

FINAL YEAR PROJECT

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

Steel: A Developer Ecosystem For The Flint Programming Language

Authors:

Mohammad Chowdhury

Supervisor(s):

Professor Susan Eisenbach

Dr Robert Chatley

June 17, 2019

Submitted in partial fulfillment of the requirements for the MEng Computing of
Imperial College London

Abstract

Flint is a smart contract programming language that has been designed with safety in mind. It boasts features to help developers create robust smart contracts. It is unlikely that Flint will be adopted without a strong developer ecosystem. Developers typically use a myriad of tools to help them write, run and test code. The blockchain programming model differs from traditional programming paradigms as execution of contracts costs money and contracts cannot be modified after they have been deployed. Thus, it is especially important that an ecosystem exists for smart contract development which helps developers write secure, correct and economical contracts. There has been no significant work done on creating a Flint developer ecosystem - the only tools that exist are *language-flint*, *vscode-flint* & *flint/vim*. They all perform a similar function and are limited in scope. Active development has ceased on all three tools [24, 33, 31].

We have created an ecosystem for the Flint programming language. The ecosystem has a language server & a syntax highlighter. These tools provide rich language support for Flint within the editor. We also provide a set of analysis tools to help Flint developers follow best practices and understand the economics of their contracts. We have created a unit testing framework to help developers write correct Flint contracts, the framework also measures test coverage. We have created an interactive console to enable developers to quickly deploy and interact with their contracts on their local machine. We have also created a tool that can be used to launch a local blockchain. We show that the ecosystem adds value by demonstrating how it can be used to develop Flint programs safely.

Acknowledgments

I would like to thank:

- Professor Susan Eisenbach, Professor Sophia Drossopoulou and Dr Robert Chatley for their invaluable guidance and support throughout this project
- My friends, who have supported me throughout my years at Imperial
- and, of course, my parents and family, who have always been there for me and who without I would not be where I am today.

Contents

1	Introduction	1
1.1	Contributions	2
2	Background	4
2.1	Engineering Smart Contracts	4
2.1.1	Software Engineering	4
2.1.2	Smart Contract Programming	6
2.2	IDE (Language Aware Editor)	6
2.2.1	Language Server Protocol (LSP)	7
2.2.2	Code Analyser	9
2.2.3	Syntax Analyser - Error Tolerance	9
2.2.4	Learnable Programming	12
2.2.5	Existing Tools	13
2.3	Contract Analysis	17
2.3.1	Remix IDE analysis	18
2.3.2	Solgraph	18
2.3.3	Smart Check	19
2.3.4	Formal Verification	19
2.4	Contract Deployment and Interaction	20
2.4.1	Truffle Console	20
2.4.2	Truffle Deploy	21
2.4.3	ZeppelinOS	21
2.5	Testing	21
2.5.1	Importance of Testing	21
2.5.2	Testing Smart Contracts	22
2.5.3	Testing Language	23
2.5.4	Truffle Test	24
2.6	Code coverage	24
2.6.1	Methods to implement code coverage	24
2.6.2	Types of code coverage	24
2.6.3	Existing Tools	26
2.7	Flint	28
2.7.1	Anatomy of Flint Contract	28
2.7.2	Caller Protections	31
2.7.3	Type states	32
2.7.4	Structs and Enums	32

2.7.5	Traits	33
2.8	Execution of Smart Contracts	35
2.8.1	Ethereum Platform	35
2.8.2	Transactions	36
2.8.3	Contract Deployment	37
2.8.4	Contract Interaction	38
2.8.5	Contract Events	39
2.8.6	Flint Code Generation	39
2.9	Conclusion	40
3	Ecosystem	42
3.1	Server Client Model	42
3.1.1	VSCode extension	43
3.2	Flint-Colour	44
3.2.1	Description	44
3.2.2	Implementation	44
3.2.3	Conclusion	45
3.3	Flint-Block	45
4	Language Server	47
4.1	Motivation	47
4.2	Language Server Protocol (LSP)	47
4.3	Implementation	48
4.3.1	Code Validator	48
4.3.2	Error Tolerance Rules	49
4.4	Conclusion	50
5	Contract analysis	51
5.1	Caller & State Protection Analysis	51
5.1.1	Description of Analysis	51
5.1.2	Implementation	52
5.1.3	Conclusion	52
5.2	Contract visualisation	53
5.2.1	Description of Analysis	53
5.2.2	Implementation	56
5.2.3	Conclusion	56
5.3	Gas Estimation	57
5.3.1	Description Of Analysis	57
5.3.2	Implementation	58
5.3.3	Conclusion	59
6	Testing Framework	60
6.1	Introduction	60
6.2	Anatomy Of Test File	63
6.3	Flint-Test DSL	63
6.4	Contract Deployment	64

6.4.1	Constructor Arguments	65
6.5	System Architecture	66
6.5.1	Deploying a Contract	67
6.5.2	Calling Functions	67
6.5.3	Checking For Reverted Transactions	68
6.6	Test Framework Features	68
6.6.1	Testing Caller Protections	68
6.6.2	Testing State Protections	70
6.6.3	Testing For Exceptions	72
6.6.4	Testing Events	73
6.6.5	Testing Ether Transactions	74
6.7	Code coverage	76
6.7.1	Implementation	76
6.8	Conclusion	81
7	Interactive Console (REPL)	82
7.1	Introduction	82
7.2	REPL Features	82
7.2.1	Deploying a Contract	83
7.2.2	Calling Functions	83
7.2.3	Querying Event Logs	84
7.3	Implementation	85
7.3.1	Overview	85
7.3.2	Deploying a Contract	86
7.3.3	Contract Interaction	87
7.4	Conclusion	87
8	Evaluation	89
8.1	Testing Framework	89
8.1.1	Speed Of Testing Framework	90
8.1.2	Testing Access Control : Caller Capabilities	91
8.1.3	Testing Access Control : States	95
8.1.4	Testing Events	97
8.1.5	General Testing Of Contracts	99
8.1.6	Testing For Exceptions	101
8.1.7	Testing Ether Transactions	102
8.1.8	Code Coverage	104
8.1.9	Feature Comparison	106
8.1.10	Summary	108
8.2	REPL (Interactive Console)	109
8.2.1	Deploying A Contract	109
8.2.2	Calling Functions	111
8.2.3	Sending Money To A Contract	113
8.2.4	Querying Event Logs	113
8.2.5	Summary	115
8.2.6	Conclusion	116

