, otherwise we randomly select a function to call and continue searching the state space of the contract.
However, the implementation described above may not terminate, if the state space of a smart contract contains cycles. Our implementation cannot detect or handle cycles. Therefore, to guarantee termination, we introduce a bound variable which sets a limit on the depth of transactions we can consider. The default value is 5, which means that the holistic verifier will only look 5 transactions deep from the initial state to check the specification. Listing 4.32 presents this implementation.

```
procedure Will_A()
{
    var bound, number_callable_functions: int;
    var callable_functions: [int]int;

    bound := 5;
    call init_Contract();
    while(¬A ∧ bound > 0) {
        call number_callable_functions, callable_functions := callable_functions();
        if (number_callable_functions = 0) {
            break;
        }
        call select_function(number_callable_functions, callable_functions);
        bound := bound - 1;
    }
    assert (A);
}
```

Listing 4.32: Boogie *Will(**A**)* encoding

However, the introduction of a bound, changes the semantics of our Boogie encoding of `will`. The Boogie encoding becomes an approximation of the `will` semantics we would like to have. This is because our implementation cannot verify specifications which require more than 5 transactions. We discuss the semantics of the implementation of our holistic operators in the evaluation, 6.3. We also discuss our implementation of holistic specifications, into the Flint compiler, in section 5.3.7.

4.8 Remarks

We introduced the functional and holistic predicates, which we added to Flint's syntax. We discussed how we can translate these into Boogie, so that Flint's verification system can verify Flint contracts against these user-supplied specifications. We can see how the functional and holistic predicates we add allow for smart contract authors to write expressive specifications capturing the behaviour of the contracts. We also discussed approaches to detecting out-of-bounds violations, or division by zero errors, and others, which can be used to prevent immutability bugs.

In the next chapter, we discuss our implementation of Flint's verification system, where we combine all our proposed changes and verification theory. Specifically, we outline how we implement functional and holistic specification verification discussed in this section, as well as the other immutability bug prevention features discussed here.

Chapter 5

Flint Verification Implementation

In this chapter, we outline our implementation of the verification features discussed in previous chapters. In particular, we discuss how we implement translation from Flint to Boogie, and the design considerations we have to consider to do this correctly. We also discuss how we use Boogie for the verification of functional specifications; as well as detecting inconsistent assumptions; array-out-of-bounds errors; and as well as verifying holistic specifications; and returning meaningful verification error messages back to the user.

5.1 Verification Architecture

Here we briefly mention the overall architecture of the Flint compiler, to give context to how the verifier fits in the overall system. Figure 5.1, gives a high level diagram of the components of the Flint compiler. The orange boxes are components which we modified and the green box is the verifier component we introduced to the architecture. Each box contains the approximate number of lines we introduced to each component.

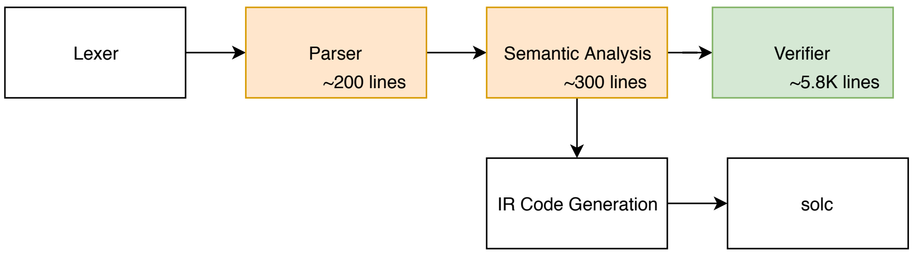


Figure 5.1: High level overview of the Flint compiler

We modified the parser to extend Flint's syntax to include the verification constructs we describe in chapter 4. Figure 5.2 illustrates the components which together form the semantic analysis of the Flint compiler. We introduced the call graph generator component, the output of which is consumed by the Boogie IR resolver. This is discussed further in section 5.2.4.

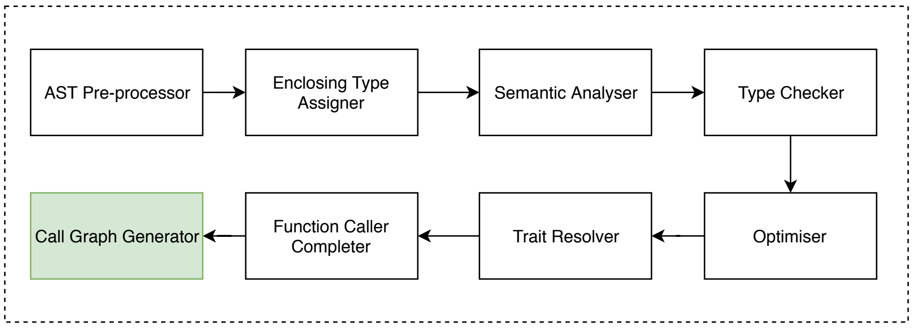


Figure 5.2: Overview of the semantic analysis stage of the Flint compiler

5.2 Design Considerations

In this section we discuss the design considerations required to successfully translate Flint constructs to Boogie. These are based on our observations of the Boogie syntax, and our proposed translation scheme.

5.2.1 Global variable assign and declare

In Flint it is possible to assign and declare global variables, in one line. Global variables in Boogie can only be assigned values, within a procedure. As discussed in section 3.6.1, we must take each global variable assignment and place it within the contract's translated *init* function.

5.2.2 Function variable declarations

Procedure variables must be declared at the beginning of the procedure in Boogie, however they can be declared at any point in a Flint function.

```
func localVariable() {
    let a: Int = 10
    let b: Int = a + 10
    a += 1
    let c: Int = a + b
}
```

Listing 5.1: Flint variable declarations

```
procedure localVariable()
{
    var a, b, c: int;

    a := 10;
    b := a + 10;
    a := a + 1;
    c := a + b;
}
```

Listing 5.2: Boogie variable declarations

Therefore, during our translation implementation we must make sure to collect all the local variable declarations in a Flint function and place them at the beginning of the function's generated procedure.

5.2.3 Expressions and Statements

As discussed in section 3.11, Flint expressions may be translated into a pair of Boogie expressions and statements. As shown in listings 3.60 and 3.59. Flint assignments and function calls must be done as separate Boogie statements, our translation implementation must support this.

5.2.4 Modifies clause

As discussed in section 3.5.3, anytime a global variable is modified, the enclosing procedure must have a corresponding `modifies` clause in it's procedure declaration. It is also possible to modify 'hidden' global state in Flint, such as the contract's type state, or a struct's `nextInstance` global variable. They are hidden in that, their values cannot be explicitly accessed in Flint. Any Boogie procedure modifying the contract's type state, or instantiating a struct, must have the corresponding `modifies` clause. As these changes to Boogie's global variables are implicit, we look to infer the `modifies` keyword, for Flint functions which implicitly modify Boogie global state

As discussed in section 3.5.3, we introduced the `mutates` keyword to Flint's function declaration syntax. This makes explicit the global variables that the Flint function modifies. This is a pattern which increases visibility into the behaviour of the function, at the function declaration level. It also provides information that the Boogie translator can use, to calculate the necessary Boogie `modifies` clause for the generated Boogie procedures.

To infer the functions which modify the 'hidden' global variables, we generate a call graph of the Flint code. We then annotate each Flint function with the 'hidden' global state they modify. Finally we propagate this information to each calling node in the call graph. We do this n times, where n is the number of functions declared in the contract, this ensures that the information propagates completely to each node. We give an example of how the 'hidden' modified global state is propagated, in listing 5.3 below.

```
C :: (any) {
    func a()
    {
        let w: Wei = Wei(0) // Modifies nextInstance_Wei variable
    }

    func b()
    {
        a() // Therefore, this function modifies nextInstance_Wei variable
    }

    func c()
    {
        b() // Therefore, this function modifies nextInstance_Wei variable
    }
}
```

Listing 5.3: Example Flint contract with modifies inference

Traits

The Flint compiler 'resolves' traits, by removing the trait declaration and adding the trait-defined methods to it's conforming struct or contract types. Since the verification stage occurs after the trait resolution, traits are transparent to the Boogie translator which can translate the added function declarations, as it would any other.

However, we need to consider the `modifies` clause of the generated Boogie code, of the trait-defined function. Flint's syntax does not allow trait-defined functions to express `mutates` clauses. This is because traits themselves, do not contain any state.

```
struct trait Asset {

  // Moves 'amount' from 'source' into 'this' asset.
  func transfer(source: inout Self, amount: Int)
  {
    if source.getRawValue() < amount {
      fatalError()
    }

    // setRawValue, in the Wei struct, modifies state!
    let _: Int = source.setRawValue(value: source.getRawValue() - amount)
    let _: Int = setRawValue(value: source.getRawValue + amount)
  }

}
```

Listing 5.4: Extract from Asset trait [1]

However, their conforming types do, and they can call functions which modify global state. But since trait-defined functions do not have a `mutates` clause, the generated procedure will not include a `modifies` clause, which may be required for any trait-defined functions which call other functions which modify state. An example of this is shown, in listings 5.4 and 5.5

```
struct Wei: Asset {
  var rawValue: Int = 0

  func setRawValue(value: Int) -> Int
    // Modifies state
    mutates (rawValue)
    post(self.rawValue == value)
  {
    rawValue = value
    return rawValue
  }
}
```

Listing 5.5: Extract from Wei struct [2]

[1]Taken from https://github.com/YianniG/flint/blob/master/stdlib/Asset.flint, accessed June 16, 2019

For all trait-defined functions, we must find the functions they call and determine the state that these functions modify. These should then be added to the modifies clause of the trait-defined function. This can be done using a similar method to the one described in section 5.2.4, to infer the `modifies` clause of 'hidden' global state.

5.3 Implementation

In this section, we discuss the process we implemented the process to translate Flint smart contracts, and verify their specifications. We outline how we split the verification process into discrete stages: translation, resolution and verification. This approach made the verification system extensible, by spreading the complexity of the verification across multiple components. The Flint translator (see section 5.3.1 below) translates the Flint constructs to Boogie. which are consumed by all of the other verification features that the Flint compiler supports. Figure 5.3 outlines the structure of the Flint verification stage we introduce to the Flint compiler.

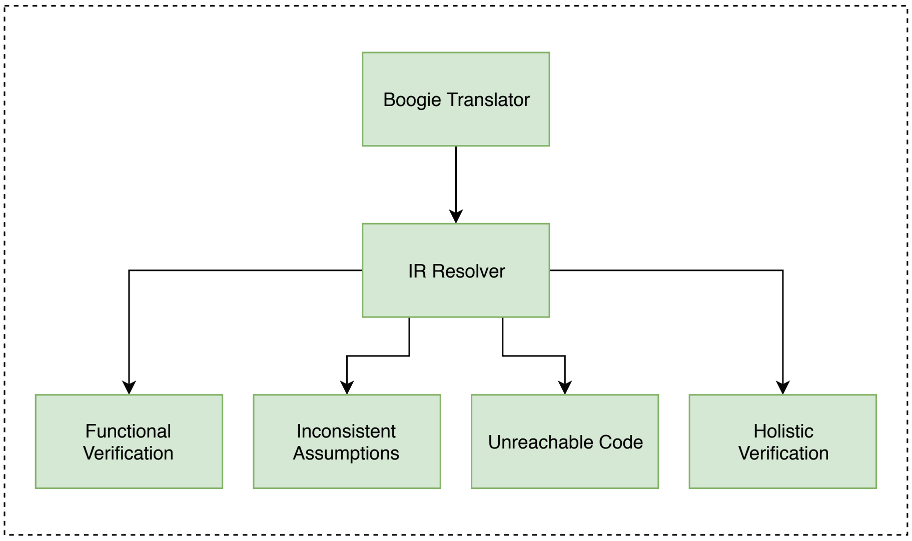


Figure 5.3: Overview of verification stage the of the Flint compiler

Following the design considerations discussed in section 5.2, we introduce a Boogie intermediate representation (IR). This provides an intermediate step, which is a partial translation from Flint to Boogie, and allows us to keep some of the Flint constructs before they are completely removed in the translation to Boogie. The IR allows us to infer modified global state for procedure modifies clauses; collect procedure local variable declarations; and include declarations and assignments of global variables in the contract's init procedure. In subsection 5.3.1, we discuss the Boogie IR in further detail; how it is generated, and how it is used.

Once the Boogie IR AST has been generated, it is consumed by further downstream verification processes. In the following subsections, we discuss how each of the verification functionalities offered by the Flint compiler consume the Boogie IR AST to produce functional; holistic; reachability and inconsistency verifications.

5.3.1 Translation

To translate the Flint compiler, we recursively visit each node in the AST, returning the corresponding translated AST node. The Boogie AST node produced is not the final Boogie AST node, which will be rendered to form the output file to be verified. It is instead an intermediate representation node, which must be translated again into the final Boogie AST node. We discuss this later on in this subsection.

[2]Taken from `https://github.com/YianniG/flint/blob/master/stdlib/Asset.flint`, accessed June 16, 2019

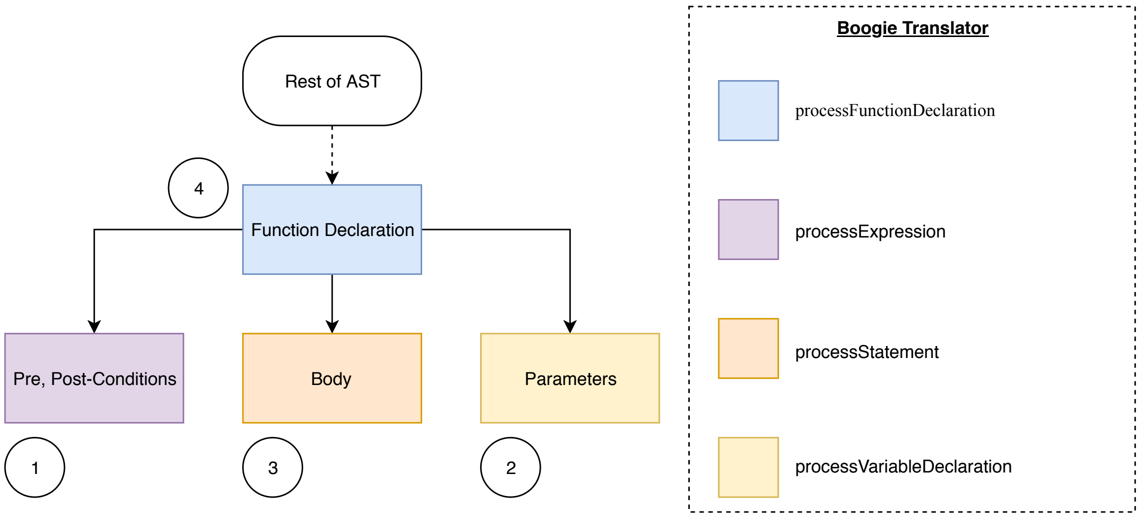


Figure 5.4: Extract of how the Boogie translator processes each Flint function node

The AST visitor uses a post-order to explore each node. Figure 5.4, numbers each Flint node in the order that it is visited. Each node first visits it's children and then produces the corresponding Boogie AST node for that construct. For each node, the Boogie translator uses the translation scheme described in chapter 3 to translate the Flint language construct. We give an example of how a Flint function declaration is translated into our Boogie IR below, fig 5.5.

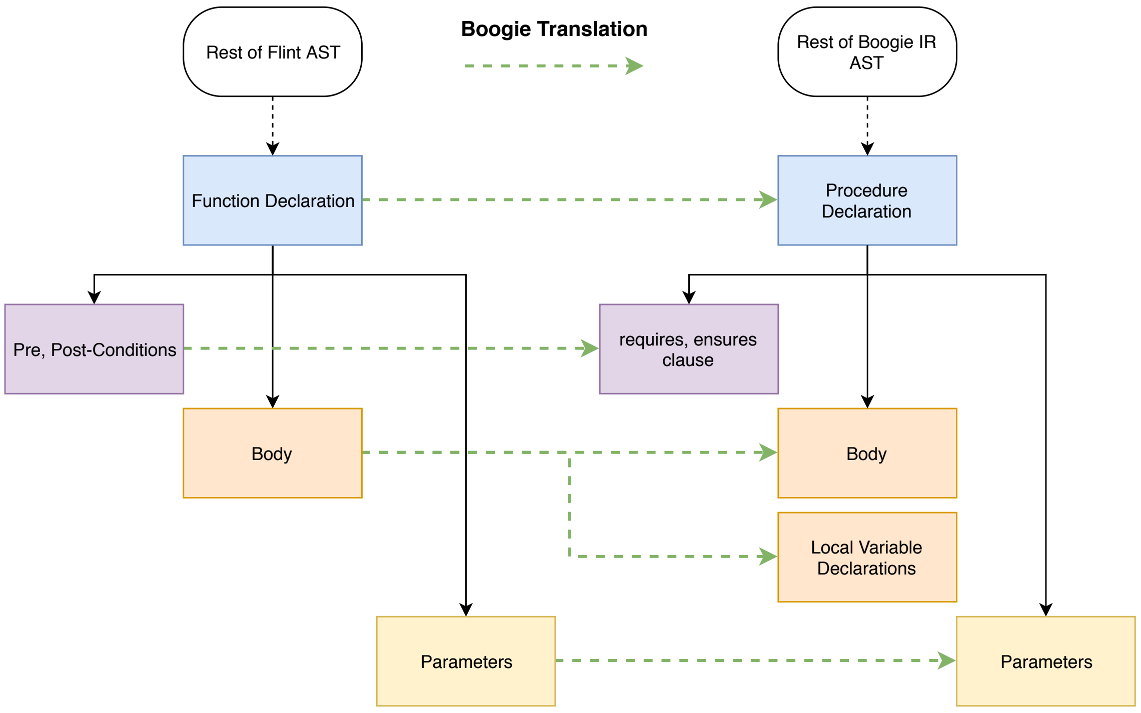


Figure 5.5: Extract of how the Boogie translator translates Flint function Flint AST node

During the translation of the Flint AST, the translator maintains state about the smart contract. For example, the names of declared variables within the current function scope are tracked, so the translator can discern whether an identifier is referring to a parameter, local or global variable. Other information which is maintained includes the name of the current top level declaration; the defined state variables; defined enums; the current scope context and similar.

Boogie AST and Intermediate Representation

In order to translate the Flint contract, we need a Boogie AST to convert the Flint AST into. Using the Boogie grammar introduced in section 2.8, we define the Boogie AST nodes. Each Boogie AST node is a struct, defined in `Verifier/Boogie/BoogieAST.swift`[3], with properties which can be used to access its child nodes.

In order to perform the `modifies` clause inference, discussed in section 5.2.4, the Boogie translation returns a Boogie IR AST node. The IR nodes allow us to process the AST a final time, keeping semantic information from the source Flint code, before the generating the final Boogie node. The Boogie IR nodes are defined in `Verifier/Boogie/BoogieTranslationIR.swift`[4].

For example, in the IR, the modified declarations are annotated with the property `userDefined`. This tells the compiler whether the variable was defined by the user or if it was created by the compiler, such as the *nextInstance* global variable. If it was created by the compiler, it is the compiler's responsibility to infer the modifies clauses for that parameter.

Once the IR resolution has occurred, the IR is translated into the Boogie AST, which is rendered to form the file that the Boogie verifier checks. We discuss both these stages, in more detail, below.

Intermediate Representation Resolver

The IR resolver converts the Boogie IR AST into the Boogie AST, so it can be rendered and verified by Boogie. Only some of the AST nodes have a different Boogie IR AST node than the Boogie AST, therefore the IR resolver only needs to translate these nodes:

- **BIRProcedureDeclaration**. Represents a procedure declaration, in an unresolved state. Does not contain inferred modifies clauses, contains BIRInvariant and maintains the cause of the declaration (struct/contract `init` or function declaration).
- **BIRInvariant**. This node maintains the semantic information that the given property is an invariant. This is used by the IR resolver to convert this node into Boogie pre and post-conditions.
- **BIRModifiesDeclaration**. Tracks whether the modified global variable is 'hidden' state, or user-defined.
- **BIRTopLevelDeclaration**. Contains BIRProcedureDeclaration nodes.
- **BIRTopLevelProgram**. Contains BIRTopLevelDeclaration nodes.