8.3	Language Server & Contract Analysis	116
8.3.1	Language Server	116
8.3.2	Contract Analysis	118
8.4	Entire Ecosystem	120
8.4.1	Strengths & Limitations	120
8.4.2	Conclusion	121
9	Conclusion	122
9.1	Future Work	123
9.2	Challenges	123
A	User Manual	131
A.1	VSCode Extensions	131
A.2	Flint Language Server	131
A.2.1	VSCode Extension	131
A.2.2	Command Line Interface	131
A.3	Analysis Tools	131
A.4	Flint-Test	132
A.5	Flint-REPL (Interactive Console)	132
B	Third Party Libraries	134
C	BNF's	135
C.1	Flint-Test DSL BNF	135
C.2	Flint-REPL BNF	136
D	Flint Contracts Used In Main Body Of Report	138
E	Contracts Used In Evaluation	140
E.1	Flint	140
E.2	Solidity	143

Chapter 1

Introduction

Ethereum is an open blockchain platform that enables developers to create decentralized applications, called smart contracts, that run on a permissionless blockchain [93]. This permits programmers to run their programs with complete transparency and facilitates execution of programs in a trustless environment. Solidity is currently the official smart contract programming language for the Ethereum blockchain. However, Solidity smart contracts have been subject to many flaws leading to the loss of over \$40 million dollars worth of assets. The language Solidity is a major contributor to these security flaws, it was not designed with safety in mind [8, 59]. Flint was designed to address this issue and be a replacement for Solidity. Flint is a statically typed smart contract programming language which boasts features to make smart contract development safer [31].

The process of software engineering involves more than just the programming language. Developers typically use a myriad of tools to aid them whilst developing. This includes but is not limited to language aware editors, unit testing frameworks, interactive consoles and package management systems. JetBrains survey of the development ecosystem in 2018 reports that 69% of developers regularly use lightweight language aware desktop editors such as VSCode, 82% use full fledged integrated development environments and 70% of developers use unit tests [78]. The developer ecosystem is an integral part of software development. Without it, developers are more likely to make mistakes, produce incorrect code and generally have a poor experience whilst developing. It is important to both independent software developers and large corporations. Companies such as Google have software engineering practices that make use of many tools to aid the process of software development [38]. Thus, for Flint to truly be a viable option for smart contract programming, it needs to have a developer ecosystem. There has been no significant work done on creating a Flint developer ecosystem - the only tools that exist are *language-flint*, *vscode-flint* & *flint/vim*. They all perform a similar function and are limited in scope. Active development has ceased on all three tools [24, 33, 31].

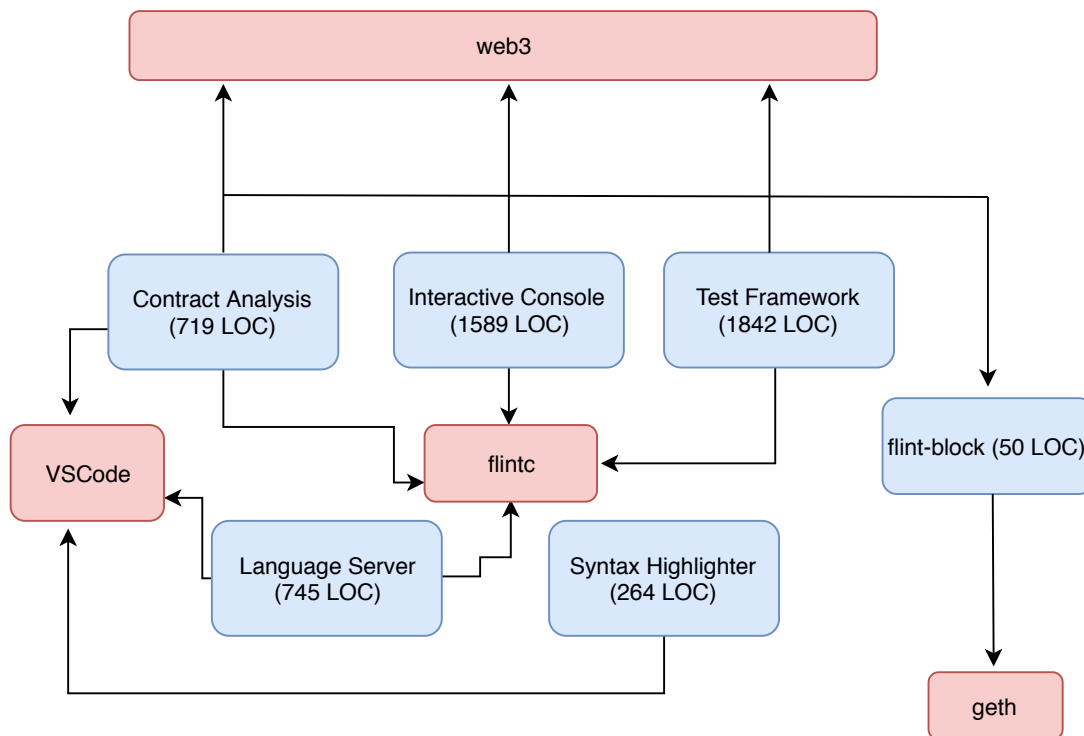


Figure 1.1: Contributions to Ecosystem (in blue)

1.1 Contributions

This project proposes to create a developer ecosystem for Flint. By exploring different language ecosystems and understanding the process of software engineering with smart contracts, we create a set of tools to help Flint developers develop safer and more secure smart contracts. The ecosystem should enable a developer to better understand and leverage the safety features that Flint provides, ease the process of writing code and interacting with it and provide a way for developers to test their code. Programmers have to pay to run their programs on a blockchain, the ecosystem should provide a tool to help programmers understand the economics of their contracts.

We have created the following set of tools to achieve these goals:

- **Interactive console (flint-repl).** It enables developers to deploy contracts to a local blockchain and interact with it. Currently, smart contracts can only be run on the blockchain and the process of manually deploying is cumbersome. Thus, the ecosystem needs to provide a convenient way for developers to deploy and interact with contracts whilst they are developing. Please refer to chapter 7 for more information on the interactive console.
- **Flint language server (flint-lsp).** Provide live feedback to the developer as they are writing code i.e. report syntax and semantic errors. This will ease the process of development as there is no need for manual compilation and the programmer gets feedback instantly about their code. The tool conforms

to the language server protocol so it is not tied to any particular editor. Please refer to chapter 4 for more information on the language server.

- **Contract analysis tools (flint-ca).** Analyse the programmers contracts and provide insights about how well they are using Flint's safety features and help them understand the economics of their contracts. These tools are available from within the editor and can be used as the developer is writing code. Please refer to chapter 5 for more information on the contract analysis tools.
- **Unit testing framework with test coverage (flint-test).** Provide a way for developers to test their code, testing increases a developers confidence in the correctness of their code. Test coverage is used to increase visibility on the reliability of a codebase i.e. uncover untested code. Testing is especially important for smart contracts as they cannot be modified once deployed. Please refer to chapter 6 for more information on the test framework.
- **Local Blockchain (flint-block).** Provide a quick and easy way for developers to launch a local blockchain which comes pre-loaded with users. Please refer to chapter 3 for more information on this tool.
- **Syntax Highlighter (flint-colour).** Simplify the process of reading Flint code. Please refer to chapter 3 for more information on the syntax highlighter.

Figure 1.1 is a diagrammatic representation of our contributions. The boxes marked in blue are components we have developed, the boxes marked in red are third party libraries we have used. The diagram also contains an approximate size of each component using lines of code (LOC) to measure size. The arrows between the components should be read as *use*.

Chapter 2

Background

In this chapter, we discuss what the process of software engineering entails, explore existing ecosystems and present an overview of Flint and the platform it executes on. For our discussion on ecosystems, we have chosen to focus on the Solidity ecosystem. We have chosen Solidity as it is the official language for the Ethereum blockchain and currently the most popular smart contract programming language.

Throughout this chapter we refer to Truffle, which is a development framework for Solidity programs. It is a command line program which streamlines the process of building, testing and deploying Solidity smart contracts [86].

2.1 Engineering Smart Contracts

2.1.1 Software Engineering

The software engineering life cycle can be broken into roughly 4 sections; design, implementation, deployment and maintenance. Most development ecosystems provide a set of tools to help developers in each of these stages.

The design stage usually consists of discussions between engineers, drawing up system specifications and planning developer time. This is generally language agnostic and there exist tools which ease this process that can be used irrespective of the programming language used e.g. Trello [82].

The implementation part of the software engineering life cycle includes writing, running and testing code. The JetBrains developer survey suggests that many software engineers use a language aware editor whilst programming [78]. The benefits of this are many fold, editors provide functionality such as auto-completion, find and replace, live code validation, quick code navigation and other language specific features e.g. detecting code smells. Please refer to section 2.2 for more details on code editors.

During development engineers run their code and interact with it to ensure that it

is working as expected. This is an integral part of the developer experience. Most programming languages offer programmers the ability to quickly run programs they are developing on their local machines. From the authors own experience and from talking to other developers, we know that it is highly convenient to be able to run programs locally on your machine and interact with them. It enables developers to catch errors early, have confidence their code is working correctly and enables rapid experimentation. Some of the most popular languages in the world such as JavaScript, Python and Java all have tools to enable programmers to quickly run programs on their local machines and interact with it [72, 51, 39, 92]. Languages like JavaScript and Python provide programmers with a REPL (read-eval-print loop) which is an interactive programming environment. Truffle which is the official development ecosystem for Solidity provides a REPL to enable developers to quickly deploy contracts to a local blockchain and interact with them [83].

Testing is an integral part of software development. It enables developers to ensure that software they are developing is meeting the intended specification increasing their confidence in the correctness of their code. This in turn helps reduce the number of bugs and provides a safety net for developers when they need to refactor. Many developers utilise unit testing as part of their regular development life cycle [78]. Test Driven Development is an approach to writing code which is popular in industry, it was introduced by Kent Beck in his book 'Test Driven Development: By Example'. Many software engineering guides advocate for this method [49]. Test driven development involves writing tests for a piece of code before writing it. The test initially fails and the programmer implements enough functionality to make the test pass. This process is then repeated for all the code that is added to the application. Truffle provides a unit testing framework for Solidity smart contract developers. Testing is especially important for smart contracts as they cannot be modified after they have been deployed.

The next stage of the software engineering life cycle is deployment. This is the process of moving code from a developer's local machine to a production environment. This includes deploying code to machines for use over the internet or if the codebase is a library then deploying it to a repository for use by other programmers. In the context of smart contract programming, deployment will consist of deploying to a production blockchain or uploading to a smart contract repository. The Ethereum foundation manage their own smart contract repository. Developers can use *EthPM* (Ethereum Package Manager) to interact with this repository [87]. The process of deployment is highly dependent on the nature of the code being deployed. Large companies have their own proprietary build systems which are developed in house to compile and deploy their code. They are complex systems as the codebases they work on are highly complex and require careful management of dependencies to ensure deployment is smooth [38]. However, many language ecosystems come bundled with tools to ease this process. The tools can automatically configure dependencies, compile the code and then deploy it. Truffle provides a deployment tool which makes it easy for developers to compile and deploy their smart contracts to a

blockchain [85]. We will explore the process of deployment to the blockchain further in section 2.8.3.

The last stage of the software development life cycle is maintenance. This includes bug tracking, monitoring and collecting telemetry data. Many language agnostic solutions exist to address this part of the software engineering cycle e.g. JIRA can be used for bug tracking [43]. Another important aspect of writing code is version control, however this is generally a solved problem as there are many version control systems and they are all language agnostic e.g. git.

In conclusion, we have found that although development ecosystems can vary from language to language. They all usually provide tools to ease the different aspects of software engineering.

2.1.2 Smart Contract Programming

Many of the ideas and principles from traditional software engineering also apply to smart contract development. However, the blockchain programming model imposes a few constraints on the programmer. Public blockchains such as Ethereum cost money to use. Users are required to pay whenever they wish to deploy and/or interact with contracts. For private blockchains, whether it costs or not depends on the owner of the blockchain. This means that for a programmer to be able to effectively develop smart contracts, it is important that the ecosystem provides a local blockchain. This will give the programmer the ability to test their contracts without worrying about the cost of execution. Also, public blockchains such as Ethereum are not suitable for rapid feedback cycles, they limit the number of transactions that can be processed per second for security reasons. Contracts deployed to a blockchain cannot be modified after deployment. Therefore, it is important that the ecosystem has mechanisms to help the programmer write correct contracts. We discuss the blockchain and its mechanics in more detail in section 2.8.1 [94, 93, 50].