The other nodes in the IR AST, use the Boogie AST nodes. To translate the whole IR AST, we only need to process the nodes listed above. Figure 5.6 gives an example of how the Boogie IR nodes are translated to the Boogie AST nodes.

Preamble and Postamble

As we discussed in our design considerations 5.2.3, translating Flint expressions can produce Boogie expressions and statements. We enable this behaviour by using a preamble and postamble mechanism. Translating a Flint expression returns a triplet of `([BStatement], BExpression, [BStatement])`: the first element contains a list of statements as a preamble; the final element contains a list of statements as a postamble; and the middle element is the translation of the Flint expression. There is a requirement on the calling function, that the preamble statements must be placed before the statement containing the expression, and the postamble must be placed after the statement containing the expression. Listing 5.7 gives an example of how the preamble is used to translate a nested Flint expression.

[3]`https://github.com/YianniG/flint/blob/master/Sources/Verifier/Boogie/BoogieAST.swift`
[4]`https://github.com/YianniG/flint/blob/master/Sources/Verifier/Boogie/BoogieTranslationIR.swift`

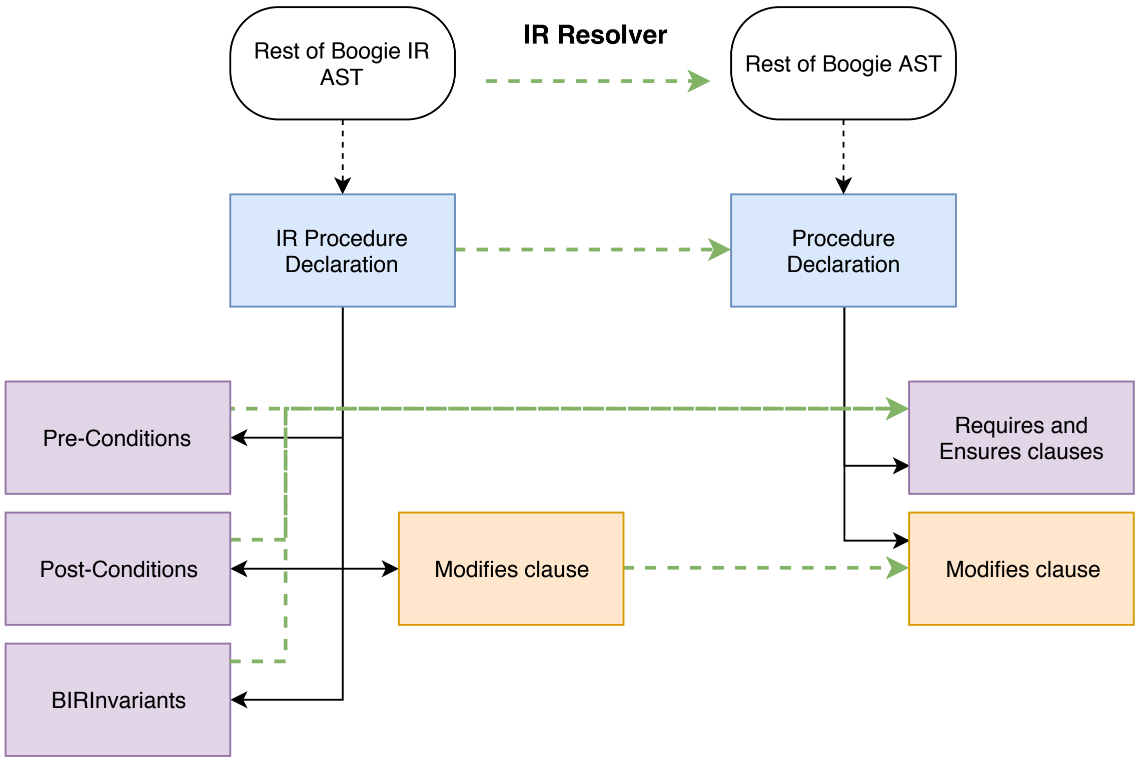


Figure 5.6: Extract of how the Boogie IR resolver translates the IR AST to the Boogie AST

```
let i: Int = 10
let j: Int = 5

i = (i += 1) + (j  /= 2) + 3
```

Listing 5.6: Flint code with nested assignments

```
var i: int;
var j: int;

i := i + 1;      // preamble
j := j div 2;    // preamble
i := i + j + 3; // Translated Expression
```

Listing 5.7: Example of expression preamble

5.3.2 Rendering the Boogie code

Once the Boogie AST has been generated, we must render the Boogie code, to be consumed by the Boogie verifier. We do this by recursively visiting each node in the AST, producing it's string representation. Each Boogie AST node has a `render` function, which returns the string representation of that node. If the AST node has children, when the parent's `render` function is called, it calls it child's node `render` function. Each node combines its children's string representations and combines them to form the overall representation of the Boogie program. Listing 5.8, gives an example of how the Boogie While statement is translated.

```
struct BWhileStatement {
  let condition: BExpression
  let body: [BStatement]
  let invariants: [BLoopInvariant]
  let ti: TranslationInformation

  func render() -> (String, SourceMapping) {
    let (invariantString, invariantMapping) = invariants
      .map({ $0.render() }).reduce(("", Set<TranslationInformation>()),
                                   { x, y  in
                                      return (x.0 + "\n" + y.0, x.1.union(y.1))
                                    })
    let (bodyString, bodyMapping) = body
      .map({ $0.render() }).reduce(("", Set<TranslationInformation>()),
                                   { x, y in
                                      return ("\(x.0)\n\(y.0)", x.1.union(y.1))
                                    })
    return("""
        \(ti)
        while(\(condition))
        // Loop invariants
        \(invariantString)
        {
          \(bodyString)
        }
    """, invariantMapping.union(bodyMapping).union(Set([ti])))
  }
}
```

Listing 5.8: Boogie While struct definition which renders Boogie code

Generating translation information

Once the Boogie code has been rendered, we generate an in-memory mapping from Boogie line numbers to the source smart contract line numbers. We do this by searching the generated Boogie string for 'markers' of the form: `// #MARKER# 313938334`, which indicates the source Flint line that generated the Boogie line. This is required as the errors returned by the Boogie verifier, are with respect to the line numbers of the Boogie code it verified. Therefore, we must generate this mapping in order to determine the original offending Flint line. We discuss how this is used, to generate verification error messages, further in section 5.4. The `SourceMapping` result type returned by `render`, in listing 5.8, gives the source information which is used to create the mapping from Boogie line number to the original Flint source.

5.3.3 Boogie Verification

As part of the Flint installation, we include Boogie. Once the Flint translation has been generated, the resulting Boogie code is placed in a file so the Boogie verifier can verify it. To do this, the compiler generates a temporary file in the OS's defined temporary working directory, and is passed to the Boogie verifier for verification. We execute the Boogie verifier by calling the Boogie executable with the following arguments:

```
mono boogie/Binaries/Boogie.exe [path/to/tmp/boogie.bpl file] /inline:spec
```

Using the method described in chapter 3, we translate the Flint smart contract into its Boogie representation and execute the Boogie verifier. This verifies the smart contract's functional specification, as well as detecting array out-of-bounds accesses. We discuss how we parse the output generated by the Boogie verifier later, in section 5.4.1.

Procedure inlining

Inlining procedures is a technique which takes the body of a called procedure, and copies it into the call location of any calling functions. In the command given above, we use execute the Boogie verifier and pass the flag `/inline:spec`. This flag tells the Boogie verifier to inline any annotated procedure calls. We give an example of the effects of procedure inlining below.

```
var i: int;

procedure {:inline 10} A()
    modifies i;
{
    i := i + 1;
}

procedure {:inline 10} B()
    modifies i:
{
    i := 0;
    call A();
    assert (i = 1); // Fails
}
```

```
var i: int;

procedure A()
    modifies i;
{
    i := i + 1;
}

procedure B()
    modifies i:
{
    i := 0;

    // inlined A:
    i := i + 1;

    assert (i = 1); // Verifies
}
```

As we can see, inlining the procedure call gives the verifier more information about the effect of a called procedure. This allows the verifier to specify properties about outcomes of procedure calls, when pre and post-conditions of the procedure are not used to specify the behaviour of the procedure. The example above gives an example of a property that the verifier could verify after applying procedure inlining.

The procedure annotation `{:inline 10}` tells the compiler to inline the procedure to a maximum depth of 10 times. When a procedure is recursive or there are deep chains of procedure calls, a procedure could be inlined an unbounded infeasibly large number of times. Therefore, the verifier requires a maximum bound to apply the inlining, after which it keeps the procedure call was before.

The procedure inlining technique can increase the proving power of the verification system, with the extra function information, without requiring a more complete specification of the program. We evaluate the use of inlining functions, on proving power of Flint's verification system, in section 6.2.

5.3.4 Functional Specifications

To verify a Flint smart contract's functional specification we render the Boogie AST, generated by the Boogie translator discussed above (see section 5.3.1). Our translation scheme translates the Flint functional specifications into the Boogie AST, therefore we only need to render and execute the Boogie verifier on it. Once Boogie has finished verification, we parse it's output, using the approach described below 5.4.1, and return the verification outcome to the user.

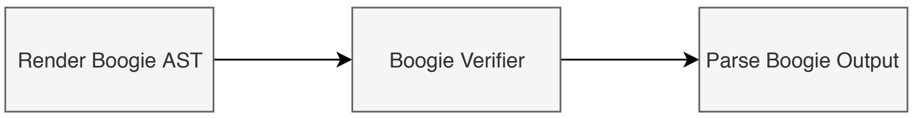


Figure 5.7: Overview of verification stage the of the Flint compiler

5.3.5 Inconsistent Assumptions

To detect inconsistent assumptions, we augment the Boogie code created from translating the Flint smart contract into a form which makes inconsistent assumptions explicit and we then parse the verification errors generated by the verifier to determine whether any of the Boogie procedures had inconsistent assumptions.

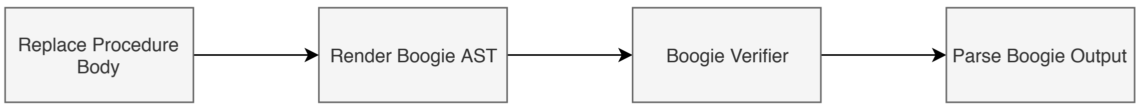


Figure 5.8: Overview of inconsistent assumptions detection

As discussed in section 4.4, inconsistent assumptions occur when a procedure's pre-conditions are logically equivalent to $false$. When this happens, it is trivial for the verifier to prove any condition. Therefore, we replace the procedure bodies of all procedures with $assert(false)$. We would expect the verifier to raise an error for this proof obligation, however it is possible that it does verify, when the enclosing procedure has inconsistent assumptions. Therefore, after we have replaced each procedure body with $assert(false)$, we parse the verification output and track which procedures failed to verify. The ones which were able to successfully verify have inconsistent assumptions. We record this and raise a compiler warning.

```
contract C {
    var j: Int = 0

    invariant (j == 10)
}

C :: (any) {
  func inconsistentAssumptions()
      pre (j == 5)
  {
      // ... Function body
  }
}
```

Listing 5.9: Contract with inconsistent assumptions

```
var j_C: int;

procedure inconsistentAssumtptions()
  // Pre Conditions
  requires (j_C = 5);

  // Invariants
  requires (j_C = 10);
  ensures (j_C = 10);

  modifies j_C;
{
  // Replaced procedure body
  assert (false);
}
```

Listing 5.10: Boogie translation of contract to detect inconsistent assumptions

5.3.6 Unreachable Code

In this section we discuss our implementation of the detection of unreachable code. We proposed a method of doing this in section 4.5, which we implement here. Figure 5.9, demonstrates the verification pipeline we implement to verify if any unreachable code exists.

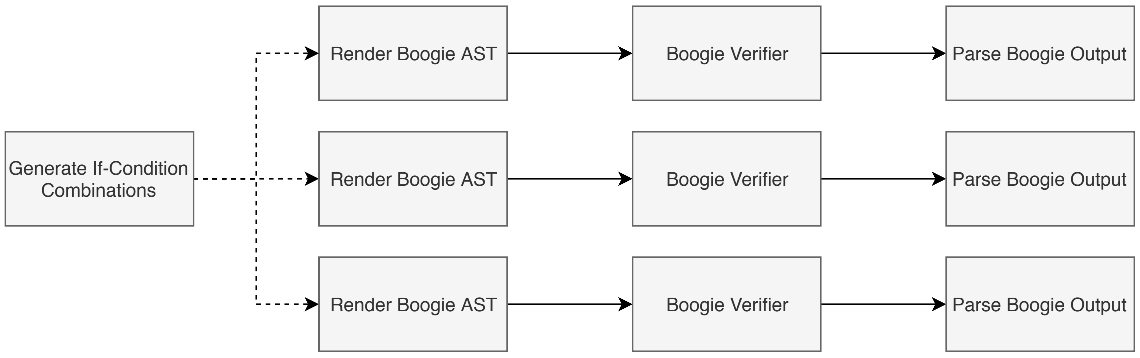


Figure 5.9: Overview of unreachable code detection

In order to implement checking for unreachable code, we must traverse the Boogie AST for every if-condition and assert whether it is true or false. If we test more than one if-statement, in a procedure, at a time, it is possible that the testing of other if-conditions could interfere with other if-condition verifications. An example of this is shown in listing 5.11.

```
assert (a > 10); // 1 - Fails
//if (a > 10) {
//}

assert (b > 10); // Since 1 fails, this assertion isn't attempted
//if (b > 10) {
//}
```

Listing 5.11: Example of how if-condition tests can interfere

To avoid this, we test only one if-statement at a time. As we traverse the Boogie AST, every if-statement we visit emits a new Boogie AST which test the if-condition. As we test if the if-condition is always true or always false, we generate 2 new Boogie ASTs. For n if-statements used within a smart contract, the verifier is invoked $2 * n$ times. We discuss the limitations of this approach in the evaluation section 6.4.2.

```
func unreachable(a: Int)
    pre (a > 10)
{
    if (a > 10) {
        // ...
     } else {
        // Execute important function
     }
}
```

```
procedure unreachable(a: int)
    requires (a > 10);
{
    // Test true case
    assert (a > 10);
}
```

```
procedure unreachable(a: int)
    requires (a > 10);
{
    // Test false case
    // Fails ⟹ Unreachable
    assert (¬(a > 10));
}
```

5.3.7 Holistic Specifications

In this subsection we discuss how we implemented the automatic verification of holistic specifications in the Flint compiler. In order to translate the holistic specification into Boogie, we add an extra post-processing step to the Boogie translation pipeline, previously discussed in section 5.3. In this extra step, we translate the holistic specifications into their Boogie representation; create the code which the Symbooglix will execute; parse Symbooglix's output; and return any verification failure to the user.

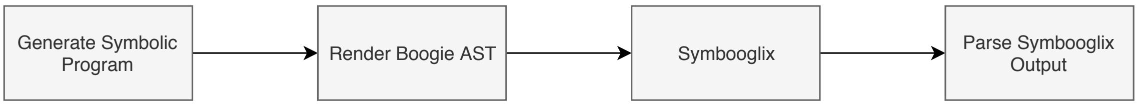


Figure 5.10: Overview of holistic specification verification

To begin, we first expanded the grammar of the Flint parser to include the `will` operator in contract declarations. This also involved extending Flint's abstract syntax tree to include a representation of `will`. Then to translate `Will(A)`, we first translate the Flint expression $\boldsymbol{A}$ into Boogie. We then use the method described in section 4.7.3, to create the Boogie encoding for *Will(**A**)*. Finally, the Boogie representation of the holistic specification is appended to a copy of the translated Flint smart contract and Symbooglix executes it.

When Symbooglix terminates, it generates an output file summarising the results of the symbolic execution. We parse this to file to determine whether Symbooglix discovered any execution paths which did not satisfy the specification.

```
# Termination Counter (ONLY_NON_SPECULATIVE) info
TerminatedWithoutError: 675
TerminatedAtFailingAssert: 2
TerminatedAtUnsatisfiableEntryRequires: 0
TerminatedAtFailingRequires: 0
TerminatedAtFailingEnsures: 0
TerminatedAtUnsatisfiableAssume: 0
TerminatedAtUnsatisfiableEnsures: 0
TerminatedAtUnsatisfiableAxiom: 0
TerminatedWithDisallowedSpeculativePath: 0
TerminatedAtGotoWithUnsatisfiableTargets: 0
TerminatedWithDisallowedExplicitBranchDepth: 0
TerminatedWithDisallowedLoopBound: 0
TerminatedWithUnsatisfiableUniqueAttribute: 0
```

Listing 5.12: Symbooglix output file

If there are any verification failures, me map them back to the holistic specification responsible. If required, this allows us to return an error message back to the user, that the contract doesn't satisfy its holistic specification. We give an example holistic warning, produced by the compiler, below.

```
This holistic spec could not be verified at line 6, column 8:
  will (i == 10)
       ^^^^^^^^^
  Warning
       Number of runs: 12
       Number of successes: 10
       Number of failures: 2
```

Listing 5.13: Example of holistic error message returned by Flint

5.4 Compiler Verification output

In order to provide meaningful failure messages to the user, we must translate the Boogie verifier's output into something meaningful, within the context of the Flint smart contract being verified. In this section, we discuss how we do this by parsing the output from the Boogie verifier and mapping each failing Boogie proof obligation to the responsible Flint line.

5.4.1 Parsing Boogie errors

Boogie outputs errors in a structured, human readable, form. The verifier emits errors for pre and post-conditions, assertions, and loop invariants. We are able to match each type of error to a template, thus allowing the Flint compiler to detect the type of verification error which as occurred. Below we show example verification output generated by Boogie.

```
Boogie program verifier version 2.3.0.61016, Copyright (c) 2003-2014,
    Microsoft.
program.bpl(417,1): Error BP5002: A precondition for this call might not
    hold.
program.bpl(345,1): Related location: This is the precondition that might
    not hold.
Execution trace:
    program.bpl(406,1): anon0
    program.bpl(415,1): anon6_Then
program.bpl(534,1): Error BP5001: This assertion might not hold.
Execution trace:
    program.bpl(508,7): anon0
    program.bpl(512,1): anon6_LoopHead
    program.bpl(512,1): anon6_LoopDone
    program.bpl(534,1): anon5

Boogie program verifier finished with 12 verified, 2 errors
```

Listing 5.14: Example Boogie Verification output

We can see that each verification failure has an associated failure code, and execution trace. Table 5.1 lists the failure codes emitted by the verifier when a proof obligation cannot be satisfied.

Failure Code	Proof obligation
BP5001	Assertion
BP5002	Pre-condition
BP5003	Post-condition
BP5004	Loop Invariant

Table 5.1: Boogie failure types

We extract the failure type and the file where the failure occurred. When applicable, we also parse the associated failure locations that the verifier has detected. For example, we also extract the location of the failing pre-condition when a procedure call cannot be verified. An example output illustrating the form of the related information, produced by the verifier, is also shown above.

5.4.2 Generating Flint Errors

In order to return meaningful verification failure messages to the user, we must be able to find the responsible line in the original Flint smart contract so we can return this to the user. Figure 5.11 gives a high level diagram of how the Flint verification error message, shown in listing 5.18, is generated.

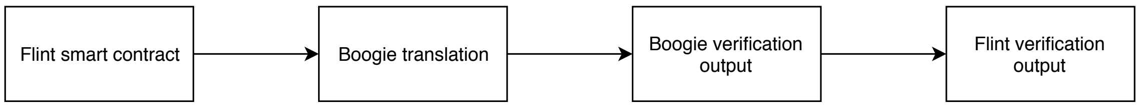


Figure 5.11: Overview of how a Flint verification error is generated

In order to determine the Flint line responsible for the Boogie verification failure on any given line, we need to create a mapping from Boogie line numbers to Flint line numbers. While we are constructing the Boogie AST, we do not track the final position, in the rendered Boogie file, of each node in the AST. Instead, when we render the Boogie AST to form the Boogie file and we annotate each line with an identifier unique to its responsible Flint line. An example of the annotated Boogie code is shown in listing 5.16.

This approach requires rendering the Boogie AST, and then re-parsing the generated file to determine the mapping from Boogie line to Flint unique identifier. In the examples below, we would create a mapping from Boogie line 4 → id 313938334, and a mapping from id 313938334 → Flint line 3. We could have maintained this mapping as we render the Boogie file. However, this would have increased the complexity of Boogie AST rendering, as would have to track the current Boogie line which is being rendered.

```
1  let i: Int = 10
2  // Line 3 has id 313938334
3  assert (i == 100)
```

Listing 5.15: Example Flint code

```
1  var i: int;
2  i := 10;
3  // #MARKER# 313938334
4  assert (i = 100);
```

Listing 5.16: Annotated Boogie code

```
example.bpl(4,1): Error BP5001: This assertion might not hold.
```

Listing 5.17: Boogie verifier output

Therefore, when a Boogie verification error occurs, such as will happen in listing 5.15, we can locate the identifier in the Boogie code and use it to determine the responsible Flint line. Listing 5.18 is an example verification error message, returned by the Flint compiler. We can see that maintaining the mapping from Flint line to Boogie line allows the Flint compiler to return meaningful error messages to the user.

```
Error in Example.flint:
Could not verify assertion holds at line 3, column 0:
    assert (i == 100)
    ^^^^^^^^^^^^^^^^
Contract specification not verified
```

Listing 5.18: Example verification output generated by Flint compiler

5.5 Verification Testing framework

It is important that we test our verifier to make sure that the Boogie translations are done successfully and that the semantics of the Flint operators are translated correctly. In this section, we discuss how we created a corpus of translation and semantic tests, and created a test runner, to test our verifier. For every new feature, we added test cases to check its behaviour. We used the verifier test suite extensively throughout the development of the verifier, to make sure that we were not introducing bugs or breaking old functionality. As of May 24, 2019, the test suite contains 22 test smart contracts.

5.5.1 Test runner

We developed the test runner to execute the Flint verifier over a corpus of Flint smart contracts, and identify failing smart contracts. We did this by creating the `run_verifier_tests.py`[5] Python[6] script which scans the `Tests/VerifierTests/tests` directory, within the Flint compiler repository, and executes the Flint compiler on each one. It does this in parallel to minimise the time it takes for the test suite to run. The test runner records whether each smart contract was successfully verified, with exit code 0, or failed. Listing 5.19 gives an example of the output generated by the test runner.

```
Verification tests
Total: 22
Passed: 19
Skipped: 0
Failed: 3
        Fail: Tests/VerifierTests/tests/verifyLoops.flint
        Fail: Tests/VerifierTests/tests/verifyHolisticEscrow.flint
        Fail: Tests/VerifierTests/tests/reportSyntaxErrors.flint
```

Listing 5.19: Output generated from verifier test runner

5.5.2 Translation tests

The translation tests check that the verifier is able to translate Flint constructs, of varying complexity. Listings 5.20 and 5.21 are snippets of two translation test smart contracts, showing two different levels of Flint language complexity being tested.

```
//VERIFY-CHECK
enum A: Int {
  case one
  case two
  case three
}

contract Test { }

Test :: (any) {
  public init() {}

  public func test() {
    var o: A = A.one
    var t: A = A.three
    assert(o != t)
  }
}
```

Listing 5.20: Enum test contract [7]

```
//VERIFY-CHECK
contract Array {
  var as: [Int]
  var bs: [Int: Int]

  var cs: [[Int]] = []
  var ds: [[[Int]]] = []
  var es: [Int: [[Int]]] = [:]
}

Array :: (any) {
  func dictKeys() {
    let bsKeys: [Int] = bs.keys

    let esKeys: [Int] = es.keys
    if esKeys.size > 0 {
     let esSubSize: Int = es[esKeys[0]].size
    }
  }
}
```

Listing 5.21: Extract of array test contract [8]

The `examples/casestudies`[9] directory in the Flint repository was used as inspiration to create the initial translation tests. The test suite smart contracts exercise a variety of Flint's language constructs. We make each test case focused on testing a particular aspects of the Flint language, so it is quick to discern issue based on the name of the failing smart contract.

5.5.3 Semantic tests

The semantics tests check whether the verifier is maintaining semantics of the original Flint constructs. We do this by checking whether certain proof obligations can or cannot be satisfied. To do this, we introduce a simple annotation language, for the test runner, which we use on the semantic

[5] https://github.com/YianniG/flint/blob/master/Tests/VerifierTests/run_verifier_tests.py
[6] https://www.python.org/
[7] https://github.com/YianniG/flint/blob/master/Tests/VerifierTests/tests/enum.flint
[8] https://github.com/YianniG/flint/blob/master/Tests/VerifierTests/tests/verifyArrayDictionaries.flint
[9] https://github.com/YianniG/flint/tree/master/examples/casestudies

test Flint smart contracts. The annotations are parsed by the test runner, expressing whether the indicated proof obligation should successfully verify or not. It was important that we could explicitly test whether specific proof obligations should or should not verify.

The annotation language we introduce are the two operators `//VERIFY-PASS` and `//VERIFY-FAIL`. These Flint comments indicate whether the proof obligation, on the following line, is expected to pass or fail verification. The verification test runner parses test contracts and compares the outcome of running the verifier on the smart contract with the expected verification output, indicated by the annotations. If the verification output does not match the annotations, the test runner returns a failure. Listing 5.22 gives an example semantic test, found in the verifier test suite.

```
//VERIFY-CHECK
contract VerifyPrePostConditions {
  var ten: Int = 10

  invariant (10 == 10)
  invariant (ten == 10)
}

VerifyPrePostConditions :: (any) {
public init() { }

func testParamsPreConditions(input: Int, increment: Int)
    pre (input == 10)
  {
    let input2: Int = input + increment
    //VERIFY-PASS
    assert (input2 == 10 + increment)
  }

  func failPreCondition()
  {
    //VERIFY-FAIL
    testParamsPreConditions(input: 0, increment: 0)
  }

  //VERIFY-FAIL
  func failPostCondition()
    mutates(ten)
    post (ten == 10)
  {
    ten = 5
  }
}
```

Listing 5.22: Extract of semantic test contract [10]

5.6 Flint Verification System CLI

By default, Flint's verification system is turned on for whenever a Flint contract is compiled. As part of the implementation of our verification features to the Flint compiler, we also extended the Flint command-line API. In figure 5.23 below, we outline Flint's updated CLI. We added the options found under `Verifier Options`, which allows smart contract authors to configure the output and behaviour of Flint's verification system.

[10] https://github.com/YianniG/flint/blob/master/Tests/VerifierTests/tests/verifyPrePostConditions.flint

5.7 Remarks

We have implemented a well designed and extensible verification module for the Flint compiler. We discussed the architecture we implemented in the verifier, and we also explored the design considerations which lead to this architecture. We discussed how our use of an intermediate representation allowed us to spread the complexity of the translation system over two stages, translation and resolution.

Our implementation of the translation scheme proposed in chapter 3 allows us to translate a variety of Flint contracts to Boogie. Furthermore, we discussed how our rendering of the Boogie AST and parsing of Boogie verification output, allows the Flint verification system to return meaningful messages to the smart contract author. We discussed how we implemented and use a test runner, and produced a test suite of smart contracts, to test that our translation and verification of Flint contracts is semantically preserving.

Finally, we discussed how the Flint verification system is turned on by default and how we introduced command line flags which allow the smart contract author to interact with Flint's verification system.

```
Usage:

  $ flintc <input files>

Arguments:

  input files - The input files to compile.

Compiler Options:
  --emit-ir [default: false]
    - Emit the internal representation of the code.

  --ir-output [default: ]
      - The path at which the IR file should be created.

  --emit-bytecode [default: false]
    - Emit the EVM bytecode representation of the code.

  --skip-code-gen [default: false]
    - Skip code generation

  --dump-ast [default: false]
    - Print the abstract syntax tree of the code.

  --verify [default: false]
    - Verify expected diagnostics were produced.

  --quiet [default: false]
    - Suppress warnings and only emit fatal errors.

  --no-stdlib [default: false]
    - Do not load the standard library

Verifier Options:
  --dump-verifier-ir [default: false]
    - Emit the representation of the code used by the verifier.

  --print-verifier-output [default: false]
    - Emit the verifier's raw verification output

  --skip-holistic [default: false]
    - Skip checking holistic specifications

  --skip-verifier [default: false]
    - Skip automatic formal code verification

  --print-holistic-run-stats [default: false]
    - Emit the holistic verifier's engine's run stats

  --max-transaction-depth [default: 5]
    - Set the max transaction depth to explore for the holistic verifier

  --holistic-max-timeout [default: 86400]
     - Set the max timeout (s) for the holistic verifier
```
Listing 5.23: Flint compiler's command line interface

Chapter 6

Evaluation

In this chapter we evaluate the result of our contributions to the Flint compiler. Due to the nature of the project, this is mostly a qualitative assessment, however we use quantitative methods where appropriate.

Our contributions use the cutting edge of formal automatic verification, enabled by the Boogie verifier [11], and a novel translation from Flint to Boogie to create an automatic verification system within the Flint compiler. Before our work in this project, the Flint compiler contained none of the specification syntax, or verification systems, we evaluate in this section. We use the Boogie verification system and our translation scheme to automatically verify Flint smart contracts against user-supplied specifications. Our translation from Flint to Boogie (see chapter 3) translates Flint's functional and holistic specifications into Boogie, where we either use static analysis or symbolic execution (see section 4.7.3) to determine whether Flint smart contracts satisfy their specifications. We show that we are able to mitigate against a class of immutability bugs with our introduction of functional and holistic specification syntax into Boogie.

In the following sections, we qualitatively discuss the specification syntax and its limitations. We also compare the expressiveness and proving power of Flint's verifier, with other the existing automatic smart contract verification systems: the Solidity compiler, and the VerX system. Finally, we evaluate the other verification functionality that the Flint verifier offers, and briefly comment on the verifier's performance and technology stack.

6.1 Translation to Boogie

In this section we evaluate our translation scheme and implementation from Flint to Boogie. It is important for our translation scheme to maintain the semantics of Flint, otherwise the outcome of the Boogie verification does not give any meaningful guarantees for the original Flint smart contract.

In chapter 3, we propose our translation scheme and argue it's correctness. We propose translations for most Flint constructs and types, we can translate every Flint contract in the compiler's test suite, and we can translate 23 out of the 25 contracts in the compiler's example case studies. Here we look at our translation scheme and implementation, and discuss it's limitations.

Translation Omissions

In section 3.2 we propose a translation for the string Flint type, however due to time constraints we did not implement the translation to Boogie. Therefore the Flint verifier does not support verification of contracts with string variables. Both of the case study Flint contracts, which the Flint verifier cannot translate, use string variables. Also due to time constraints, we do not propose a translation for Flint fallback functions.

Flint Language Semantics

As mentioned earlier, it is important that our translation scheme maintains the semantics of Flint, otherwise the verification result does not hold for the original Flint contract. In chapter 3, we outline the translation scheme we propose, and argue that it is semantic preserving. However, we only present our translations and informally argue their correctness. We present no formal proofs of correctness that the translation preserves the semantics of the original Flint compiler. To mitigate this, as discussed in section 5.5, we use a test suite to check the correctness of our translations. The test suite contains smart contracts which exercise each Flint language construct, and asserts the correct behaviour.

Furthermore, in order to present detailed proofs, that the semantics of our translations preserve Flint's semantics, we must formalise Flint's semantics. Currently, the behaviour of the Flint compiler defines the semantics of each Flint construct. Therefore, we must first formalise Flint's semantics and then it can be used to formally show that our translation scheme preserves the semantics of the original Flint constructs. We discuss this further in the future work section 7.1.1.

EVM Semantics

Our translation scheme does not capture the semantics of the EVM, executing the Flint code deployed to the Ethereum blockchain. For example, the EVM has memory constraints which we do not capture, furthermore we assume no hash collisions when hashing dictionary elements. Any behaviour which occurs as a result of these or other low level EVM properties is not modelled by our translation scheme.

Boogie's Limitations

With our proposed translation scheme and functional and holistic specifications, we can translate a Flint contract and specification into Boogie. Whether the specification can be verified is dependant on Boogie's proving power. Boogie is a sound, but incomplete verification tool. If it cannot verify a specification, it does not mean that the specification cannot be satisfied, only that Boogie cannot satisfy it. As of June 16, 2019, Boogie benefits from active development. As Boogie's proving power increases, the complexity of the Flint specifications which it can prove will also increase. Below we give an example of an NP-class problem, which gives an example of the limitations of Boogie's proving power.

```
func testPrime7919(a: Int, b: Int)
  pre (a > 1 && a < 7919);
  pre (b > 1 && b < 7919);
{
   assert (a * b != 7919) // 7919 is prime, this function verifies.
}

func testPrime27644437(a: Int, b: Int)
  pre (a > 1 && a < 27644437);
  pre (b > 1 && b < 27644437);
{
   assert (a * b != 27644437) // 27644437 is prime, but this fails to verify
}
```

Listing 6.1: Limitations of Boogie's thereom proving power

Limitations of Array/Dictionary Literal Translations

We are able to model Flint arrays, structs, dictionaries, integers and addresses using our translation scheme. Therefore we can use our translation scheme to prove properties about these Flint constructs. Since these Flint constructs are the building blocks of Flint smart contracts, basic Flint contract functionality can be verified using our approach. We proposed an approach to represent Flint array literals, 3.8. However this approach only works when the literal is used within a Flint function, because the approach requires assigning each element to the Boogie map, which represents the Flint literal.

```
var array: [Int] = [0, 1, 2]
```

Listing 6.2: Array literal declaration

```
var array: [int]int;
var size_array: int;

array[0] := 0;
array[1] := 1;
array[2] := 2;
size_array := 3;
```

Listing 6.3: Translation of array literal declaration

However, this approach does not work, if a user wants to use an array or dictionary literal in a functional or holistic specification. This is because we do not propose a way of representing the array literal as a logical formula, therefore the specification language of Flint is not as expressive as the rest of Flint. Only a subset of Flint expressions can be used in Flint specifications. Namely expressions, which are not assignments, collection literals, or function calls.

Operator Short-Circuiting

In section 3.9 we propose our scheme for translating Flint operators and in section 5.3.1 we propose our approach to implementing our translation scheme, for Flint statements which generate pre and post-amble Boogie statements. We describe how, for Flint expressions such as function calls and compound assignments, we must first emit Boogie statements to perform the Flint expression's action, and then use the result later. We give an example below.

```
let b: Bool = func2() || func1()
```

Listing 6.4: Example Flint expression requiring a Boogie preamble

```
var b, f_result1, f_result2: bool;

// Preamble
call f_result1 := func1();
call f_result2 := func2();

// Collecting result from Preamble
b := f_result1 ∨ f_result2;
```

Listing 6.5: Incorrect translation of binary operands

Our Flint translation does not correctly implement the short circuiting behaviour of the binary logical operators `||` and `&&`. Our translation scheme evaluates both operands and then evaluates the expressions. An example of our translation not preserving the short-circuiting semantics of the binary operators is given, in listing 6.5, above. In this example, only `func1` should be called, if it returns `true`. However, `func1` and `func2` will always called. In the further work section 7.1.2 we discuss our proposal to update our translation scheme and implementation to correctly respect the short-circuiting semantics of the boolean operators.

Caller Group Translation

In section 3.6.2, we present our translation for Flint caller groups. Here we discuss the limitations, for static analysis, of the approach we present. When we translate protection blocks with caller groups, which are global contract properties, we generate Boogie procedure pre-conditions which the verifier can use to statically determine if the callee has authorisation to call the function. However, if the caller group is a function, we cannot translate this directly into a Boogie procedure pre-condition. As we discuss in section 3.6.2, Boogie pre-conditions cannot contain procedure calls. Therefore, we introduce Boogie statements at the beginning of the generated procedure to dynamically test the caller group. Semantically, there is no issue with this approach as this translation matches Flint's semantics. However, this approach does not allow the Boogie verifier to statically determine whether a calling procedure satisfies the Flint function's caller group, when the caller group is a function, because it is not mentioned in the procedure's pre-conditions. Therefore the Flint verifier cannot highlight to the user if any calling functions do not satisfy a called function's caller group.

Trait Modifies Clause Inference

In our implementation of our Flint to Boogie translation, we must infer the global state that trait-defined functions implicitly modify, (see section 5.2.4). This is required so the generated Boogie procedures have the required *modifies* clauses. We argue that inferring the global state that trait-defined functions modify is the correct behaviour. However by inferring the state that the trait-defined function modifies, it is not made explicit that a potentially public function could modify contract state. Therefore it is possible that a smart contract author could look at the declaration of a trait function, and it's implementation, and conclude that the function doesn't modify state, when a conforming type does[1]. The Asset trait is an example of this, shown below.

```
struct trait Asset {
  // .. Other declarations

  func transfer(source: inout Self,
    amount: Int)
  {
    if source.getRawValue() < amount
    {
      fatalError()
    }

    source.setRawValue(value:
    source.getRawValue() - amount)
    setRawValue(value: getRawValue()
    + amount)
  }
}
```

Listing 6.6: Extract of Asset trait[2]

```
struct Wei: Asset {
  // .. Other declarations

  func setRawValue(value: Int) -> Int
    mutates (rawValue)
    post(self.rawValue == value)
  {
    rawValue = value
    return rawValue
  }
}
```

Listing 6.7: Extract of Wei struct[3]

In the examples above, we can see how the Wei struct's inherited `transfer` method modifies contract state because it calls `setRawValue`, but the trait's function declaration does not need to make this explicit.

Observed Alternative Assertion Semantics

In section 3.15.2, we propose translating Flint's `assert` statement, into Boogie's `assert` proof obligation statement. Flint's semantics of the `assert` statement, were such that if the assertion was not satisfied at run-time the contract would revert. Our Boogie translation preserves these semantics as the verifier stops considering its current path, if the `assert` cannot be satisfied. However, `assert(A)` was also used as a quick hand way of writing:

```
if (!A) {
  fatalError()
}
```

That is, the intention is not to use the `assert` statement as a proof obligation, but instead a run-time test that `A` holds. The contract author is only interested in the execution paths where `A` holds. Our translation does not maintain this use case. We give an example below:

```
public func transfer(to: Address, value: Int) -> Bool
    mutates (balances)
{
    // Both assertions fail verification
    assert(balances[caller] >= value)
    assert(to != 0x00000_00000_00000_00000_00000_00000_00000_00000)

    balances[caller] -= value
    balances[to] += value

```

[1]Side note: A potential way of solving this issue could be through the use of the `mutating` modifier. Flint's syntax and semantics could be updated to allow trait-declared functions to be annotated with the `mutating` modifier. Trait-declared functions, which have the `mutating` modifier, would indicate that conforming functions are allowed to modify global state, otherwise a conforming function may not modify global state.