2.2 IDE (Language Aware Editor)

A language aware editor is an integral part of any developers toolbox. They usually come in three flavours; full fledged desktop IDE, light weight desktop editor with language aware plugins and cloud based editors. IDE's come packaged with a suite of features that help developers write code. For example, IntelliJ IDEA is an integrated development environment for the Java language produced by JetBrains has many Java specific features. It has smart code completion, supports many popular Java frameworks and can detect code smells. They are very popular amongst developers [78, 39]. Writing an IDE from scratch is a time consuming task, however IDEs like IntelliJ usually come with a full fledged plug in system which enables developers to augment the functionality of the IDE and add support for additional languages [40]. Lightweight desktop editors are also popular with the development community, notable examples are editors like Microsofts VSCode & GitHubs Atom

[78]. These editors come with a set of minimal features to aid developers in writing code. Language support is added via extensions. The benefit of these editors in comparison to the full fledged development environments is that they are a smaller and have a lower memory footprint. The last option is a cloud based editor. The Mozilla foundation have released a toolkit facilitating the development of cloud based editors, however it is currently still in the early stages of development [79].

Language support can be added to an editor for Flint in the following ways:

- Developing an entirely new editor specifically for Flint.
- Extending an IDE for another language e.g. adding a plugin for IntelliJ.
- Extending a lightweight desktop editor.

The first option would require development of a complete editor from scratch. This would involve re-implementing features that are already present in other editors. Also, many editors have a rich plugin ecosystem surrounding them which support development. Creating an entirely new editor will result in Flint developers losing access to these ecosystems. The benefit of this approach is that it grants you flexibility with regards to the design of the editor, it can be tailored for Flint development.

The second and third option are similar in the sense that you are extending an existing piece of software to add support for Flint. However, extending an IDE means that users have to pay the cost of running the entire IDE. There is little benefit to the user in running the entire IDE as it is unlikely that they will be able to make use of many of the the features that the IDE provides as it was designed for another language. Extending a lightweight desktop editor means that developers only need to run what is necessary to support development of Flint code.

Plugins are generally run as part of the editor process [56]. Thus, running plugins introduces additional overhead for the editor. This can affect the development experience e.g. slow down the editor if the plugins are computationally expensive. Additionally, the plugin API's for different editors are not standardised meaning that tools written for one editor are not easily portable to another editor. Thus, developers are forced to use a particular editor. These drawbacks can be mitigated by using a protocol like the LSP, we will discuss this in the next section.

2.2.1 Language Server Protocol (LSP)

The Language Server Protocol (LSP) aims to solve this problem by defining a standardized communication format between language tools and editors. The protocol was created by Microsoft [46]. In the LSP protocol, you have a client which is the editor and the language tool which is known as the language server. The client communicates with the language server using the LSP protocol.

This enables the language tool, referred to as server from now on, to be run in a different process to the editor. Thus, the editor will not pay an overhead for running the tool. It also means the author of the tool is free to run computationally intensive tasks and is not constrained by the plugin API of any editor. Since the protocol defines a standard, the author of the tool can develop a single tool and support many different editors [46]. Currently, a large selection of editors support the LSP protocol, this list includes Visual Studio Code, Visual Studio, Atom, Sublime and Vim [80]

The editor and server communicate via JSON-RPC. The communication between editor and server is language agnostic and is centered around the structure of the document i.e. line and column numbers. Figure 2.1 illustrates a sequence of request-reply messages between the language server and the editor. The editor fires up the

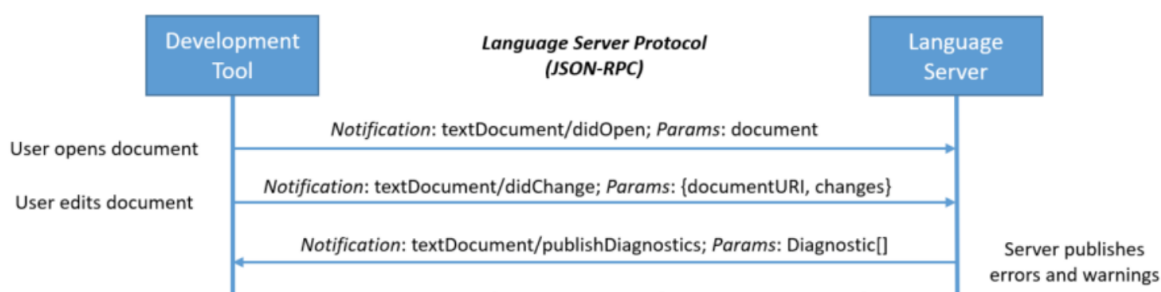


Figure 2.1: Example communication between client and server using LSP [56]

language server whenever it loads a file containing the source code for a language that the server supports. The editor sends notifications to the language server notifying it whenever the source code changes. The language server reacts to notifications and responds appropriately. In the case of Figure 2.1 the language server responds by sending back a list of diagnostics [56]. Diagnostics are used to communicate what issues (if any) are present in the source code. This includes both syntax and semantic errors.

Figure 2.2 illustrates the structure of a diagnostic that the server sends to the client. The important parts are the range, message and severity. The range tells the editor the location of the issue. Figure 2.3 is an example of a range object. The message is used to communicate to the developer what the issue is. The severity is used to indicate if the diagnostic is a warning, hint or an error. The editor uses this to provide visual cues to the user about the nature of the issue e.g. errors are underlined in red [71].

In addition to diagnostics, the LSP supports many other features such as jump to definitions, auto-completion, signature help, hover tool tips and more. The full list of features that the LSP protocol supports can be found on their website.

```

interface Diagnostic {
  /**
   * The range at which the message applies.
   */
  range: Range;

  /**
   * The diagnostic's severity. Can be omitted. If omitted it is up to the
   * client to interpret diagnostics as error, warning, info or hint.
   */
  severity?: number;

  /**
   * The diagnostic's code, which might appear in the user interface.
   */
  code?: number | string;

  /**
   * A human-readable string describing the source of this
   * diagnostic, e.g. 'typescript' or 'super lint'.
   */
  source?: string;

  /**
   * The diagnostic's message.
   */
  message: string;

  /**
   * An array of related diagnostic information, e.g. when symbol-names within
   * a scope collide all definitions can be marked via this property.
   */
  relatedInformation?: DiagnosticRelatedInformation[];
}

```

Figure 2.2: Structure of diagnostic [56]

```

{
  start: { line: 5, character: 23 },
  end : { line 6, character : 0 }
}

```

Figure 2.3: Structure of LSP range [71]

2.2.2 Code Analyser

To create an effective language server, the tool needs to understand the source code. This can be done by reusing the compiler for the language, the compiler can be used to process the code and find any issues. Since the code which is being processed is in the process of being written, it is likely that the code is syntactically incorrect. This places an additional constraint on the parser used in the language server as it needs to be able to tolerate invalid code. This is normally not an issue with bottom up parsers as they can handle multiple syntax errors. However, the parser for Flint is a recursive descent one. It cannot tolerate more than one syntax error. Thus, we have to be able to augment the Flint parser to be error tolerant and be able to continue parsing when it encounters a syntax error. In the next section, we will discuss the theory behind implementing error tolerance. Additionally, the code analyser used by the language server needs to be fast as it should provide live feedback to the user as they type.

2.2.3 Syntax Analyser - Error Tolerance

There are two different approaches to error tolerance:

- Error Repair
- Error Recovery

Error repair is where the parser attempts to modify an incorrect program i.e. fix syntax issues and continues parsing. This method has some significant drawbacks - there is no guarantee that the parsers modification respects the programmers intent, it is difficult to implement and has high time complexity [16].

Error recovery is where the parser recovers from an error and attempts to continue parsing. There are three different types of error recovery: local recovery, regional recovery and global recovery. Local recovery is when the error recovery scheme considers a small section of text at the location of the error when making decisions. Regional recovery is when a larger portion of the source text is considered when making decisions e.g. the entire function is considered. Global recovery schemes consider the entire program when making decisions. Local recovery schemes have the lowest costs in terms of memory and time, however they can sometimes make sub-optimal decisions. Local recovery schemes are also the ones that are most applicable to recursive descent parsing. Both regional and global recovery schemes are computationally expensive and are difficult to implement and many of them are not applicable to a recursive descent parser [37].

Properties of a good error recovery scheme:

1. It should return control to the parser as soon as possible.
2. It should be possible to recover from most errors and continue parsing.
3. The parser should not introduce spurious errors.
4. The parsing speed should not be affected when the error recovery code is not being executed i.e. parsing of valid code should not be slowed down.
5. The error recovery scheme should be both space & time efficient.

The first two properties are necessary of any error recovery scheme as the primary goal of such schemes is to recover from errors and put the parser back in a valid state. The third property is a property which is difficult to achieve, one can employ heuristics to ensure that in most cases the error recovery scheme does not introduce any spurious errors. We will revisit this property in a few paragraphs to illustrate in which cases this property may be difficult to achieve [37].

The most popular error recovery scheme in use is panic mode. It's popularity is mainly due to its simplicity and speed. Panic mode works as follows, whenever the parser encounters a syntax error it starts to discard tokens until it reaches a token in the synchronising set. The tokens in the synchronising set are defined statically and do not change during the parsing process. The tokens within the synchronising tokens are ones that once they are reached, it should be easy to continue parsing. The

tokens are generally terminators i.e. ones that terminate statements, code blocks & functions. In Flint an example of a synchronisation token would be the new line token as it is used to terminate statements. Once the parser consumes a synchronising token, it returns to normal parsing [16, 37].

To illustrate how this works, let us look at the following Flint code snippet:

```
1 // The new line has been put there explicitly to aid the reader
2 // invalid syntax
3 let x:Int = 1 +, 2 \\n
4
5 // valid syntax
6 let z:String = "test"
```

The code snippet is syntactically incorrect because of the comma after the addition operator. When the parser encounters the comma after the addition operator, it will go into panic mode and start discarding tokens until it reaches the newline token. At this point, the parser would synchronise and continue parsing normally. Panic mode is generally a good error recovery scheme as it is fast and simple to implement. It can also be tuned to specific languages to improve error recovery. However, one of the main issues with panic mode is that the synchronisation set is static, it does not take the current state of the parser into context when deciding the synchronisation point. The following example will help illustrate this further:

```
1 // invalid syntax as the func key word has been misspelt
2 mutating fnc test() {
3     ...
4 }
```

The above code snippet is invalid because the `func` keyword has been misspelt. A naive implementation of panic mode may synchronise at the close curly brace and skip the entire function, thus the parser loses information about this function. This can introduce errors later on, during the semantic pass whenever this function is used. One can observe from the grammar of Flint that the only token that can follow the `mutating` keyword is `func` and the only thing that can follow `func` is an identifier. The parser can then use this information to deduce that it is at the beginning of a function declaration and instead of synchronising at the close curly brace, it can synchronise at `test` and continue parsing. This means that the parser skips less input and retains more information about the source code. This method of error recovery which generates the synchronisation sets dynamically based on the current token is known as Wirth's Follow Set method. It is better than panic mode as it uses the follow set of a token in the grammar to decide what the synchronisation set should be after a syntax error occurs at that token [37]. The difference between panic mode and Wirth's method highlights why property 3 can be difficult to achieve. If we were to use panic mode as our choice of error recovery, then the parser would lose information about the `test` function. This could then cause the language server to report incorrectly that the function does not exist if in later pieces of code the developer tries to call this function. This is an example of a spurious error. In this case, Wirth's follow set method will enable us to prevent this spurious error. However, in some cases even with this method it is not possible to prevent spurious errors. The

following code listing illustrates this point further:

```
1 // invalid syntax as contract has been misspelt
2 contrwt ident {
3     ....
4 }
```

As contracts are one of the possible top level declarations in a Flint source file, the parser does not have enough information to deduce that this is a malformed contract declaration. Using a normal implementation of Wirth's follow set method, the parser will skip the entire contract declaration, this can lead to spurious errors being introduced later on. One can employ heuristics to reduce the amount of information lost. For example, in this situation the Levenshtein distance can be used to decide that `contrwt` is close enough to `contract` that we can assume that is what the programmer meant. The parser can now parse this contract declaration and prevent any spurious errors related to this contract declaration. However, this is not a general approach and will not work in all cases.

To wrap this section up, I will briefly mention the idea of incremental parsing. Suppose we have a piece of text X which has been parsed and a parse tree X' exists for X . If some changes are made to X e.g. a line is deleted and we produce Y then the parse tree of Y will be very similar to X' . The main idea behind incremental parsing is to reuse as much of X' as possible when creating the parse tree for Y [7]. This means that subsequent parses of a text can be less memory intensive as you do not need to allocate new parse tree nodes, it can also be less CPU intensive as you do not need to re-parse the entire file. Incremental parsing is not very relevant today as these techniques were developed during a time where computers had tight CPU and memory constraints. This is no longer an issue with modern day computers.

2.2.4 Learnable Programming

I will briefly mention some of the ideas in Bret Victor's discussion on Learnable programming. The purpose of the discussion was to highlight that current developer ecosystems can be improved to help people learn programming more easily [47].

Bret mentions two points about learning:

1. 'Programming is way of thinking, not a rote skill' [47]
2. People understand what they can see [47]

Although, we are not creating the development ecosystem for pedagogical purposes. It is useful to take ideas from Bret's post in order to make our ecosystem more developer friendly. Bret mentions in his paper that to make programming more learnable, both the environment and language need to play a part [47]. For the rest of this discussion, we will be focusing on the environment specific parts.