[2]Taken from: https://github.com/flintlang/flint/blob/master/stdlib/Asset.flint

[3]Taken from: https://github.com/flintlang/flint/blob/master/stdlib/Asset.flint

```
    emit Transfer(from: caller, to: to, value: value)
    return true;
}
```

Listing 6.8: Extract from SimpleToken.flint case study[4]

We argue that this is not a limitation of our translation scheme, and instead a misuse of the `assert` statement. Flint provides other syntax, such as pre-conditions and if-statements, to achieve this. In the example above the `assert` statements are acting as pre-conditions, therefore we argue the new pre-condition syntax should be used for this case. We give an example below:

```
public func transfer(to: Address, value: Int) -> Bool
    mutates (balances)

    pre(balances[caller] >= value)
    pre(to != 0x00000_00000_00000_00000_00000_00000_00000_00000)
{
    balances[caller] -= value
    balances[to] += value

    emit Transfer(from: caller, to: to, value: value)
    return true;
}
```

Listing 6.9: Updated SimpleToken.flint example using pre-conditions

Remarks

Given the analysis of the limitations of our translation scheme above, we acknowledge that our translation scheme does not encompass all of Flint's constructs, and does not guarantee a complete semantic translation from Flint to Boogie. However, it is strong enough for the Flint compiler to verify all of the smart contracts and functional specifications we describe below.

6.2 Functional Specifications

In this section we evaluate the expressiveness, of Flint's functional specifications; the limitations of the verification system, for functional specifications; and the level of feedback given to the user, for an unsatisfied specification.
The Flint compiler is currently the only tool which can verify Flint smart contracts, therefore we compare our functional specifications with the functional specifications used by the Solidity compiler's SMT checker [10], as it also offers automatic formal verification of user-supplied functional specifications[5].

6.2.1 Expressiveness

Here we evaluate the expressiveness of the Flint functional specifications, and compare with Solidity functional specifications. We look to evaluate the syntax used to write functional specifications, the specification predicates, and the language constructs supported by the verifier.

Specification Predicates

Here we discuss the different specification predicates supported by Flint and Solidity. We explore how, and whether, each language supports invariants, assertions, and pre and post-conditions.

In table 6.1, we list the predicates that Flint and Solidity provide. At a high level, Solidity's smaller set of predicates give the contract author greater levels of control in how to formulate their specification. Flint's larger set of predicates gives the compiler more contextual information and control over how to verify the specification. We discuss this in more concrete terms.

[4]https://github.com/YianniG/flint/blob/master/examples/casestudies/token/StandardToken.flint
[5]The Solidity compiler's SMT checker is still in development, bug fixes and new features are constantly being added. Our analysis was performed in May 2019, using Solidity compiler v0.5.9.

Solidity	Flint
`require`	`pre`
`assert`	`post`
	`assert`
	`invariant`

Table 6.1: Flint and Solidity Specification Predicates

```
pragma experimental SMTChecker;

contract C
{
  function g(uint x) public {
    uint y = x + 1;
    require (y < 100);
    // some computation with y

    assert (y > 100);
  }
}
```

Listing 6.10: Example of Solidity `require` and `assert`

Through the use of `require`, the contract author can introduce pre-conditions for their functions, midway through the function body, this is shown in listing 6.10. `require` statements are checked at run-time to make sure it is safe to make the assumption, else the transaction is reverted, this makes the Solidity verifier's approach sound. Flint does not have an explicit predicate like `require`, which allows the introduction of assumptions during a function's body[6]. Flint's `pre` predicate is semantically the same as `require`, however it cannot be used in the function's body. Solidity's `assert` statements can also be used at the end of a function body to represent post-conditions, shown in listing 6.10. However, the issue with non-explicit pre and post-conditions is that it can be hard to prove properties over recursive functions. The verifier does not know what conditions must be true to call the function, and the properties which must hold by the end of the function. The Flint verifier is able to prove properties over recursive calls, as it can use the explicit pre and post-conditions given to show that the property holds for the base and recursive case, however the Solidity compiler cannot do this. We give an example recursive Solidity contract, and it's Flint translation in listings 6.11 and 6.12.

```
pragma experimental SMTChecker;

contract C
{
 uint a;
 function g() public {
   if (a > 0)
   {
     a = a - 1;
     g();
   } else
     assert(a == 0); // Fails
 }
}
```

Listing 6.11: Example of Solidity recursive call[7]

```
contract C {
  var a: Int

  invariant (a >= 0)
}

C :: (any) {
  public init(a: Int)
    pre (a >= 0)
  {
      self.a = a
  }

  public func g()
    mutates (a)
  {
    if (self.a > 0) {
      self.a -= 1
      g()
    } else {
      assert (a == 0) // Verifies
    }
  }
}
```

Listing 6.12: Example Flint recursive call

We can see how the `require` and `assert` Solidity operators give contract authors lots of flexibility in how to express their specifications. However, the Flint operators encode more information, such as the purpose of the predicate (for example, post-condition or assertion), which allows the verifier to prove properties over complex control flows, such as recursive functions.

[6]It is possible to emulate this behaviour using if-statements and the `fatalError()` function. We give an example listing, in the appendix, A.1. However with this approach, the compiler is not able to statically determine if the function's callers satisfy its pre-condition.

[7]Taken from: `https://github.com/ethereum/solidity/blob/develop/test/libsolidity/smtCheckerTests/functions/functions_recursive.sol`, June 2019

Solidity does not have a separate predicate to express contract invariants. If a contract author would like to express a property which must be maintained throughout the lifetime of the contract, they must manually add the pre and post-conditions to all of their contract functions. In Flint, the compiler does this for us, we give an example below.

```
pragma experimental SMTChecker;
contract C
{
  uint a;
  function g() {
    require (a > 0);
    // .. function body
    assert (a > 0);
  }

  function h() {
    require (a > 0);
    // .. function body
    assert (a > 0);
  }
}
```

Listing 6.13: Example contract checking invariant

```
contract C {
  var a: Int

  invariant (a > 0)
}

C :: (any) {
  func g() {
    // .. function body
  }

  func h() {
    // .. function body
  }
}
```

Listing 6.14: Example contract with invariant

We can see that it is possible to encode contract invariants in Solidity, the author can use `require` and `assert` statements to ensure that their invariant property holds by the end of each function. However the burden is on the contract author to make sure that the every Solidity function contains pre and post-conditions and that each one expresses exactly the same property. Flint does not give this level of control to the contract author, they cannot customise Flint's invariant semantics, instead the compiler guarantees that the invariant semantics are consistently and systematically applied to each function in the contract.

Overall Solidity's predicates allows for greater control over the expression of functional specifications, but we see that this can limit what the verifier is able to prove and places a burden on the programmer to make sure that their specification is consistent throughout their contract. Conversely Flint limits where it's functional predicates can be used, but it is able to prove properties over recursive functions, and is able to express the same properties as Flint.

Supported Language Constructs

We aim for Flint's proposition syntax to be expressive, in order for the specifications to be able to capture as much of the behaviour of the contract as possible. To evaluate how expressive Flint's propositions are, we compare the smart constructs that it's verifier supports with Solidity's verifier.

Development on the Solidity SMT Checker began in October 2018[8], currently only a subset of the Solidity constructs and types are supported. As discussed earlier in the section, Flint's verifier also supports a subset of Flint's language constructs. Below we outline what the Solidity verifier can and cannot translate[9]. We look at each class of smart contract construct, for Flint and Solidity, and compare what each verification system is and isn't able to express and verify.

Unsupported Construct Error

When the Solidity compiler attempts to verify an unsupported construct, it will emit a warning indicating that the construct is unsupported, and continue verifying what it can of the remainder of the contract. In contrast, Flint will stop the contract verification if it detects an unsupported construct. Even though the Solidity compiler's error message is more informative, it's approach is less safe than Flint's, as it still compiles the smart contract code. Because of the importance

[8]Taken from commit history: https://github.com/ethereum/solidity/commits/

[9]Taken from Solidity test suite: https://github.com/ethereum/solidity/tree/develop/test/libsolidity/smtCheckerTests

of avoiding bugs, it is important to verify the contract specification before a binary is produced. Therefore we argue that any kind of verification failure, by default, should stop compilation.

```
Error from contract.flint:
  Flint to Boogie translation error. Unsupported construct used.
Contract specification not verified
Failed to compile.
```

Listing 6.15: Flint compiler unsupported error

```
contract.sol:5:3: Warning: Assertion checker does not yet support
    constructors.
  constructor() public {}
  ^----------------------^
```

Listing 6.16: Solidity compiler error

Global Properties and Functions

Function	Flint	Solidity
`gasleft`	N/A	✓
`fatalError` / `revert`	✓	×
`pre` / `require`	✓	✓
`assert`	✓	✓
`send`	✓	×

Table 6.2: Supported global functions

Property	Flint	Solidity
Application binary interface	N/A	✓
Block Coinbase	N/A	✓
Block Difficulty	N/A	✓
Block Timestamp	N/A	✓
Block Number	N/A	✓
Transaction Gas Price	N/A	✓
Transaction Origin	N/A	✓
Sender Address	✓	✓
Received Wei Amount	✓	✓
Raw Transaction Call Data	N/A	×
Contract Balance	N/A	✓

Table 6.3: Supported global properties

The tables above outline the functions and properties which each verifier supports. We can see that both verifiers support a large subset of their language constructs. In particular, we see that the Flint verifier supports every global function and property, given by the Flint language. The Solidity verifier supports 13 out of 19 functions and properties, which limits the contracts which can be verified.

Functions

Behaviour	Flint	Solidity
Function Call	✓	✓
External Call	✓	✓
Function In-lining	✓	✓
Multiple Return values	N/A	✓
Recursive Functions	✓	×
Contract Constructor	✓	×

Table 6.4: Supported function behaviours

The Flint verifier is able to translate contract and struct function declarations, as well as function calls. The Solidity verifier is also able to translate contract function declarations, however we can see that it is unable to translate contract constructors or recursive functions. Both of these constructs are key components of Solidity contracts, this also limits the contracts which the Solidity compiler can verify. Other than multiple return values, which the Flint language doesn't support, the Flint verifier is able to translate all the function behaviours listed in table 6.4. Therefore, we can see that the Flint verifier can verify more types of contracts than the Solidity verifier.

Expressions and Data Structures

In table 6.5 and 6.6 below, we compare the types of expressions and data structures which are supported by the Flint and Solidity verifiers.

Expression Group	Flint	Solidity
Local Variables	✓	✓
Global Variables	✓	✓
Boolean Operators	✓	✓
Integer Operators	✓	✓
Compound Operators	✓	✓
Strings	×	×
Type Conversion	×	×
Array / Mapping Properties	✓	×
In-Line Assembly	N/A	×
`self` / `this` Keyword	✓	×

Table 6.5: Supported types of expressions

Data Structure	Flint	Solidity
Enums	✓	✓
Tuples	N/A	✓
Structs	✓	×
1-D Array / Mapping	✓	✓
Multi-Dimensional Array / Mapping	✓	×

Table 6.6: Supported Data Structures

We can see that the Flint verifier supports a larger variety of expression constructs, than the Solidity verifier. Due to time constraints, strings and type conversions are not supported. The Solidity compiler also does not support structs. Many smart contracts use structs, we prioritised support for structs in the Flint verifier so we could verify as many contracts as possible. In this regard, the Flint verifier can verify more types of smart contract than the Solidity compiler.

Control Flow

In table 6.7 we compare the supported control flow constructs of each verifier.

Structure	Flint	Solidity
If-Statement	✓	✓
For-Loop	✓	✓
While-Loop	N/A	✓

Table 6.7: Supported Control Flow types

We see that the Solidity and Flint verifiers support the control flow constructs of their languages. Control flow is an important part of a smart contracts, therefore it is important that the verifier can verify properties over them.

Specification Propositions

In section 4.1.1, we introduced the specification predicates: `returns`, `returning`, `arrayContains`, `dictContains`, `arrayEach` and `prev`. Each of these predicates expanded the functional specification language of Flint, enabling the user to write expressive propositions. Here we evaluate the propositions, by comparing them with the equivalent Solidity syntax.

For our comparison, we take the `returns` predicate. `returns` allows the user to specify, as a postcondition, a logical property which holds over the return value of the function. Solidity does not introduce a keyword, however it is possible to use assertions to emulate this behaviour by creating a temporary variable to contain the return value of the function. This allows the contract author to **assert** that the property holds over the temporary value, which contains the function's the return value, and then return the temporary value. An example of this is shown in listing 6.18 below.

```
func RPS(s1: Int, s2: Int)
  pre(s1 >= 0 && s1 <= 2)
  pre(s2 >= 0 && s2 <= 2)
  // ROCK == 0
  // PAPER == 1
  // SCISSORS == 2
  post (returns ((s1 == 0 && s2 == 2)
            || (s1 == 1 && s2 == 0)
            || (s1 == 2 && s2 == 1)))
{
  var outcome: Int = ((3 + s1 - s2))
  return outcome % 3 == 1 // S1 win?
}
```

Listing 6.17: Flint `returns` example [10]

```
function RPS(uint s1, uint s2) {
  require(s1 >= 0 && s1 <= 2)
  require(s2 >= 0 && s2 <= 2)

  uint outcome = ((3 + s1 - s2));
  uint returnValue = outcome % 3 == 1

  assert (returnValue == ((s1 == 0
    && s2 == 2)
       || (s1 == 1 && s2 == 0)
       || (s1 == 2 && s2 == 1)))
  return returnValue;
}
```

Listing 6.18: Solidity equivalent

We can see that the use of Flint's `returns` keyword makes the function's post-condition explicit, which aids understanding of the function behaviour. Furthermore, expressing the post-condition required 1 line, instead of 2 lines and a temporary local variable. Finally, adding the post-condition did not require changing the body of the function where, as a result of the change, unintentional side effects could occur. The addition of the `returns` enables concise specifications on the return value of Flint functions, in comparison to Solidity. The same is true of `returning`, `arrayContains`, `dictContains`, `arrayEach` and `prev`.

6.2.2 Provability

Flint's verification system only has utility if it can verify useful, non-trivial, specifications. Here we evaluate the complexity of specifications that the Flint compiler can verify. As a reference, we also compare the complexity of the specifications that the Solidity compiler can verify.

As we discussed earlier in this section, the Flint verifier cannot prove all provable specifications. Boogie's proving power is dependant on Z3's proving power, which is a sound but incomplete SMT solver. The Solidity verifier also uses Z3 as it's underlying SMT checker. We look to gain an understanding of the verification limitations of Solidity and Flint's systems.

In appendix B, we give example Flint smart contracts that the Flint verifier supports. We annotate the failing proof obligations, to indicate where the Flint verifier detects a violation. In appendix C, we give example Solidity smart contracts, which the Solidity verifier supports. We see that since the Flint verifier supports more of its language constructs, it therefore it can prove more properties than the Solidity verifier. We also see that for all the smart contracts which Solidity can verify, which have equivalents in Flint, Flint can verify them also. This is because Solidity and Flint share the same underlying SMT solver. As discussed earlier, the Flint verifier supports a large subset of Flint's language constructs, unlike the Solidity compiler, therefore we can annotate deployment-ready smart contracts with functional specifications. In appendix D.1, we present an example of how a functional specification might be written for an example Bank smart contract. We see that the Flint specification language is expressive enough to model the behaviour of contracts using many of Flint's language features, and that the Boogie verifier is strong enough to verify whether the specification is satisfied.

6.2.3 User Experience

For a verification system to be useful, it must also be easy to use and interpret. Here we discuss the user experience of the Flint verification system, and compare it with the Solidity verifier. We look at the differences in specification approach, and user feedback, of the two systems.

With the introduction of pre and post-conditions to function declarations, and the addition of struct and contract invariants, it is possible to write Flint functional specifications within Flint

[10]Inspired by: https://github.com/YianniG/flint/blob/master/examples/casestudies/RockPapersScissors.flint

smart contracts. These `pre`, `post`, and `invariant` predicates make explicit the purpose of each proof obligation. This contrasts with the Solidity verifier, where `assert` and `require` statements are interleaved in the function body, and asserts can be function post-conditions or function body proof obligations. We argue that using specification predicates, with explicit roles, makes it easier to understand and reason about the smart contract's specification.

The Flint verifier returns a verification status back to the user, when it has completed verifying a smart contract. If the contract successfully verifies a success message is returned, otherwise the compiler emits an error message outlining the location and type of verification failure that has occurred. This kind of meaningful information gives insight into the smart contract's erroneous behaviour.

```
Error in Example.flint:
Could not verify assertion holds at line 3, column 0:
    assert (i == 100)
    ^^^^^^^^^^^^^^^
Contract specification not verified
```

Listing 6.19: Example verification output generated by Flint compiler

However, verification feedback that the Flint verifier returns to the user is limited, in that the errors may not identify the root cause of why a proof obligation fails. The feedback that the verifier returns to the smart contract author, only describes that a failure has occurred, it may not provide a way to determine the actual root cause of the failure. For example, the Solidity compiler returns counter-examples for any verification failures it detects, which gives the smart contract author more information to help them understand the smart contact behaviour which does not satisfy its specification. We give an example below:

```
example.sol:9:9: Warning: Assertion violation happens here
        assert(y < 4);
        ^-----------^
  for:
  x = 9
  y = 4
```

Listing 6.20: Example verification output generated by Solidity compiler

We argued that functional specifications written in Flint are easier to understand, because of the explicit role of each predicate. We also acknowledge that the Flint verifier, while offering the same basic functionality, does not return as much information as the Solidity verifier. We propose generating counter-examples, using the Boogie representation of the contract, as a potential area of future work as it would allow the smart contract author to have a better understanding of their specification (see section 7.1.4).

6.2.4 Remarks

Flint's verifier supports a large subset of Flint's constructs, allowing the verifier to verify complex propositions which use many of Flint's features. We discussed the expressiveness of the specification language, and the benefits of Flint's large set of specification predicates. However, we acknowledge that supporting only a subset of Flint's language means that not all Flint smart contracts can be verified. This could be an area of further work, and further work could look to expand the types of specifications that Flint supports.

6.3 Holistic Specifications

In this section we evaluate the expressiveness and proving power of the holistic specifications we introduce to Flint. We do this by qualitatively evaluating the holistic specifications which Flint supports, and comparing them to the VerX system. We also discuss the limitations of our approach and areas of future work.

The VerX system[11] is currently the only other system which can automatically verify holistic specifications over smart contracts. Therefore we compare the expressiveness and guarantees of Flint's holistic specifications to VerX's. The work behind the VerX system is currently unpublished, therefore we cannot fully scrutinise it's implementation and compare it with our own. However we can compare the specification operators and the guarantees that the VerX system advertises.

Expressiveness

We extend Flint's syntax to support the holistic operator `will`. Smart contract authors can use this specification predicate to assert properties which may hold over multiple transactions on their smart contract. Here we evaluate the `will` predicate, and compare it with the `once` predicate offered by the VerX verification system.

The `will` operator has the same semantics as VerX's `once` operator, therefore we can directly the expressiveness each system's specifications.

```
// The escrow never allows the beneficiary to withdraw the
// investments and the investors to claim refunds.
property exclusive_claimRefund_and_withdraw {
  always(
    !(once(FUNCTION == Escrow.claimRefund(address))
     && once(FUNCTION == Escrow.withdraw()))
  );
}
```

Listing 6.21: Example VerX specification[12]

From listing 6.21 above, we see that the `once` predicate can be combined with other logical operators (in this case `&&`, and `!`). It is not possible to negate the outcome of `will`, and `will` cannot be combined with binary logical operators. It is possible to achieve to `once(A) && once(B)` in Flint, this is done by writing `will(A)` and `will(B)` as separate lines, shown in listing 6.22.

```
contract C {
  var i: Int = 0
  var j: Int = 20

  will(A)
  will(B)
}
```

Listing 6.22: Flint holistic example

Furthermore, VerX introduces the `FUNCTION` keyword which allows the user to assert that a specific function was called. Flint introduces no new operators for use in it's holistic specifications. We can see that in this regard, VerX has a more expressive specification language.

We discuss in the further work section, the addition of other predicates and keywords for Flint's holistic specifications (see section 7.1.5)

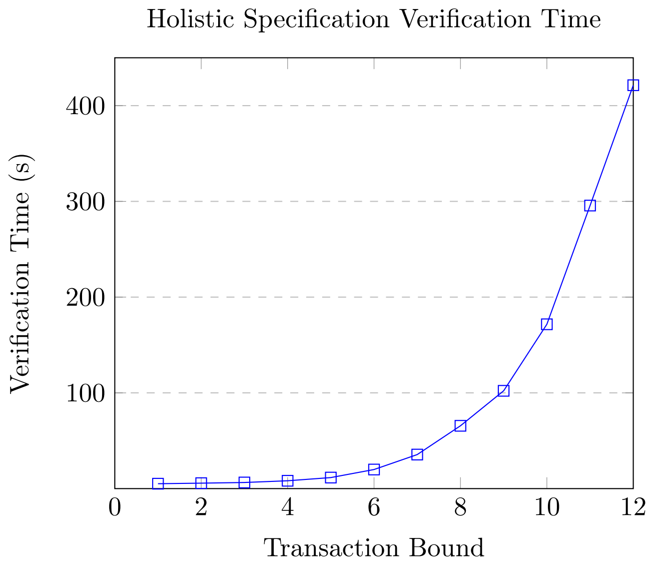
Proving Power

Here we evaluate the ability for Flint's verifier to check whether a holistic specification is satisfied. We compare Flint's verification guarantees, and the inherent limitations to our approach, with the VerX system and discuss potential areas of future work.

As we discussed in section 4.7.3, our method of checking holistic specifications provides a bounded guarantee. An inherent limitation of using symbolic execution is that they cannot explore unbounded models. Therefore we place a bound on the number transactions, to make the verification tractable. This bound changes the semantics of the Will operator, as we can only detect specification errors which occur within the transaction bound. For a bound of 5 (the default value), we can only prove that the holistic specification is satisfied if satisfaction can be shown in the first

[11] https://verx.ch
[12] Taken from: https://verx.ch

5 transactions. The approach is dependant on the proving power of the symbolic executors. As further states are explored, the path constraints that the solver has to solve become more complicated, the symbolic execution engine must be powerful enough to achieve this. We graph the verification time for the Flint smart contract A.2 below, to illustrate the increasing complexity of the path constraints.



The VerX system can prove their specifications over unbounded models. The system is able to verify that the specification holds for each possible state of a smart contract. They do this through abstract interpretation, which allows them to transform an unbounded model, into a bounded one, which they can then symbolically execute. The specification which is to be verified determines the abstraction used, this is because the abstraction process removes contract details which it deems unnecessary to verify the specification. In most cases, the abstraction process allows for successful verification of the specification[13]. Therefore through this process, the VerX system is able to verify more specifications than the Flint verifier, as it can verify properties which occur beyond the Flint verifier's transaction bound.

Remarks

We see that the Flint verifier can check holistic specifications, up to a given transaction bound. The bounded model checking reduces the strength of the prover and the guarantees provided by the verifier. Our verification approach is sound, but not complete as the verifier can only prove properties up to a bound of k transactions (approximately 5). We discuss that this is an area for further work, in section 7.1.5. In conclusion, our support of the the holistic predicate `will`, allows users to verify the behaviour of their contracts over multiple transactions.

6.4 Comparison of Verification Functionality

The Flint compiler's verification system offers more than just the verification of user supplied specifications. In this section, we compare the other functionality offered by the Flint compiler, powered by the verification system, with that offered by the Solidity compiler.

6.4.1 Inconsistent Assumptions

As we discuss later in this section, the Flint compiler does not provide any methods to validate the functional specification that the user has given. It is important to validate specifications to make sure that the specification captures the behaviour the author is interested in. Inconsistent

[13]We are currently waiting for the full VerX system paper to be published. These comments are informed through informal discussions we have had with the VerX team

assumptions can be the result of unexpected interactions between pre-conditions, as such, they are typically unintentional. Therefore, detecting inconsistent assumptions provides a basic form of validation of the author's specification. Here we evaluate the output of the inconsistent assumptions detection system.

The Solidity compiler does not detect inconsistent assumptions. It is possible to `require(false);` and `assert(false);`, without the compiler raising a warning to the user. As discussed in section 4.4, the Flint compiler detects if a function has inconsistent assumptions and raises a compiler warning.

```
contract C {
    var counter: Int = 1
    invariant (counter > 0)
}

C :: (any) {
    func inconsistent()
        pre (counter == 0)
        pre (counter > 20)
    {
        // Verifies
        assert (false);
    }
}
```

Listing 6.23: Inconsistent assumptions example

```
Warning in verifyInconsistentAssumptions.flint:
This function has inconsistent pre-conditions.
It will trivially verify. at line 6, column 4:
  func inconsistent()
  ^^^^^^^^^^^^^^^^^^^

  Note in verifyInconsistentAssumptions.flint:
  Caused by at line 8, column 12:
      pre (counter == 0)
          ^^^^^^^^^^^^^^

  Note in verifyInconsistentAssumptions.flint:
  Caused by at line 9, column 12:
      pre (counter > 20)
          ^^^^^^^^^^^^^^

  Note in verifyInconsistentAssumptions.flint:
  Caused by at line 3, column 4:
    invariant (counter > 0)
              ^^^^^^^^^^^^^


Contract specification verified!
```

The output of the Flint compiler, when it detects inconsistent assumptions, specifies the affected function and it's affected pre-conditions and invariants which may be the cause in the inconsistent assumptions. However, the verifier does not detect the minimal set of pre-conditions, and invariants, which cause the inconsistent assumptions. As shown in listing 6.23 above. This would precisely identify the responsible pre-conditions and invariants that the contract author must revisit. Currently, the smart contract author must determine which subset of the pre-conditions and invariants are causing the inconsistent assumptions. Therefore detecting the minimum set of failing pre-conditions and invariants could also be a potential area of future work for the Flint verifier.

Detecting inconsistent assumptions is a unique feature offered by the Flint verifier, providing a quick and useful way to validate specifications.

6.4.2 (Un)Reachable code

Following our discussion in section 4.5, the Flint compiler can detect instances of unreachable, or always reachable, code. In particular, we are able to detect when an if-condition is always true or false. This functionality is also offered by the Solidity compiler. Smart contract authors typically do not intend on adding unreachable code, this functionality detects this unintentional bug [14]. Here we evaluate the performance, power and algorithm of the reachability functionality of Flint's verifier.

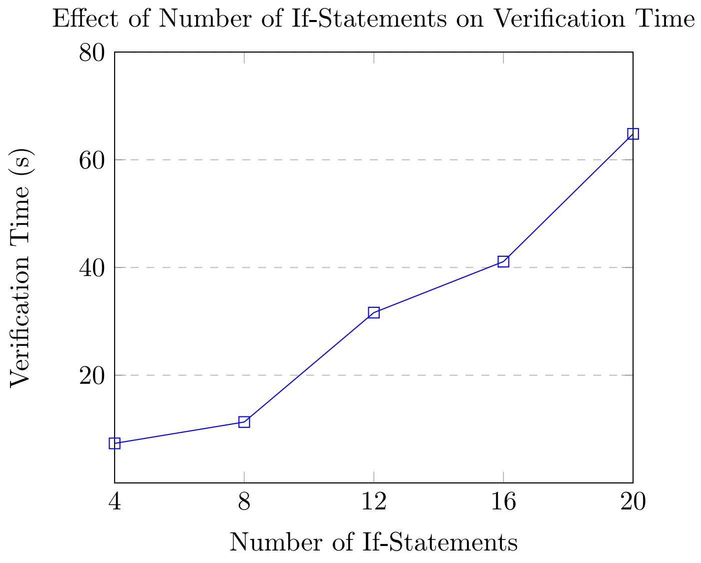
Like the Solidity compiler, whether the Flint verifier can determine the if-condition's value depends on the ability for Z3 to prove the condition. Our formulation of the if-condition expression, follows from our translation scheme in chapter 3. Because the Flint compiler is able to verify a larger subset of smart contract constructs, for example structs, it is able to show (un)reachability for a larger amount of if-conditions. Below we give simple examples of reachable and unreachable code that both the Flint compiler and Solidity compiler can prove.

[14] This type of bug has an associated CWE classification: https://cwe.mitre.org/data/definitions/670.html

```
if (3 < 1) {
    // .. never be executed
}
```

```
func equal(a: Int, b: Int)
    pre (a == b)
{
    if (a >= b) {
        // .. always be executed
    }
}
```

We also evaluate the performance of our implementation, to detect (un)reachable code. Our current implementation has a time complexity of order $\mathcal{O}(n)$, with respect to the number of if-statements. This approach leads to $2n$ invocations of the Boogie verifier, which increases verification time. We mitigate this by performing the n verifications in parallel, however this approach increases the load on the system. Below we outline affect of detecting (un)reachable code on the verifier. It is possible to further optimise our approach and reduce the lower-bound on the number of Boogie invocations required, we discuss this in further work section 7.1.4.



As we illustrate above, as the number of if-statements increases, so does the time to detect (un)reachable code. However, we would argue that the functionality and guarantee that the author's smart contract does not contain any (un)reachable code is worth the time.

We note however that the guarantee given by the Flint verifier is that any (un)reachable code which is detected has been shown to be (un)reachable. However, code which has not been shown to be (un)reachable is not guaranteed to not be (un)reachable. This is a result of Z3 being a sound but incomplete theorem prover [27].

6.4.3 Array Out-Of-Bounds

Following our discussion in section 4.2, the Flint compiler is able to detect array out of bounds accesses. Here we discuss the formal guarantees of the Solidity and Flint implementations of this feature as well as the feature's utility to a smart contract author.

Due to Z3 being sound, but incomplete, [27] any array accesses which do not raise an out-of-bounds warning have a corresponding proof, generated by Z3, to show that this is always the case. However, if an out-of-bounds error is raised then this does not guarantee that the array access is definitely out-of-bounds, only that the verifier cannot show that it is in-bounds. There may be cases which require complex proofs, to show that the array access is in-bounds, and Z3 is not strong enough. The output below gives an example of the output generated by the compiler, when an out-of-bounds error is detected.

```
Error in outOfBoundsContract.flint:
Potential out-of-bounds error: Could not verify that array access is
within array bounds at line 76, column 29:
    let k: Int = j.ss[i].js[1]
```

```
                                      ^^^^^
```
Listing 6.24: Flint Out-Of-Bounds Error message

The Solidity compiler, which also offers this functionality, also uses the Z3 theorem prover [10] and same problem formulation, therefore has the same formal guarantees as the Flint compiler.

This feature detects a specific case of immutability bugs, code SWC-110 by the Smart Contract Weakness Classification Registry[15]. This feature could be implemented by hand, using explicit assertions before array accesses, by the contract author. Instead the Flint compiler feature removes this burden from the contract author, thus guaranteeing complete coverage, and reduces the complexity of final contract code.

6.4.4 Divide By Zero

Following our discussion in section 4.3, the Flint compiler is able to detect a division by zero. Here we discuss the formal guarantees of the Solidity and Flint implementations of this feature as well as the feature's utility to a smart contract author.

Due to Z3 being sound, but incomplete, [27] any divisions which do not raise a division-by-zero warning have a corresponding proof, generated by Z3, to show that this is always the case. However, if a division-by-zero error is raised then this does not guarantee that a division by zero will happen, only that the verifier cannot show that it definitely will not. There may be cases which require complex proofs to show this and Z3 is not strong enough. The output below gives an example of the output generated by the compiler, when a division-by-zero error is detected.

```
Error in Arithmetic.flint:
Potential divide-by-zero error: Could not verify that denominator is
not zero at line 68, column 24:
    let i: Int = 100 / j
                       ^

Contract specification not verified
Failed to compile.
```
Listing 6.25: Flint division by zero verification error

The Solidity compiler, which also offers this functionality, also uses the Z3 theorem prover [10] and same problem formulation, therefore has the same formal guarantees as the Flint compiler.

This feature could be implemented by hand, using explicit assertions before the denominator, by the contract author. Instead this compiler feature removes this burden from the contract author, thus guaranteeing complete coverage, and reduces the complexity of final contract code.

6.4.5 Remarks

The Flint verifier offers a variety of functionality to prevent dangerous patterns or behaviour in Flint smart contracts. Currently, the verifier can detect out-of-bounds array accesses, division-by-zero, (un)reachable code, and inconsistent assumptions. However there is scope to use the Boogie representation of Flint smart contracts and the verifier to detect other dangerous patterns and prevent bugs. The Flint language has been in development since January 2018, as such the number of smart contracts written in it is small. We are not aware of any Flint smart contracts currently deployed on the Ethereum blockchain. Therefore our test suite of example Flint smart contracts are limited to the Flint translations of common 'buggy' Solidity smart contracts. To continue developing useful verification features we must first see how Flint contracts are used in the wild, and analyse the anti-patterns that authors are using. Until this point, we can only anticipate and mitigate what we believe are the dangerous ways to write a Flint smart contract.

6.5 Performance of Verification

Here we comment on the performance of the Flint verifier. Adding a verification system to the Flint compiler has impacted Flint compilation time, however due to the nature of compilation, we

[15] https://smartcontractsecurity.github.io/SWC-registry/

only consider performance up to a threshold. We evaluate whether the performance of the compiler has been affected so significantly, that a Flint user would be incentivised to disable the verification system. We compare compilation time with, and without, the verification system enabled.

Contract Name[16]	Lines	Verifier Enabled (s)			Verifier Disabled (s)
		Total[17]	Translation	Verification	
FlintDAO.flint	147	3.12	0.25	2.37	1.12
SimpleDAO.flint	145	4.14	0.26	1.43	1.14
Bank.flint	111	2.19	0.17	1.29	0.66
SimpleAuction.flint	77	2.52	0.11	1.48	0.82
Wallet.flint	39	5.92	0.09	3.45	0.55
RockPapersScissors.flint	33	3.93	0.08	3.58	0.49
Factorial.flint	23	5.44	0.08	4.55	0.44

Table 6.8: Flint compilation time, with verifier enabled and disabled

Table 6.8 shows the compilation time of 7 case study smart contracts, with the verifier enabled and disabled. We only compared the verification of functional specifications, we looked at the performance of verifying holistic specifications earlier in this section (see section 6.3). We can see that smart contract verification has increased compilation time by a factor of approximately 5.2 times. Furthermore, we can see that the Flint to Boogie translation time, is insignificant in comparison to the verification time.

The compilation time, with verification enabled, did result in a significant relative increase. However, the resulting compilation time is acceptable, in absolute terms, and does not detract from the user experience. A compilation time of 5 seconds would not incentivise a user to disable Flint's verification system. The automatic verification that the Flint compiler offers is worth the increased compilation time.

6.6 Comments on the verification technology stack

In this section, we discuss the benefits and limitations of the technology stack being used to verify Flint smart contracts.

The verification of functional specifications is powered by the Boogie verifier, and the verification of holistic specifications is powered by Symbooglix. Both of these tools use the .NET framework, which is supported on Windows, Mac OS and Linux. On Mac and Linux the Mono[18] run-time is needed to execute the .NET framework. The addition of these technologies increases the size of the Flint executable and increases the complexity of the installation. We argue that the verification features powered by this stack are worth the installation, configuration and space overhead.

Flint's verifier benefited from Boogie's proving power, and it's supporting ecosystem. Symbolic execution of smart contracts, for verifying holistic specifications, would have required us to implement our own symbolic execution engine, if we did not use Boogie, instead we could use Symbooglix. We acknowledge that installing the extra dependencies has an impact on the overall Flint installation user experience, however the technology stack clearly enabled the core functionality of the Flint verifier.

6.7 Remarks

We have seen that the Flint compiler now supports automatic formal verification of user-supplied functional and holistic specifications. Furthermore, the Flint compiler also offers other functionality, powered by the verifier, which are used to mitigate against out-of-bounds, division-by-zero,

[16]Taken from Flint case studies: https://github.com/YianniG/flint/tree/master/examples/casestudies
[17]Total compilation time, with verifier enabled
[18]https://www.mono-project.com/

inconsistent assumptions and unreachable code errors. These functions are enabled by default, the Flint compiler will automatically detect potential immutability bugs, even if the contract author has not provided a specification.

Our translation scheme is good enough for the Flint compiler to return meaningful verification results. We have also seen that the verifier supports enough of Flint's language constructs, and Flint's specification language is expressive enough, for the verifier to be used to verify production-ready or deployed contracts.

Finally, we have seen that while the Flint compiler does not support complex holistic specifications, such as those supported by VerX, it is able to verify specifications using the `will` predicate, for a given transaction bound.

Concluding our evaluation of our contributions to the Flint compiler, we acknowledge that the Flint language has not been officially released and is still in development. We are not aware of any Flint smart contracts, by authors external to Flint development, deployed on the Ethereum blockchain. By not having Flint smart contracts written by external authors, we cannot fully understand the dangerous patterns that Flint smart contracts suffer, and therefore cannot design techniques to prevent and warn authors of them. Without more Flint smart contracts, and authors, we are limited in our ability to evaluate whether our contributions are effective at preventing immutability bugs.

Chapter 7

Conclusion

The code of a smart contract cannot be changed once it is deployed, the author must be certain of the contract's behaviour. Currently, there are very few tools available for automatic formal verification of smart contracts, and formal verification is the only method which provides the strong guarantees that the author requires. We hope that our contributions to the Flint compiler represent a positive contribution to the area of automatic formal verification of smart contracts.

Many groups are interested in formal verification of smart contracts. It has the ability to revolutionise the space by allaying security fears, a common concern with smart contracts due to their handling of money. The Ethereum Foundation, ChainSecurity and others are currently working on automatic verification, developing tools and approaches to make smart contracts safer. It is a rapidly developing area, with lots of research and interest.

Our novel approach, translating Flint to Boogie to verify contracts, enabled many features which would not have otherwise been feasible for this project. The Boogie verifier allowed us to implement a functional verification system within the Flint compiler with the proving power of the Z3 solver, without having to translate Flint code all the way down into low level logical propositions. Furthermore, the tooling as part of the Boogie ecosystem enabled the addition of verification of holistic specifications into Flint with a symbolic execution engine, Symbooglix, which consumed Boogie code. As Symbooglix and Boogie offered powerful verification functionality, we were able to focus on extending and translating Flint syntax, in a way which would provide meaningful verification results. The combination of our translation scheme and harnessing the proving power of the Boogie ecosystem allowed us to implement verification of user-supplied functional and holistic specifications into the Flint compiler, but also check for out-of-bounds, division-by-zero, and unreachability errors.

In our evaluation we have shown that the specification language is expressive enough for smart contract authors to express the intended behaviour of their smart contracts, and the verifier powerful enough for the contract to be verified at compile-time. We hope that the progress made in adding formal verification to Flint, makes formal verification more accessible and encourages its use in smart contracts.

7.1 Future work

Deciding the verification approach, and designing the translation scheme from Boogie to Flint, required significant investigative work. To conclude, in this section, we present areas of work in which we were unable to develop further, due to time constraints.

7.1.1 Formalise Flint's semantics

As discussed in section 6.1, we only present informal arguments of correctness, that our Boogie translation scheme is semantic preserving. Therefore we must formalise Flint's semantics, so that we can construct a proof of correctness. Work must also be done to verify that the compiler conforms to the formal semantics of Flint.

7.1.2 Flint Translation to Boogie

Here we outline further work which could be done to improve our translation scheme.