The first idea we take from Bret Victor's post is the idea that it should be easy to read the source code. To facilitate this the environment can provide helpful information to

the user. Figure 2.4 is an example of how the environment can help the programmer. In this image, the environment is telling the programmer what this function does [47]. Examples of features which help developers learn are signature completion of functions, description of what the function does as a tool tip when the function is used and providing a description of the parameters as the user types.

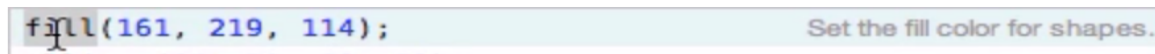


Figure 2.4: Example of an environment that helps developers read the source code [47]

The second idea we take from Bret Victor's post is the idea that visualising is a key part of learning how to program and understanding code. Bret Victor mentions that programmers should be able to visualise their code in a meaningful way. Figure 2.5 illustrates this point further. The right picture is a visualization of the code on the left.

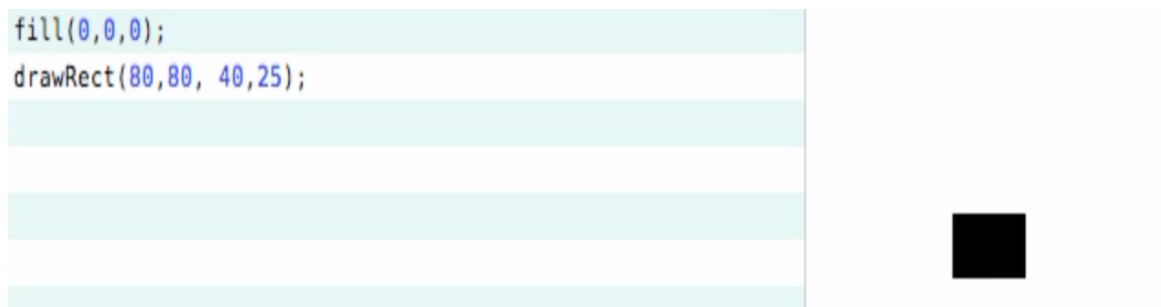


Figure 2.5: Example of an environment that visualises the code in a meaningful way [47]

2.2.5 Existing Tools

In this section, we will survey some of the tools present in the Solidity ecosystem which provide language support for Solidity within the editor.

2.2.5.1 vscode-solidity

vscode-solidity is a plugin for the VSCode editor which provides features that help the developer when developing Solidity contracts. Figure 2.6 is a screenshot of the tool. We will list some of the more pertinent features of the extension:

- Syntax highlighting
- Code completion
- Linting
- Compilation of code from the editor

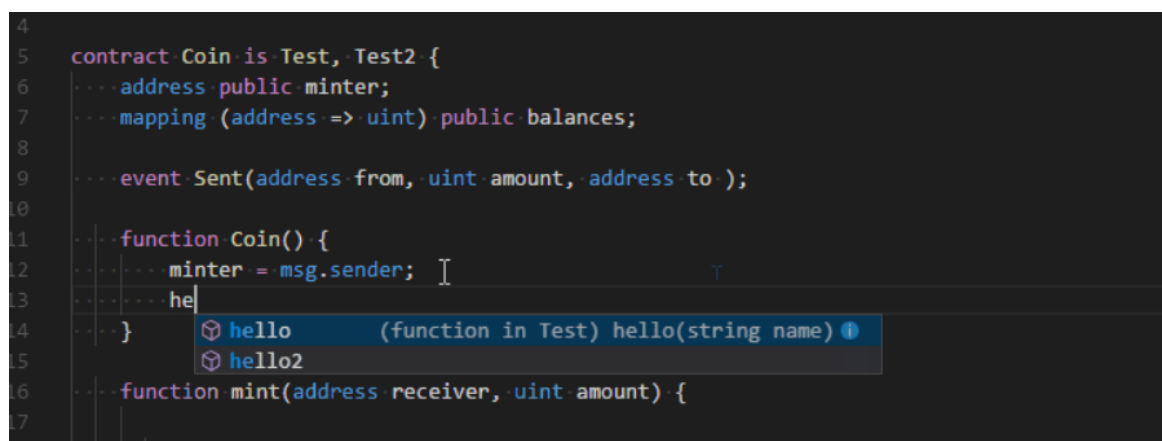


Figure 2.6: vscode-solidity [44]

2.2.5.2 Remix

The Ethereum foundation have created a browser based IDE for Solidity which can run both locally and on the cloud. It supports the development, deployment and debugging of smart contracts written in Solidity. Figure 2.7 is a snapshot of the Remix IDE [62].

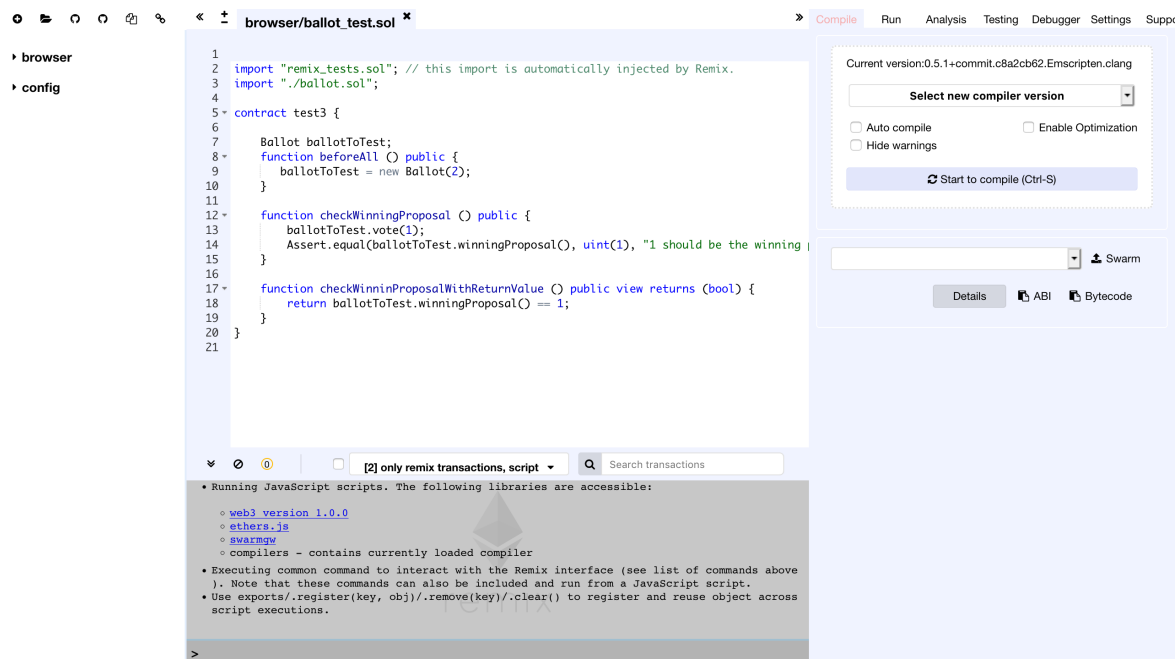


Figure 2.7: Remix IDE [62]