Boolean Operator Short-Circuiting

Currently we do not translate the boolean operators `&&` and `||` in a semantic preserving way. This is because we do not first evaluate the left-hand-side operator, and then the right-hand-side, instead we evaluate the boolean expression all at once. Instead, we should check each operand separately. The listings below, give an example of an approach using an if-statement and temporary variable to do this.

```
func f()
{
    let b1: Bool = false

    // booleanFunction() -> Bool
    // should not be called
    let b2: Bool = b1 &&
    booleanFunction()
}
```

Listing 7.1: Flint example using short-circuting

```
procedure f()
{
    var b1, b2, temp: bool;
    b1 := false;
    temp := false;

    if (b1) {
        call temp := booleanFunction();
    }

    b2 := temp;
}
```

Listing 7.2: Example Boogie translation of operator short-circuiting

Logical Representation of Array and Dictionary Literals

Currently we cannot express array or dictionary literals in Flint pre and post-conditions, as we do not have a logical representation of array and dictionary literals. Here we propose a translation to Boogie which could be implemented to further extend Flint's specification syntax. We propose representing array and dictionary literals by using Boogie functions, which return arrays that are equivalent to the array or dictionary literal. More specifically the function `function arraySize1(a: int) returns (r: [int]int)` would return an array where `r[0] == a`. With this method, we must generate a function to generate each size of literal, until the max literal size. We then replace every occurrence of an array or dictionary literal, with a call to the Boogie function. An example of this is given in the listings below.

```
func f(arg: [Int])
    pre (arg == [0, 1, 2])
{
    // ... body
}
```

Listing 7.3: Example array literal in pre-condition

```
function array3(a: int, b: int, c: int)
    returns (result: [int]int);

axiom (∀ a, b, c: int •
       array3(a, b, c)[0] = a
    ∧ array3(a, b, c)[1] = b
    ∧ array3(a, b, c)[2] = c);

procedure f(arg: [int]int)
    requires (arg = array3(1, 2, 3));
{
    // ... body
}
```

Listing 7.4: Example Boogie translation of array literals for pre-conditions

7.1.3 Functional Specifications

Here we briefly outline some predicates which would increase the expressiveness of Flint's functional specifications.

- **Balance operator** Would allow the user to assert properties about the current balance of the contract.
- **Type State** Would allow the user to assert properties about the current type state of the contract.

- **Assumptions** To allow for assumptions to be placed in the bodies of Flint code, to help the verifier prove complex properties. These assumptions would be tested at run-time, to guarantee that the assumption always holds.

7.1.4 Verification Features

Detecting Inconsistent Assumptions

Currently we detect if a function has inconsistent assumptions, and return a warning to the user. It would be more informative to return the minimum failing subset of pre-conditions and invariants, so the user knows precisely which pre-conditions and invariants are responsible.

If any contract invariants are inconsistent, every function is detected as having inconsistent assumptions which returns a large amount of output and provides an unpleasant user experience. It would be much better to detect if the contract has any inconsistent invariants separately. We propose introducing an extra phase to the verification pipeline, which tests invariants for inconsistent assumptions first, using the technique above to find the minimal failing subset, and then test for inconsistent assumptions within functions.

(Un)Reachable Code

Currently we invoke the Boogie verifier $2n$ times, where n is the number of if-statements, to test whether each if-statement's condition can be shown to be true or false. This is required to avoid the verification of one if-condition affecting another. This cannot happen if the if-conditions are within different Boogie procedures. Therefore it is possible to reduce the number of verification invocations by packing if-condition tests, within different procedures, within the same Boogie file. Allowing the testing of multiple if-conditions within one Boogie invocation. This approach reduces the lower bound of number of invocations of the Boogie verifier, for example only one if-statement in each Flint function. However, it does not reduce the worst-case number of invocations, for example when all the if-statements are within one Flint function.

```
func f()
{
  if (true) {
    // always reachable
  }
}

func g(a: Int)
  pre (a > 0)
{
  if (a == 0) {
    // unreachable
  }
}
```

Listing 7.5: Flint example of unreachable code

```
// Can test both of the if-statements in
// the same invocation of the Boogie
// verifier
procedure f()
{
  assert (true);
}

procedure g(a: Int)
  requires (a > 0)
{
  assert (a = 0);
}
```

Listing 7.6: Example Boogie packing unreachability tests

Counter-Example Generation

Generating counter-examples for failing proof-obligations is a very useful feature to debug code and specifications, as it provides more insight into the verification of the contract. It is possible for Boogie to emit it's internal model, on a verification failure, which could be processed to generate counter examples. Otherwise it may be possible to access the Z3 theorem prover directly, and access the counter examples it generates.

7.1.5 Holistic Specifications

Expressiveness

The current holistic specification language has limited predicates and expressiveness. We propose the addition of other holistic operators, such as `was` which allows the assertion of properties over previous transactions. Similarly, the addition of a `FUNCTION` or `BALANCE` predicate would allow the user to assert properties about the current function call or the balance of the contract. Furthermore, the expressiveness of Flint's holistic specifications could be increased by expanding the syntax of holistic specifications, so that predicates can be combined with binary operators.

Abstract Interpretation

We look to see if we can provide a stronger guarantee for the verification of Flint's holistic specifications, namely whether we can prove that the holistic spec holds for all contracts states for all transactions. Program abstraction could be a method for doing this. Program abstraction can be used to transform unbounded models into bounded models, via a sound semantic approximation. Transforming a smart contract's unbounded state space into a bounded model, allows symbolic execution to be used to verify if the specification holds. The challenge with using abstract interpretation is deciding on the abstract domain to use. This depends on the specifications that the user wishes to verify. This will require significant exploratory work to determine the abstract domain to use.

7.1.6 Specification Validation

This project has mainly focused on adding verification methods to the Flint compiler. Other than detecting inconsistent assumptions, we do not address specification validation, that is to say, a verification result is only as good at the specification which it is verified against. This is not the problem we attempted to solve here, but is another crucial area which would benefit formal smart contract verification.

Appendix A

Supporting Code Listings

```
func g() {
  if (!A) { // require (A)
    fatalError()
  }

  // A holds here
}
```

Listing A.1: Example Flint equivalent of `require(A)`

```
contract C {
  var i: Int = 5
  var b: Bool = false

  will (i == 10)
}

C :: (any) {
  public init() {}

  public func add()
    mutates (i)
  {
    self.i += 1
  }

  public func invert()
    mutates (b)
  {
    self.b = b == false
  }
}
```

Listing A.2: Example Flint contract with holistic specification

```
// Bubble Sort, where the specification says the output is a permutation of
// the input.

// Introduce a constant 'N' and postulate that it is non-negative
const N: int;
axiom 0 ≤ N;

// Declare a map from integers to integers.  In the procedure below, 'a' will be
// treated as an array of 'N' elements, indexed from 0 to less than 'N'.
var a: [int]int;

// This procedure implements Bubble Sort.  One of the postconditions says that,
// in the final state of the procedure, the array is sorted.  The other
// postconditions say that the final array is a permutation of the initial
// array.  To write that part of the specification, the procedure returns that
// permutation mapping.  That is, out-parameter 'perm' injectively maps the
// numbers [0..N) to [0..N), as stated by the second and third postconditions.
// The final postcondition says that 'perm' describes how the elements in
// 'a' moved:  what is now at index 'i' used to be at index 'perm[i]'.
// Note, the specification says nothing about the elements of 'a' outside the
// range [0..N).  Moreover, Boogie does not prove that the program will terminate.

procedure BubbleSort() returns (perm: [int]int)
  modifies a;
  // array is sorted
  ensures (∀ i, j: int • 0 ≤ i ∧ i ≤ j ∧ j < N ⟹ a[i] ≤ a[j]);
  // perm is a permutation
  ensures (∀ i: int • 0 ≤ i ∧ i < N ⟹ 0 ≤ perm[i] ∧ perm[i] < N);
  ensures (∀ i, j: int • 0 ≤ i ∧ i < j ∧ j < N ⟹ perm[i] ≠ perm[j]);
  // the final array is that permutation of the input array
  ensures (∀ i: int • 0 ≤ i ∧ i < N ⟹ a[i] = old(a)[perm[i]]);
{
  var n, p, tmp: int;

  n := 0;
  while (n < N)
    invariant n ≤ N;
    invariant (∀ i: int • 0 ≤ i ∧ i < n ⟹ perm[i] = i);
  {
    perm[n] := n;
    n := n + 1;
  }

  while (true)
    invariant 0 ≤ n ∧ n ≤ N;
    // array is sorted from n onwards
    invariant (∀ i, k: int • n ≤ i ∧ i < N ∧ 0 ≤ k ∧ k < i ⟹ a[k] ≤ a[i]);
    // perm is a permutation
    invariant (∀ i: int • 0 ≤ i ∧ i < N ⟹ 0 ≤ perm[i] ∧ perm[i] < N);
    invariant (∀ i, j: int • 0 ≤ i ∧ i < j ∧ j < N ⟹ perm[i] ≠ perm[j]);
    // the current array is that permutation of the input array
    invariant (∀ i: int • 0 ≤ i ∧ i < N ⟹ a[i] = old(a)[perm[i]]);
  {
    n := n − 1;
    if (n < 0) {
      break;
    }

    p := 0;
    while (p < n)
      invariant p ≤ n;
      // array is sorted from n+1 onwards
      invariant (∀ i, k: int • n+1 ≤ i ∧ i < N ∧ 0 ≤ k ∧ k < i ⟹ a[k] ≤ a[i]);
      // perm is a permutation
      invariant (∀ i: int • 0 ≤ i ∧ i < N ⟹ 0 ≤ perm[i] ∧ perm[i] < N);
      invariant (∀ i, j: int • 0 ≤ i ∧ i < j ∧ j < N ⟹ perm[i] ≠ perm[j]);
      // the current array is that permutation of the input array
      invariant (∀ i: int • 0 ≤ i ∧ i < N ⟹ a[i] = old(a)[perm[i]]);
      // a[p] is at least as large as any of the first p elements
      invariant (∀ k: int • 0 ≤ k ∧ k < p ⟹ a[k] ≤ a[p]);
    {
      if (a[p+1] < a[p]) {
        tmp := a[p];  a[p] := a[p+1];  a[p+1] := tmp;
        tmp := perm[p];  perm[p] := perm[p+1];  perm[p+1] := tmp;
      }

      p := p + 1;
    }
  }
}
```

Listing A.3: Example Bubble Sort program in Boogie[1]

[1]Taken from: https://rise4fun.com/Boogie/Bubble

```
// Any currency should implement this trait to be able to use the currency
// fully. The default implementations should be left intact, only
// `getRawValue` and `setRawValue` need to be implemented.

struct trait Asset {
  // Initialises the asset "unsafely", i.e. from `amount` given as an integer.
  init(unsafeRawValue: Int)

  // Initialises the asset by transferring `amount` from an existing asset.
  // Should check if `source` has sufficient funds, and cause a fatal error
  // if not.
  init(source: inout Self, amount: Int)

  // Initialises the asset by transferring all funds from `source`.
  // `source` should be left empty.
  init(source: inout Self)

  // Moves `amount` from `source` into `this` asset.
  func transfer(source: inout Self, amount: Int) {
    if source.getRawValue() < amount {
      fatalError()
    }

    // TODO: support let _: Int = ...
    let unused1: Int = source.setRawValue(value: source.getRawValue() - amount)
    let unused2: Int = setRawValue(value: getRawValue() + amount)
  }

  func transfer(source: inout Self) {
    transfer(source: &source, amount: source.getRawValue())
  }

  // Returns the funds contained in this asset, as an integer.
  func setRawValue(value: Int) -> Int

  // Returns the funds contained in this asset, as an integer.
  func getRawValue() -> Int
}
```

Listing A.4: Flint Asset trait[2]

[2]Taken from https://github.com/YianniG/flint/blob/master/stdlib/Asset.flint

Appendix B

Flint Functional Specification Examples

```
//VERIFY-CHECK
contract C {
}

C :: (any) {
  public init() {}

  public func prime(a: Int, b: Int)
    pre (1 < a && a < 500)
    pre (1 < b && b < 25000)
  {
    assert (a * b != 23447)
  }


  public func factor(a: Int, b: Int)
    pre (1 < a && a < 500)
    pre (1 < b && b < 25000)
  {
    //VERIFY-FAIL
    assert (a * b != 23449)
  }

  func power()
  {
    assert (8 == 2 ** 3)
    //VERIFY-FAIL
    assert (8 == 2 ** 2)
  }

  func overflow1()
  {
    assert ((2 ** 255) &* 2 == 0)
    //VERIFY-FAIL
    assert ((2 ** 255) &* 3 == 0)
  }

  func overflow2()
  {
    assert (((2 ** 255) &+ (2**255)) == 0)
    assert ((0 &- (2 ** 255)) == 2 ** 255)
    assert ((2 ** 255) &* 3 > 0)
    assert ((2 ** 255) * 2 == (2 ** 256))
    assert ((2 ** 255) &* 2 == 0)
    //VERIFY-FAIL
    assert ((2 ** 255) &* 2 == 1)
  }
}
```

Listing B.1: Flint's Verifier Arithmetic Support[1]

```
//VERIFY-CHECK
contract Array {
  var As: [Int]
  var bs: [Int: Int]

  var cs: [[Int]]
  var ds: [[[Int]]]// = []
  var es: [Int: [[Int]]]// = [:]
}

Array :: (any) {
  public init() {
    As = []
    bs = [:]
    cs = []
    ds = []
    es = [:]
  }

  func arrayLiteral()
    mutates (As, bs, cs)
  {
    As[0] = 0
    bs[10] = 5
    cs[0] = []
    cs[0][0] = 0

    //let b: [[Int]] = []
  }

  func arrayDictSize() {
    let asSize: Int = As.size
    let bsSize: Int = bs.size
  }

  func dictKeys() {
    let bsKeys: [Int] = bs.keys

    let esKeys: [Int] = es.keys
    if esKeys.size > 0 {
      let esSubSize: Int = es[esKeys[0]].size
    }
  }

  func growSize(a: Address) {
    var bs: [Int] = [1,2,3,4]
    let bs_size_before: Int = bs.size
    bs[4] = 5
    assert(bs_size_before + 1 == bs.size)

    var As: [Address: Int] = [:]
    let as_size_before: Int = As.size
    As[a] = 5
    assert(as_size_before + 1 == As.size)
  }
}
```

Listing B.2: Flint's Verifier Array and Dictionary Support[2]

```
//VERIFY-CHECK
contract VerifyPrePostConditions {
  var ten: Int = 10
  var t: Int = 0

  invariant (10 == 10)
  invariant (ten == 10)
}

VerifyPrePostConditions :: (any) {
```

[1]Taken from https://github.com/YianniG/flint/blob/master/Tests/VerifierTests/tests/verifyArithmetic.flint

[2]Taken from https://github.com/YianniG/flint/blob/master/Tests/VerifierTests/tests/verifyArrayDictionaries.flint

```
  public init() { }

  func testPreConditions()
    pre (1 == 1)
  {
    assert (4 > 2)
  }

  func testPostConditions(i: Int)
    mutates (t, ten)

    post(t == 3)
  {
    self.ten -= i
    self.t = 3
    self.ten += i
  }

  func testPrePostConditions(i: Int) -> Int
    pre (i == 2)
    post (returns 3)
  {
    return i + 1
  }

  func testParamsPreConditions(input: Int, increment: Int)
    pre (input == 10)
  {
    let input2: Int = input + increment
    assert (input2 == 10 + increment)
  }

  func failPreCondition()
  {
    testParamsPreConditions(input: 10, increment: 5)

    //VERIFY-FAIL
    testParamsPreConditions(input: 0, increment: 0)
  }

  //VERIFY-FAIL
  func failInvariant()
    mutates(ten)
  {
    ten = 5
  }

  //VERIFY-FAIL
  func failPost(i: Int)
    mutates(t)
    post (t == 4)
  {
    self.t = i
  }
}
```

Listing B.3: Flint's Verifier Pre and Post-Condition Support[3]

```
//VERIFY-CHECK
external trait Library {
  public func getValue() -> int256
  public func setValue(amount: int256) -> int256
}

contract C {
  var v: Int = 0
  var j: Int = 10
  invariant (j == 10)
}

C :: (any) {
```

[3]Taken from https://github.com/YianniG/flint/blob/master/Tests/VerifierTests/tests/verifyPrePostConditions.flint

```
  public init() {}

  //VERIFY-FAIL
  public func failingInvariant(extAddress: Address)
    mutates (v, j)
  {
    let libInstance: Library = Library(address: extAddress)
    do {
      self.j = (call libInstance.getValue()) as! Int
    } catch is ExternalCallError {
      fatalError()
    }
  }

  func globalVariablesCanChange(extAddress: Address)
    mutates (v, j)
  {
    let libInstance: Library = Library(address: extAddress)
    let oldV: Int = self.v
    do {
      let r:Int = (call libInstance.getValue()) as! Int
    } catch is ExternalCallError {
      fatalError()
    }

    //VERIFY-FAIL
    assert (oldV == self.v)
  }

  func violatedInvariant(extAddress: Address)
    mutates (j, v)
  {
    self.j = 5
    let libInstance: Library = Library(address: extAddress)
    do {
      //VERIFY-FAIL
      (call libInstance.getValue()) as! Int
    } catch is ExternalCallError {
      fatalError()
    }
  }

  @payable
  func violatedAssetAccountingInvariant(implicit w: Wei, extAddress: Address)
    mutates (Wei.rawValue, j, v)
  {
    let libInstance: Library = Library(address: extAddress)
    var w1: Wei = w
    w1 = Wei(0)
    do {
      //VERIFY-FAIL
      (call libInstance.getValue()) as! Int
    } catch is ExternalCallError {
      fatalError()
    }
  }

  func indirectExternalCall(extAddress: Address)
    mutates (S.v, v, j)
  {
    let s: S = S()
    let sVOld: Int = s.v
    s.extCall(extAddress: extAddress)
    //VERIFY-FAIL
    assert (s.v == sVOld)
  }
}

struct S {
  var v: Int = 0

  invariant (v >= 0)

  public init() { }

  func extCall(extAddress: Address)
    mutates (v)
  {
    let libInstance: Library = Library(address: extAddress)
    do {
      (call libInstance.getValue()) as! Int
    } catch is ExternalCallError {
      fatalError()
    }
  }

  func testStructCouldChange(extAddress: Address)
    mutates (v)
  {
    let libInstance: Library = Library(address: extAddress)
    do {
      let vOld: Int = v
      (call libInstance.getValue()) as! Int
      assert (v >= 0)

      //VERIFY-FAIL
      assert (self.v == vOld)
    } catch is ExternalCallError {
      fatalError()
    }
  }
}
```

Listing B.4: Flint Verifier External Call Support[4]

```
//VERIFY-CHECK
contract C {
  var x: Int = 0

  var a: Int = 0
  invariant (a >= 0)
}

C :: (any) {
  public init() {}

  func f()
    mutates (x)
    pre (x < 10000)
  {
    x = x + 1
  }

  func g(b: Bool)
    mutates (x)
  {
    x = 0
    if (b) {
      f()
    }
    //VERIFY-FAIL
    assert(x == 0)
  }

  func h(x: Int) -> Int {
    return k(x: x)
  }

  func k(x: Int) -> Int {
    return x
  }

  func m() {
    let x: Int = h(x: 2);
```

[4]Taken from https://github.com/YianniG/flint/blob/master/Tests/VerifierTests/tests/verifyExternalCalls.flint

```
    assert(x > 0)
  }


  func l()
    mutates (a)
  {
    if (a > 0) {
      a = a - 1
      l()
    } else {
      assert(a == 0)
    }
  }
}
```

Listing B.5: Flint's Verifier Function Support[5]

```
//VERIFY-CHECK
struct L {
  public init() {}

  public func loopTest() {
    var js: [Int] = [1,2,3]
    for let j: Int in js {
      test(i: j)
      assert(j <= 3)
    }

    for let j: Int in (0...5) {
      test(i: j)
    }

    var bound: Int = 10
    for let k: Int in (0..<10) {
      test(i: k)
      assert (k < bound)
    }
  }

  func assignmentLoop() {
    var i: Int = 0
    for let k: Int in (0..<10) {
      i = 3
    }

    assert (i > 0)
  }

  func literalLoop() {
    for let j: Int in [1,2,3,4] {
      test(i: j)
      assert(j<5)
    }
  }

  func dictLoop() {
    var d: [Int: Address] = [:]
    for let v: Address in d {
      test(i: 1, a: v)
    }
  }

  func dictLoop2() {
    var d: [Int: Int] = [:]
    d[2] = 4
    d[4] = 6
    for let v: Int in d {
      assert(d[2] == v || d[4] == v)
    }
  }
```

[5]Taken from https://github.com/YianniG/flint/blob/master/Tests/VerifierTests/tests/verifyFunctions.flint

```

  func loopSum() -> Int
    post (returns 15)
  {
    var total: Int = 0
    for let t: Int in (0...5) {
      total += t
    }
    return total
  }

  func test() {
    var js: [Int] = [1,2,3]
    assert (js.size == 3)

    var ks: [Int: Int] = [:]
    assert (ks.size == 0)
  }

  func test(i: Int) { }

  func test(i:  Int, a: Address) { }
}

contract C {}

C :: (any) {
  public init() {}
}
```

Listing B.6: Flint Verifier Loop Support[6]

[6]Taken from https://github.com/YianniG/flint/blob/master/Tests/VerifierTests/tests/verifyLoops.flint

Appendix C

Solidity Functional Specification Examples

```
pragma experimental SMTChecker;

contract C
{
  uint x;
  function f() internal {
    require(x < 10000);
    x = x + 1;
  }
  function g(bool b) public {
    x = 0;
    if (b)
      f();
    // Should fail for `b == true`.
    assert(x == 0);
  }
  function h(bool b) public {
    x = 0;
    if (!b)
      f();
    // Should fail for `b == false`.
    assert(x == 0);
  }

}
// ----
// Warning: (209-223): Assertion violation happens here
// Warning: (321-335): Assertion violation happens here
```

Listing C.1: Solidity Verifier Support Function Call[1]

```
pragma experimental SMTChecker;
contract C
{
  function h(uint x) public pure returns (uint) {
    return k(x);
  }

  function k(uint x) public pure returns (uint) {
    return x;
  }
  function g() public pure {
```

[1]Taken from: https://github.com/ethereum/solidity/blob/develop/test/libsolidity/smtCheckerTests/functions/function_inside_branch_modify_state_var_3.sol

```
    uint x;
    x = h(2);
    assert(x > 0);
  }
}
```

Listing C.2: Solidity Verifier Function Inlining Support[2]

```
pragma experimental SMTChecker;
contract C {
    function f(uint x) public pure {
        uint y;
        for (y = 2; x < 10; ) {
            y = 3;
        }
        // False positive due to resetting y.
        assert(y < 4);
    }
}
// ----
// Warning: (213-226): Assertion violation happens here
// Warning: (142-147): Underflow (resulting value less than 0) happens here
// Warning: (142-147): Overflow (resulting value larger than 2**256 - 1)
    happens here
```

Listing C.3: Solidity Verifier For-Loop Support[3]

```
pragma experimental SMTChecker;

contract C
{
  bool b;
  function g(bool _b) internal returns (bool) {
    b = _b;
    return b;
  }
  function f() public {
    if (g(false) && (b == true)) {}
    if ((b == false) && g(true)) {}
    if (g(false) && g(true)) {}
    if (g(false) && (b == true)) {}
    if (g(true) && b) {}
  }
}
// ----
// Warning: (156-179): Condition is always false.
// Warning: (190-213): Condition is always true.
// Warning: (224-243): Condition is always false.
// Warning: (254-277): Condition is always false.
// Warning: (288-300): Condition is always true.
```

Listing C.4: Solidity Verifier Short-Circuiting Support[4]

```
pragma experimental SMTChecker;

contract C
{
  uint[] array;
```

[2]Taken from: https://github.com/ethereum/solidity/blob/develop/test/libsolidity/smtCheckerTests/functions/functions_identity_2.sol

[3]Taken from: https://github.com/ethereum/solidity/blob/develop/test/libsolidity/smtCheckerTests/loops/for_loop_6.sol

[4]Taken from: https://github.com/ethereum/solidity/blob/develop/test/libsolidity/smtCheckerTests/control_flow/short_circuit_and_touched_function.sol

```
  function f(uint x, uint p) public {
    require(x < 100);
    require(array[p] == 200);
    array[p] -= array[p] - x;
    assert(array[p] >= 0);
    assert(array[p] < 90);
  }
}
// ----
// Warning: (201-222): Assertion violation happens here
```

Listing C.5: Solidity Verifier Compound Operator Support[5]

[5]Taken from: https://github.com/ethereum/solidity/blob/develop/test/libsolidity/smtCheckerTests/operators/compound_sub_array_index.sol

Appendix D

Flint Functional Specification Case Studies

```
//VERIFY-CHECK
// Contract declarations contain only their state properties.
contract Bank {
  var manager: Address
  var balances: [Address: Wei]
  var accounts: [Address]
  var lastIndex: Int = 0

  invariant (lastIndex == accounts.size)

  var totalDonations: Wei

  event didCompleteTransfer (from: Address, to: Address, value: Int)
}

// The functions in this block can be called by any user.
Bank :: account <- (any) {
  public init(manager: Address)
  // Specify Wei.rawValue, as this state is not defined in current contract
    mutates(Wei.rawValue)
  {
    self.manager = manager
    balances = [:]
    accounts = []
    totalDonations = Wei(0)
  }

  // Returns the manager's address.
  public func register()
    mutates (accounts, lastIndex)

    post (lastIndex - 1 == prev(lastIndex))
    post (accounts.size == 1 + prev(accounts.size))
  {
    accounts[lastIndex] = account
    lastIndex += 1
  }

  public func getManager() -> Address
    post (returns manager)
  {
    return manager
  }

  @payable
  public func donate(implicit value: Wei)
    mutates (Wei.rawValue)

    post (totalDonations.rawValue == prev(totalDonations.rawValue) +
   prev(value.rawValue))
  {
    // This will transfer the funds into totalDonations.
    totalDonations.transfer(source: &value)
  }
}

// Only the manager can call these functions.
Bank :: (manager) {

  // This function needs to be declared "mutating" as its body mutates
  // the contract's state.
  public func freeDeposit(account: Address, amount: Int)
    mutates (Wei.rawValue)

    pre (dictContains(balances, account))
  {
    var i: Int = (0-1)
    var w: Wei = Wei(amount)
    balances[account].transfer(source: &w)
  }

  public func clear(account: Address)
    mutates (balances, Wei.rawValue)

    pre (dictContains(balances, account))
    post (balances[account].rawValue == 0)
  {
    // Wei could be being destroyed here
    //VERIFY-FAIL
    balances[account] = Wei(0)
  }

  // This function is non-mutating.
  public func getDonations() -> Int
    post (returns totalDonations.rawValue)
  {
    return totalDonations.getRawValue()
  }
}

// Any user in accounts can call these functions.
// The matching user's address is bound to the variable account.
Bank :: account <- (accounts) {
  public func getBalance() -> Int
    pre (dictContains(balances, account))
    post (returns balances[account].rawValue)
  {
    return balances[account].getRawValue()
  }

  public func transfer(amount: Int, destination: Address)
    mutates (balances, Wei.rawValue)

    pre(dictContains(balances, destination))
    pre(dictContains(balances, account))
    pre (destination != account)
    pre (amount > 0)
  {
    // Transfer Wei from one account to another. The balances of the
    // originator and the destination are updated atomically.
    // Crashes if balances[account] doesn't have enough Wei.
    balances[destination].transfer(source: &balances[account], amount:
   amount)

    // Emit the Ethereum event.
    emit didCompleteTransfer(from: account, to: destination, value: amount)
  }

  @payable
  public func deposit(implicit value: Wei)
    mutates(Wei.rawValue)

    pre(dictContains(balances, account))
  {
    balances[account].transfer(source: &value)
  }

  public func withdraw(amount: Int)
    mutates(balances, Wei.rawValue, manager, accounts, lastIndex,
   totalDonations)

    pre(dictContains(balances, account))
    pre(amount > 0)
  {
    // Transfer some Wei from balances[account] into a local variable.
    let w: Wei = Wei(&balances[account], amount)

    // Send the amount back to the Ethereum user.
    send(account, &w)
    assert(w.getRawValue() == 0)
  }
}
```

Listing D.1: Example functional specification for a Bank smart contract[1]

[1]Taken from https://github.com/YianniG/flint/blob/master/Tests/VerifierTests/tests/verifyBank.flint

Appendix E

Outline of Translations From Flint to Boogie

In this appendix, we simply present each Flint construct and it's translated Boogie representation, with no exposition. For further explanation refer to chapter 3.

Types

Name	Flint Type (in code)	Boogie Type (in code)
Int (256-bit integer)	`Int`	`int`
Address (160-bit)	`Address`	`int`
Boolean	`Bool`	`Bool`
String	`String`	`(int, [int]int)`
Struct	`Struct`	`int`

Dynamic Types

Name	Flint Type (in code)	Boogie Type (in code)
Dynamic-size list of type T	`[T]`	`[int]t`
Fixed-size list of type T and size n	`T[n]`	`[int]t`
Dictionary of keys K, and values V	`[K: V]`	`[k]v`

where t, k, v are the Boogie translations of Flint types T, K, V

Arrays

```
// Create array
let array: [Int] = [0,1,2]

// Append a new value
array[3] = 3

// Query the array size
let size: Int = array.size
```

Listing E.1: Flint access array size

```
// Create array
array[0] := 0;
array[1] := 1;
array[2] := 2;
size_array := 3;

// Append a new value
array[3] := 3;
size_array := size_array + 1;

// Query the array size
size := size_array;
```

Listing E.2: Access array size Boogie translation

```
// Append a new value
array[3] := 3;
if (3 == size_array) {
    // Index of 3 is equal to size of array, so we are appending to the array
    size_array := size_array + 1;
}

// Query the array size
size := size_array;
```

Listing E.3: Boogie array access to update size shadow variable

Dictionaries

```
let d: [Address: Int] = [0x1: 1]

// Assigning into the dictionary
d[0x2] = 2
```

Listing E.4: Flint assign into dictionary

```
d[0x1] = 1;

// Assigning into the dictionary
d[0x2] = 2;
tmp_counter := 0;
contains_value := false;
while (tmp_counter < size_d) {
    if (keys_d[tmp_counter] == 0x2) {
        contains_value := true;
    }
    tmp_counter := tmp_counter + 1;
}
if (contains_value) {
    keys_d[size_d] = 0x2;
    size_d := size_d + 1;
}
```

Listing E.5: Boogie translation of assigning into dictionary

Range Types

```
let start: Int = 3
let end: Int = 5
let sum: Int = 0
for let i: Int in (start...end) {
    // Loop body
    sum += i
}
```

Listing E.6: Flint for-loop with range declaration

```
start := 3;
end := 5;
sum := 0;

loop_counter := start;
while (loop_counter ≤ end) {
    i := loop_counter;

    // Loop body
    sum := sum + i;

    loop_counter := loop_counter + 1;
}
```

Listing E.7: Boogie translation or for-loop with range

Constants and variables

```
var v: Int
let l: Int
```

Listing E.8: Flint declarations

```
var v: int;
var l: int;
```

Listing E.9: Boogie declarations

Naming

```
contract C {
    // global1_C
    var global1: Int = 0
}

C :: (any) {
    // init_C
    public init() {
        // i_init_C
        let i: Int = 0
    }

    // functionInt_C
    func function(param1: Int) { //param1_functionInt_C
        // ...
    }
}
```

Listing E.10: Flint variable name translation

Functions

Signature

```
@payable
public func deposit(account: Address, amount: implicit Wei) -> Bool
    mutates (balances)
```

Listing E.11: Flint function signature

```
procedure deposit_AddressWei(account_deposit_AddressWei: Int,
                             amount_deposit_AddressWei: Int) returns (returnValue: bool)
    modifies balances;
```

Listing E.12: Boogie procedure signature

Body

```
public func transfer(source: Address, destination: Address) -> Bool
    mutates (balances)
{
    let sourceBalance = balances[source]
    balances[source] = 0

    let destinationBalance = balances[destination]
    destinationBalance += sourceBalance
    balances[destination] = destinationBalance

    return true
}
```

Listing E.13: Flint function body example

```
procedure transfer_AddressAddressInt(source_transfer_AddressAddressInt: Int,
    destination_transfer_AddressAddressInt: Int) returns (returnValue: bool)
    modifies balances;
{
    // Declaring all local variables
    var sourceBalance: int;
    var destinationBalance: int;

    sourceBalance := balances[source_transfer_AddressAddressInt];
    balances[source] = 0;
    // Omitting shadow variable consistency statements, for brevity

    destinationBalance := balances[destination_transfer_AddressAddressInt];
    destinationBalance := destinationBalance + sourceBalance;
    balances[destination_transfer_AddressAddressInt] := destinationBalance;
    // Omitting shadow variable consistency statements, for brevity

    // return
    returnValue := true;
}
```

Listing E.14: Boogie procedure body translation example

Contracts

Declaration

Global variables

```
contract Oven {
    var timer: Int
    var power: Int = 180
}

TrafficLight :: (any) {
    public init(seconds: Int) {
        self.timer = seconds
    }
}
```

Listing E.15: Flint contract with global variables

```
var timer_Oven: int;
var power_Oven: int;

procedure init_Oven(seconds: int)
    modifies time_Oven;
    modifies power_Oven;
{
    // Global variable initial value
    power_Oven := 180;

    timer_Oven := seconds;
}
```

Listing E.16: Boogie translation of global variables

Type States

```
// Red   == 0
// Amber == 1
// Green == 2
contract TrafficLight (Red, Amber, Green) {
    // Variable / event declarations
}
```

Listing E.17: Flint contract with type states

Protection blocks

Caller group

```
contract Lottery {}

Lottery :: (lucky) {
    func lucky(address: Address) -> Bool {
    // return true or false
    }

    public func withdraw() {
        // Withdraw winnings
    }
}
```

Listing E.18: Flint predicate function caller group example taken from [1]

```
var caller: int;

procedure lucky_Lottery(address: int) returns (returnValue: bool)
{
    // return true or false
}

procedure withdraw_Lottery()
    requires (lucky_Lottery(caller));
{
    // Withdraw winnings
}
```

Listing E.19: Ideally use Boogie procedure as pre-condition

```
var caller: int;

procedure withdraw_Lottery()
{
    var authorised: bool;
    call authorised := lucky_Lottery(caller);
    if (authorised) {
        // Authorised caller, continue as normal
    } else {
        assume (false);
    }

    // Withdraw winnings
}
```

Listing E.20: Boogie predicate function translation

0-ary functions

```
contract Bank {}

Bank :: (onCall) {
    func onCall() -> Address {
        // The person on call
    }

    public func freezeWithdrawals() {
        // Freeze account withdrawals
    }
}
```

Listing E.21: Flint 0-ary function example

```
var caller: int;

procedure onCall_Bank()
    returns (returnValue: int)
{
    // The person on call
}

procedure freezeWithdrawals_Bank()
{
    var authorised_address: int;
    call authorised_address :=
        freezeWithdrawals_Bank();
    if (authorised_address = caller) {
        // Authorised caller
        // continue as normal
    } else {
        assume (false);
    }

    // Freeze account withdrawals
}
```

Listing E.22: Boogie 0-ary function translation

State property (single address)

```
contract Bank {
  let owner: Address
}

Bank :: (owner) {
  public func clearDeposits() {
    // ...
  }
}
```

Listing E.23: Flint single address state property example inspired by [2]

```
var caller: int;
var owner_Bank: int;

procedure clearDepostits_Bank()
    requires (caller = owner_Bank);
{
    // ...
}
```

Listing E.24: Boogie single address state property translation

State property (array of addresses)

[1]https://docs.flintlang.org/docs/language_guide#protection-blocks
[2]https://docs.flintlang.org/docs/language_guide#protection-blocks

```
contract Bank {
    var managers: [Address]
}

Bank :: (managers) {
  public func payEmployees() {
    // ...
  }
}
```

Listing E.25: Flint array of addresses state property example inspired by [3]

```
var caller: int;
var managers_Bank: [int]int;

procedure payEmployees_Bank()
    requires (∃ i: int •
    managers_Bank[i] = caller);
{
    // ...
}
```

Listing E.26: Boogie array of addresses state property translation

State property (dictionary of addresses)

```
contract Bank {
    var customers: [String: Address]
}

Bank :: (customers) {
  public func withdraw() {
    // ...
  }
}
```

Listing E.27: Flint dictionary of addresses state property example inspired by [4]

```
var caller: int;
type String;
var customers_Bank: [String]int;

procedure withdraw_Bank()
    requires (∃ c: String •
    customers_Bank[c] = caller);
{
    // ...
}
```

Listing E.28: Boogie dictionary of addresses state property translation

Multiple caller groups

```
contract Bank {
    let manager: Address
    var accounts: [Address]
}
Bank :: (manager, accounts) {
  func forManagerOrCustomers() {}
}
```

Listing E.29: Flint multiple state property caller groups example taken from [5]

```
var caller: int;

var manager_Bank: int;
var accounts_Bank: [int]int;

procedure forManagerOrCustomers()
    requires (manager_Bank = caller
∨ ∃ i: int • accounts_Bank[i] = caller);
{
}
```

Listing E.30: Boogie multiple state property caller groups translation

[3]https://docs.flintlang.org/docs/language_guide#protection-blocks
[4]https://docs.flintlang.org/docs/language_guide#protection-blocks
[5]https://docs.flintlang.org/docs/language_guide#protection-blocks

```
contract Bank {
    let manager: Address
}

Bank :: (manager, onCall) {
    func onCall() -> Address{
        // The person on call
    }

    func resolveIncident() {
        // ...
    }
}
```

Listing E.31: Flint multiple caller groups example inspired by [6]

```
var caller: int;

var manager_Bank: int;

procedure onCall ()
    returns (returnValue: int)
{
    // The person on call
}

procedure resolveIncident()
{
    var authorised_address: int;
    call authorised_address := onCall();
    if (caller == manager ∨
    authorised_address)
    {
        // Address is authorised
        // continue execution as normal
    } else {
        assume (false);
    }

    // ...
}
```

Listing E.32: Boogie multiple caller groups translation

Caller group variable

```
contract AddressBook {
  var book: [Address: String] = [:]
}

AddressBook :: address <- (any) {
  public func remember(name: String)
  {
    book[address] = name
  }
}
```

Listing E.33: Flint caller group variable example inspired by [7]

```
var caller: int;
type String;
var book_AddressBook: [int]String;

procedure remember_AddressBook(name: String)
{
    var address: int;
    // Assign caller group variable
    address := caller;

    book_AddressBook[address] := name;
}
```

Listing E.34: Boogie caller group variable translation

Type state protection

```
contract Poll(Open, CountingVotes,
    Result) {
  // ...
}

Poll @(Open) :: (any) {
  public func voteFor(option:
    String) {
    // ...
  }
}
```

Listing E.35: Flint type state example taken from [8]

```
// Open = 0
// CountingVotes = 1
// Result = 2
var stateVariable_Pool: int;
type String;

procedure voteFor(option: String)
    requires (stateVariable == 0);
{
    // ...
}
```

Listing E.36: Boogie type state translation

[6] https://docs.flintlang.org/docs/language_guide#protection-blocks
[7] https://docs.flintlang.org/docs/language_guide#protection-blocks
[8] https://docs.flintlang.org/docs/language_guide#protection-blocks

Expressions

Binary Expression

```
var i: Int = 0
var j: Int = 2

i = j += 1
```

Listing E.37: Flint nested example

```
i := 0;
j := 2;

j := j + 1;
i := j;
```

Listing E.38: Boogie nested example translation

Function calls

```
// Call function and capture result
let result: Int = someFunction(10,
    0x0)

// Call function and ignore result
someFunction(10, 0x0)

// Call funcion, which returns
    nothing
someFunctionNoValue(10)
```

Listing E.39: Flint function call examples

```
var result, temp: int;

// Call function and capture result
call result := someFunctionReturns(10, 0)

// Call function and ignore result
call temp := someFunctionReturns(10, 0)

// Call funcion, which returns nothing
call someFunctionNoValue(10)
```