The Remix IDE also comes with an analysis tool which can check for a variety of issues. The issues the analysis tool can detect are listed in Figure 2.8. This is a great addition to the IDE as it enables the developer to pick up on issues with their smart contract very quickly. We will explore some of the analyses that Remix performs in section 2.3. Remix also has the ability to deploy to a private blockchain,

the blockchain can be external and this is facilitated by the use of a web3 provider or an injected provider [60]. It also enables developers to deploy their contract to a browser based blockchain using a JavaScript VM [60]. Developers can then interact with their contract on this blockchain [60].

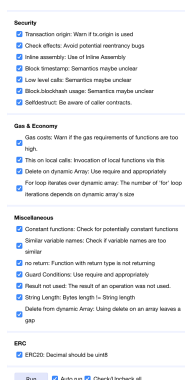


Figure 2.8: List of issues that the analysis tool picks up on [62]

It also provides the ability to write tests using a Solidity based test framework and has the ability to run your tests. Figure 2.9 is an example output of running tests in the Remix IDE [91].

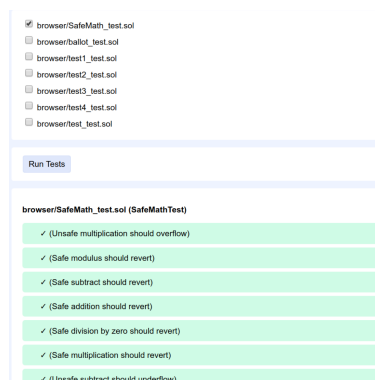


Figure 2.9: Result of running tests in Remix [23]

The Remix IDE also supports features such as recording transactions which can then be replayed in different environments. This is useful from a testing perspective as you can record transactions in the browser based private blockchain and then deploy to a more realistic blockchain and rerun the transactions to see if they behave as expected [63]. Remix IDE also supports debugging of smart contracts and the debugger behaves much like a conventional debugger, allowing developers to step through the code of a smart contract, inspect state and add breakpoints [62].

The Remix IDE contains a lot of features which are useful for smart contract development. However, there are a few issues with Remix. Firstly, it has no native

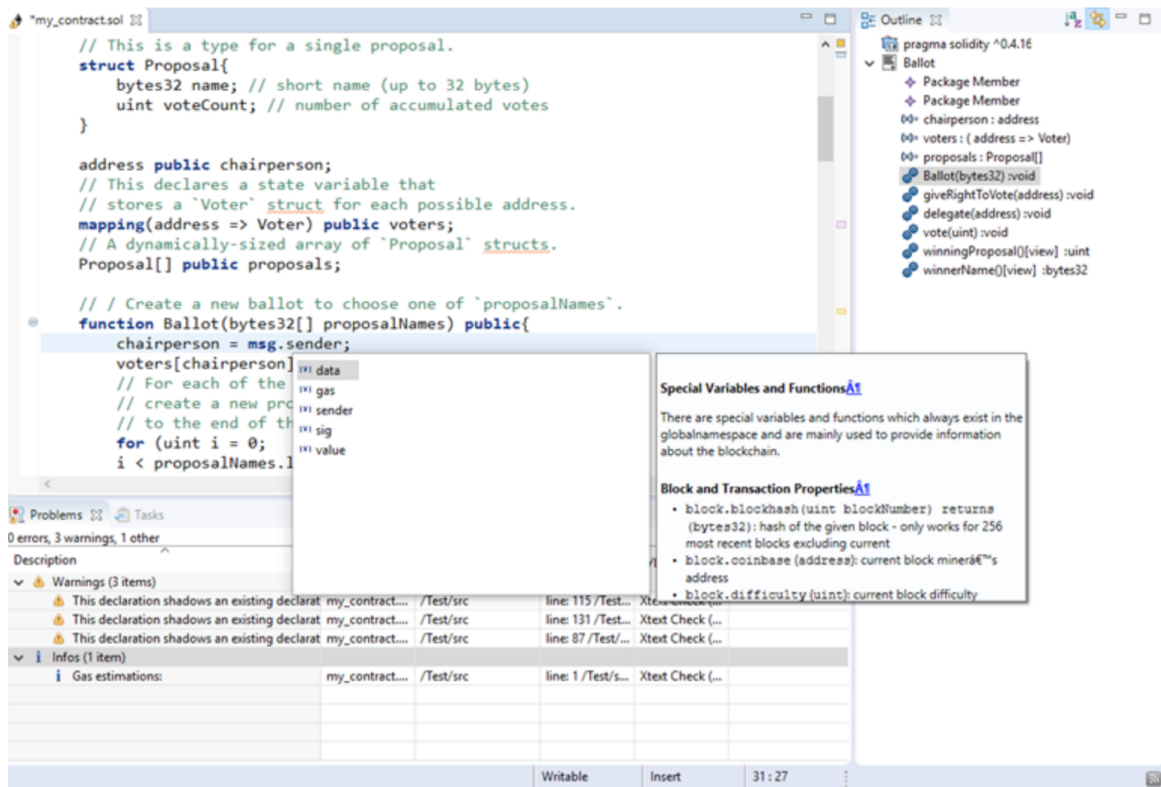


Figure 2.10: Yakindu IDE [95]

integration with version control tools or the file system, users have to install additional tools to integrate with the file system with the IDE [30]. Secondly, they have chosen to create an IDE from scratch rather than extending a widely used existing editor. This means they lose access to the rich set of plugins that have already been developed for these editors. They also provide no plugin system for Remix, thus developers cannot extend it for their uses. Finally, the IDE does not integrate automatically with Truffle. We believe the Remix IDE serves more of a playground for smart contract developers, enabling them to learn about Solidity and smart contracts in an environment that is quick to set up. For serious development, programmers will most likely use Truffle in tandem with an editor that provides language support for Solidity.