Listing E.40: Boogie translation of function calls

Dot access

```
contract Bank {
    var manager: Address;
}

func work() {
    send(self.manager, 100)
}
```

Listing E.41: Flint state property access

```
var manager_Bank: int;

procedure work_Bank() {
    call send(manager_Bank, 100);
}
```

Listing E.42: Boogie translation of state property access

```
let d: [Int: Int]
let a: [Int]

let dKeys: [Int] = d.keys
let aSize: Int = a.size
```

Listing E.43: Flint size/keys collection access

```
var d, a: [int]int;
// Shadow variables
var size_d, size_a: int;
var keys_d: [int]int;
// Remainder of declarations
var dKeys: [int]int;
var aSize: int;

dKeys := keys_d;
aSize := size_a;
```

Listing E.44: Boogie translation of size/keys collection access

Subscript Expression

```
let js: [[Int]] = [[0, 1]]

let j: Int = js[0][1]
```

Listing E.45: Flint subscript expression example

```
var js: [int]int;
var j: int;

js[0][0] := 0;
js[0][1] := 1;

j := js[0][1];
```

Listing E.46: Boogie subscript expression translation

Attempt

```
Bank :: (any) {
  func foo() {
    let result: Bool = try? bar()

    try! bar()
  }
}

Bank :: (manager) {
  func bar() {}
}
```

Listing E.47: Flint try expression example taken from [9]

```
var caller: int;

procedure foo_Bank()
{
    var result: bool;

    // try?
    // Test if function protections are satisfied
    if (caller == manager) {
        result := true;
        call bar_Bank();
    } else {
        result := false;
    }

    // Test if function protections are satisfied
    if (caller == manager) {
        call bar_Bank();
    } else {
        // Call fails
        assume (false);
    }
}

procedure bar_Bank()
    requires (caller == manager);
{
    // ...
}
```

Listing E.48: Boogie try expression translation

Literals

Basic types

Literal	Flint Representation	Boogie Representation	Explanation
Integer	10	10	Use the same representation
Address	0x10	16	Convert hex into base 10
Boolean	true	true	Use the same representation
String	-	-	Not supported

Dynamic types

```
//[0, 1, 2]
tmp_array_literal[0] == 0
tmp_array_literal[1] == 1
tmp_array_literal[2] == 2
size_tmp_array_literal == 3
```

Listing E.49: Array literal translation

```
//[0x0: 0, 0x10: 10, 0x20: 20]
tmp_dictionary_literal[0] == 0
tmp_dictionary_literal[16] == 10
tmp_dictionary_literal[32] == 20
size_tmp_dictionary_literal == 3
keys_tmp_dictionary_literal[0] == 0
keys_tmp_dictionary_literal[1] == 10
keys_tmp_dictionary_literal[2] == 20
```

Listing E.50: Dictionary literal translation

[9] https://docs.flintlang.org/docs/language_guide#dynamic-checking

Operators

Arithmetic Operators

Name	Flint	Boogie
Addition	+	+
Subtraction	−	−
Multiplication	∗	∗
Division	\	**div**
Modulo	%	**mod**
Exponentiation	∗∗	`power(n, e)`

Table E.1: Safe operators

Name	Flint	Boogie
Addition	`a &+ b`	`(a + b) mod power(2, 255)`
Subtraction	`b &- b`	`(a - b) mod power(2, 255)`
Multiplication	`a &* b`	`(a * b) mod power(2, 255)`

Table E.2: Unsafe operators

```
function power(n: int, e: int) returns (i: int);
axiom (∀ n: int • power(n, 0) = 1); // Base case

// Recursive cases
axiom (∀ n, e: int • (e mod 2 = 0 ∧ e > 0) ⟹
    (power(n, e) = power(n, e div 2)∗power(n, e div 2)));

axiom (∀ n, e: int • (e mod 2 = 1 ∧ e > 0) ⟹
    (power(n, e) = n∗power(n, (e−1) div 2)∗power(n, (e−1) div 2)));
```

Listing E.51: Boogie representation of the power operator

Boolean operators

Name	Flint	Boogie
Equal to	==	==
Not equal to	!=	!=
Logical or	||	||
Logical and	&&	&&
Less than	<	<
Less than or equal to	<=	<=
Greater than	>	>
Greater than or equal to	>=	>=

Table E.3: Boolean operators

Enumerations

```
enum CompassPoint: Int {
  case north = 1
  case south = 2
  case east = 3
  case west = 4
}
```

Listing E.52: Flint enumeration declaration taken from Flint documentation [10]

```
let direction: CompassPoint =
    CompassPoint.north
```

Listing E.53: Flint enumeration usage taken from Flint documentation [11]

```
type CompassPoint = int;
const unique north_CompassPoint:
    CompassPoint;
const unique south_CompassPoint:
    CompassPoint;
const unique east_CompassPoint:
    CompassPoint;
const unique west_CompassPoint:
    CompassPoint;
```

Listing E.54: Enumeration translation

```
var direction: CompassPoint;
direction := north_CompassPoint;
```

Listing E.55: Enumeration usage translation

[11] https://docs.flintlang.org/docs/language_guide#enumerations

Associated values

```
axiom (north_CompassPoint = 1);
axiom (south_CompassPoint = 2);
axiom (east_CompassPoint = 3);
axiom (west_CompassPoint = 4);
```

Listing E.56: Enumeration associated value

Statements

Assignment

```
let counter: Int = 10
```

```
var counter: int;
counter := 10;
```

Compound assignment

```
let i: Int = 10
let j: Int = (i += 10)

i = (i += 1) + (j  /= 2) + 3
```

Listing E.57: Flint compound assignment

```
var i: int;
var j: int;

// j = (i += 10)
i := 10;
i := i + 10;
j := i;

// i = (i += 1) + (j  /= 2) + 3
i := i + 1;
j := j div 2;
i := i + j + 3;
```

Listing E.58: Boogie compound assignment translation

Loops

```
let is: [Int] = [0, 1, 2]
let count: Int = 0
for let i: Int in is {
    count += i
}
```

Listing E.59: Flint looping through array example

```
let ds: [Address: Int] = [0x0: 0,
    0x1: 1, 0x2: 2]
let count: Int = 0
for let a: Address in is {
    count += ds[a]
}
```

Listing E.60: Flint looping through dictionary example

```
counter := initial_value
while (counter < final_value) {
    iteration_variable = iterator_value(counter)

    // Loop body

    counter := update(counter)
}
```

```
count := 0;
counter := 0;
while (counter < size_is) {
    i := is[counter];

    // Loop body
    count := count + i;

    counter := counter + 1;
}
```

Listing E.61: Boogie translation of iterating through array

```
count := 0;
counter := 0;
while (counter < size_ds) {
    a := keys_ds[counter];

    // Loop body
    count := count + ds[a];

    counter := counter + 1;
}
```

Listing E.62: Boogie translation of iterating through dictionary

Conditionals

```
if condition {
    // True case
} else {
    // False case
}
```

Listing E.63: Flint if-statement

```
if (condition) {
    // True case
} else {
    // False case
}
```

Listing E.64: Boogie if-statement translation

Become statements

```
become Green
```

Listing E.65: Flint become statement

```
// State value of 0 corresponds to Green
stateVariable_TrafficLight := 0;
```

Listing E.66: Boogie become-statement translation

Return statements

```
func ten() -> Int {
    return 10
}
```

Listing E.67: Flint return statement

```
procedure ten() returns (returnVariable: int) {
    returnVariable := 10;
    return;
}
```

Listing E.68: Boogie return statement translation

Do-Catch statements

```
let callFailed: Bool = false
do {
  call extContract.someFunction()
} catch is ExternalCallError {
    callFailed = true
}
```

Listing E.69: Flint do-catch with external call example

```
// translation of exception throwing statement
var callFailed, exception_thrown: bool;
callFailed := false;

// Translation of call function
havoc global_state;

havoc exception_thrown;
if (exception_thrown) {
    // Exception was thrown, execute catch-block
    callFailed := true;
} else {
    // No exception thrown, continue executing remainder of do-block
}
```

Listing E.70: Boogie translation of do-catch block

Structs

Struct fields

```
struct S {
    var i: Int = 10
    var j: Bool = false
}
```

Listing E.71: Flint struct example

```
var i_S: [int]int;
var j_S: [int]bool;

var nextInstance_S: int;
```

Listing E.72: Boogie translation of struct

Struct methods

```
struct A {
    var b: B

    func bValue() -> Int {
        return self.b.i
    }
}

struct B {
    var i: Int
}
```

Listing E.73: Flint example of struct method

```
// Field memory space
var b_A: [int]int;
var i_B: [int]int;

// Struct methods
procedure bValue_A(structInstance: int) returns
    (returnValue: int)
{
    returnValue := i_B[b_A[structInstance]];
    return;
}
```

Listing E.74: Boogie translation of a struct method

```
struct A {
    var i: Int

    init() {
        self.i = 0
    }
}
```

Listing E.75: Flint example struct init method

```
// Field memory space
var i_A: [int]int;
var nextInstance_A: int;

// Struct methods
procedure init_A() returns (newStruct:
    int)
    modifies nextInstance_A;
{
    var newInstance: int;
    newInstance := nextInstance_A;

    i_A[newInstance] := 0;

    nextInstance_A := nextInstance_A + 1;

    newStruct := newInstance;
    return;
}
```

Listing E.76: Boogie translation of struct init method

Instances

```
let w: Wei = Wei(0)
```

Listing E.77: Flint example of a struct variable

```
var w: int;
call w := init_Wei(0);
```

Listing E.78: Boogie translation of struct variable

Accessing properties

```
let w: Wei = Wei(0)
let v: Int = w.rawValue

// ...

struct A {
    var b: B
}
struct B {
    var i: Int
}
// ‘a’, an instantiation of struct A
let i: Int = a.b.i
```

Listing E.79: Flint example of accessing struct fields

```
// Field memory space
var rawValue_Wei: [int]int;
var b_A: [int]int;
var i_B: [int]int;

// Accessing fields
var w, v, a, i: int;
call w := init_Wei(0);
v := rawValue_Wei[w];

i := i_B[b_A[a]] ;
```

Listing E.80: Boogie translation of accessing struct fields

Calling struct methods

```
struct A {
    var b: B

    func bValue() -> Int {
        return self.b.i
    }
}

struct B {
    var i: Int
}
// ‘a’, an instantiation of struct A
let i: Int = a.bValue()
```

Listing E.81: Flint example of calling a struct method

```
// Field memory space
var b_A: [int]int;
var i_B: [int]int;

// Struct methods
procedure bValue_A(structInstance: int)
    returns (returnValue: int)
{
    returnValue := i_B[b_A[structInstance]];
    return;
}

// Accessing fields
var a, i: int;

call i := bValue(a);
```

Listing E.82: Boogie translation of calling a struct method

Structs as function arguments

```
struct A {
    var i: Int

    init() {
        self.i = 0
    }
}

func increment(a: inout A)
    mutates (A.i)
{
    a.i += 1
}

let a: A = A()
increment(&a)
assert (a.i == 1)
```

Listing E.83: Flint example of passing a struct by reference

```
// Field memory space
var i_A: [int]int;
var nextInstance_A: int;

// Struct methods
procedure init_A()
    modifies nextInstance_A;
{
    var newInstance: int;
    newInstance := nextInstance_A;

    i_A[newInstance] := 0;

    nextInstance_A := nextInstance_A + 1;
}

// Contract methods
procedure increment(a: int)
    modifies i_A;
{
    i_A[a] := i_A[a] + 1;
}

// Passing by referece
var a: int;
call a := init_A();
call increment(a);
assert (i_A[a] = 1);
```

Listing E.84: Boogie translation of passing a struct by reference

External Calls

```
external trait ExternalContract {}
contract C {
    var count: Int = 10

    invariant (count == 10)
}

C :: (any) {
    func increment() {
        let externalContract = ExternalContract(address: 0x0)
        call! externalContract.someFunction()
    }
}
```

Listing E.85: Flint external function call

```
var count_C: int;

procedure increment_C()
  // Invariant translation
  requires (count_C == 10);
  ensures (count_C == 10);
{
  var call_succeeded: bool;

  // Make sure contract invariant holds at external call location
  assert (count_C == 10);
  havoc call_succeeded;
  if (¬call_succeeded) {
      // Caller transaction is reverted
      assume (false);
  }
  // All global variables could have been modified during the external call, via contract
    rentrancy
  havoc count_C;

  // Can guarantee that that after the external call, the contract invariant will still
    hold
  assume (count_C == 10);
}
```

Listing E.86: Boogie translation of external function call

Standard Library

Global functions

Send

```
procedure send(account: int, source: int)
    modifies (rawValue_Wei);

    ensures (rawValue_Wei[source] == 0);
{
    rawValue_Wei[source] := 0;
}
```

Listing E.87: Boogie representation of send function

Bibliography

[1] *Solidity — Solidity 0.5.2 documentation*. [Online]. Available: https://solidity.readthedocs.io/ (visited on 01/22/2019).

[2] *Ethereum Project*. [Online]. Available: https://www.ethereum.org/ (visited on 01/22/2019).

[3] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale", 2018, ISSN: 0046-5070. DOI: arXiv:1802.06038v1. arXiv: 1802.06038. [Online]. Available: http://arxiv.org/abs/1802.06038.

[4] *The Story of the DAO — Its History and Consequences*. [Online]. Available: https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee (visited on 01/23/2019).

[5] *The Multi-sig Hack: A Postmortem | Parity Technologies*. [Online]. Available: https://www.parity.io/the-multi-sig-hack-a-postmortem/ (visited on 01/23/2019).

[6] F. Schrans, S. Eisenbach, and S. Drossopoulou, "Writing safe smart contracts in Flint", *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming - Programming'18 Companion*, no. June, pp. 218–219, 2018, ISSN: 0008-5472. DOI: 10.1145/3191697.3213790. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3191697.3213790.

[7] *Truffle Suite | Sweet Tools for Smart Contracts*. [Online]. Available: https://truffleframework.com/ (visited on 01/23/2019).

[8] I. Sergey, A. Kumar, and A. Hobor, "Scilla: a Smart Contract Intermediate-Level LAnguage", 2018. arXiv: 1801.00687. [Online]. Available: http://arxiv.org/abs/1801.00687.

[9] K. Bhargavan, N. Swamy, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, and T. Sibut-Pinote, "Formal Verification of Smart Contracts", *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS'16*, pp. 91–96, 2016, ISSN: 0009-7322. DOI: 10.1145/2993600.2993611. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2993600.2993611.

[10] L. Alt and C. Reitwiessner, "SMT-Based Verification of Solidity Smart Contracts", in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2018, pp. 376–388, ISBN: 978-3-030-03427-6.

[11] K. R. M. Leino, "This is boogie 2", Jun. 2008, [Online]. Available: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

[12] D. Liew, C. Cadar, and A. F. Donaldson, "Symbooglix: A Symbolic Execution Engine for Boogie Programs", *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pp. 45–56, 2016. DOI: 10.1109/ICST.2016.11.

[13] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger Final Draft - Under Review", 2015. [Online]. Available: http://gavwood.com/Paper.pdf.

[14] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts", in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, New York, NY, USA: Springer-Verlag New York, Inc., 2017, pp. 164–186, ISBN: 978-3-662-54454-9. DOI: 10.1007/978-3-662-54455-6_8. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8.

[15] *Scanning Live Ethereum Contracts for the "Unchecked-Send" Bug.* [Online]. Available: http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/ (visited on 01/25/2019).

[16] *Remix - Solidity IDE.* [Online]. Available: https://remix.ethereum.org/ (visited on 01/22/2019).

[17] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter", in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, New York, New York, USA: ACM Press, 2016, pp. 254–269, ISBN: 9781450341394. DOI: 10.1145/2976749.2978309. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2976749.2978309.

[18] B. Mueller, "Smashing Ethereum Smart Contracts for Fun and Real Profit", *Proceedings of HITB SECCONF Amsterd4m 2018*, pp. 1–54, 2018. [Online]. Available: https://github.com/b-mueller/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf.

[19] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts", *CoRR*, vol. abs/1806.01143, 2018. arXiv: 1806.01143. [Online]. Available: http://arxiv.org/abs/1806.01143.

[20] G. Rosu, "K: A semantic framework for programming languages and formal analysis tools", *Dependable Software Systems Engineering*, no. Vc, pp. 186–206, 2017. DOI: 10.3233/978-1-61499-810-5-186.

[21] F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang, "KRust: A Formal Executable Semantics of Rust", 2018. arXiv: arXiv:1804.10806v1. [Online]. Available: https://arxiv.org/pdf/1804.10806.pdf.

[22] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu, "Kevm: A complete semantics of the ethereum virtual machine", pp. 1–33, 2017.

[23] J. Jiao, S. Kan, S. Lin, D. Sanán, Y. Liu, and J. Sun, "Executable operational semantics of solidity", *CoRR*, vol. abs/1804.01295, 2018. arXiv: 1804.01295. [Online]. Available: http://arxiv.org/abs/1804.01295.

[24] *Welcome! | The Coq Proof Assistant.* [Online]. Available: https://coq.inria.fr/ (visited on 01/22/2019).

[25] L. Alt and C. R. B, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice.* Springer International Publishing, 2018, vol. 11247, pp. 376–388, ISBN: 978-3-030-03426-9. DOI: 10.1007/978-3-030-03427-6. [Online]. Available: http://link.springer.com/10.1007/978-3-030-03427-6.

[26] C. A. R. Hoare, "An axiomatic basis for computer programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969, ISSN: 00010782. DOI: 10.1145/363235.363259. [Online]. Available: http://portal.acm.org/citation.cfm?doid=363235.363259.

[27] L. de Moura and N. Bjørner, "Z3: An efficient smt solver", in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3.

[28] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli, "CVC4", in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, G. Gopalakrishnan and S. Qadeer, Eds., ser. Lecture Notes in Computer Science, Snowbird, Utah, vol. 6806, Springer, Jul. 2011, pp. 171–177. [Online]. Available: http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf.

[29] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints", in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77, Los Angeles, California: ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. [Online]. Available: http://doi.acm.org/10.1145/512950.512973.

[30] J. C. King, "Symbolic execution and program testing", *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, ISSN: 0001-0782. DOI: 10.1145/360248.360252. [Online]. Available: http://doi.acm.org/10.1145/360248.360252.

[31] B. Meyer, *Object-oriented software construction (2nd ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997, ISBN: 0-13-629155-4.

[32] S. Drossopoulou, J. Noble, and S. Eisenbach, "Holistic Specifications for Robust Programs", *Communications of the ACM*, 2019, Submitted.

[33] *F*: A Higher-Order Effectful Language Designed for Program Verification.* [Online]. Available: https://fstar-lang.org/ (visited on 01/25/2019).

[34] *Microsoft Research - Emerging Technology, Computer, and Software Research.* [Online]. Available: http://research.microsoft.com/ (visited on 01/25/2019).

[35] *Prosecco.* [Online]. Available: http://prosecco.gforge.inria.fr/ (visited on 01/25/2019).

[36] R. Leino and M. Moskal, *Co-induction simply: Automatic co-inductive proofs in a program verifier*, Jul. 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/co-induction-simply-automatic-co-inductive-proofs-in-a-program-verifier/.

[37] E. Laube, "Design and Implementation of a JML frontend to Boogie", 2006.

[38] Z. Rakamarić and M. Emmi, "SMACK: Decoupling source language details from verifier implementations", in *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, A. Biere and R. Bloem, Eds., ser. Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 106–113.

[39] *Nadia Polikarpova / Boogaloo.* [Online]. Available: https://bitbucket.org/nadiapolikarpova/boogaloo/ (visited on 01/25/2019).

[40] A. Lal, S. Qadeer, and S. Lahiri, *Corral: A solver for reachability modulo theories*, Jan. 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/corral-a-solver-for-reachability-modulo-theories/.