2.2.5.3 Yakindu

Yakindu is a professional IDE for solidity development [95]. Figure 2.10 is a screenshot of the tool. We will list some of the pertinent features of the tool:

- Code Completion
- Code Navigation
- Live Code Validation
- Code Formatting

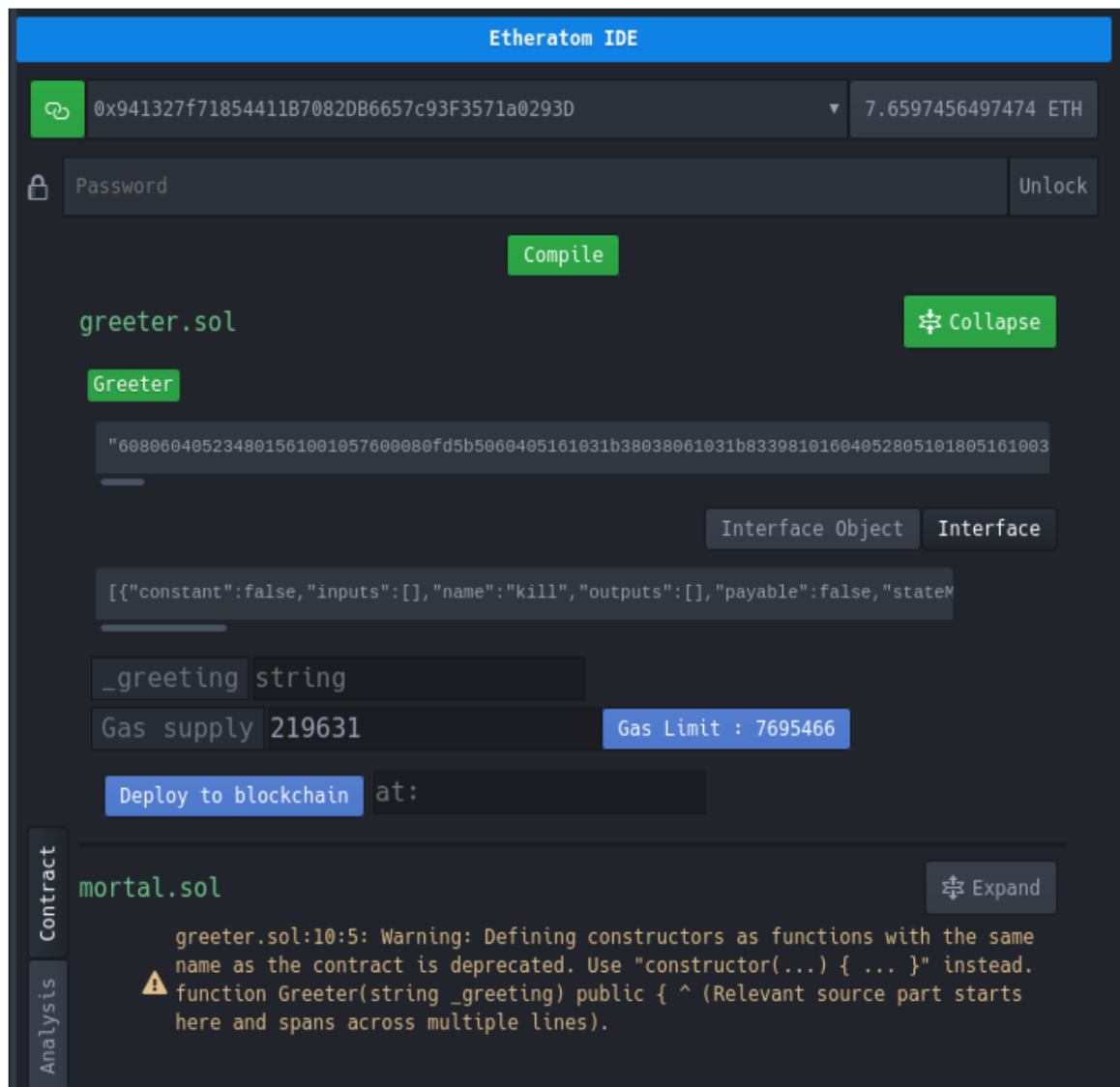


Figure 2.11: Etheratom [25]

2.2.5.4 Etheratom

Etheratom is a plugin for the Atom editor. It provides support to deploy and compile contracts straight from the editor. Figure 2.11 is a screenshot of the interface the editor provides to the user to deploy a contract from the editor. It requires the user to install a private local blockchain and configure the tool to connect to the blockchain.

2.3 Contract Analysis

Static analysis tools feature prominently in many development ecosystems. They provide insights about the source code which may not be immediately obvious to the developer. They can highlight potential security concerns, code smells and provide



Figure 2.12: Solgraph [61]

a way for a user to understand their contract. Since smart contracts cannot be modified once they have been deployed, it is useful to have tools which provide insights about the contract you are developing to help the developer notice issues before deployment. In this section, we will survey some of the tools present in the Solidity ecosystem that perform analysis of contracts

2.3.1 Remix IDE analysis

The Remix IDE statically analyses Solidity contracts and presents this information to the user. It highlights issues with the code and helps enforce best practices. It does this by notifying the user when it detects any potential security issues e.g. a developer is using inline assembly. It also looks for Solidity code smells e.g. a function not being marked as constant when it can be. It can also present users information about the gas economy of their contract e.g. warn them that iterating over a dynamic array can potentially lead to high gas costs. Please refer to section 2.8.1 for more details on what gas is. Figure 2.8 is the complete list of analyses that the Remix IDE runs.

2.3.2 Solgraph

Solgraph is a tool which visualises the functions of a Solidity smart contract. Figure 2.12 an example of the output of the tool. It displays each of the functions in a contract and colours them to highlight what properties the function has. Figure 2.13 shows the colour scheme the tool uses. This tool provides insight to the programmer about the properties of their function and it can help them identify potential security issues. It highlights which functions send money out of the contract, which functions are payable and which functions mutate state.

Legend:

- Red: Send to external address
- Blue: Constant function
- Yellow: View
- Green: Pure
- Orange: Call
- Purple: Transfer
- Lilac: Payable

Figure 2.13: Solgraph Colourscheme [61]**Figure 2.14:** SmartCheck [67]**2.3.3 Smart Check**

Smart check is static analysis tool for smart contracts. It runs a set of predefined analyses over Solidity contracts to check for known issues. It performs a variety of analyses over the Solidity code which include checking for use of deprecated constructs, unsafe use of Solidity and checking for code which can have high gas costs. Figure 2.14 is a screenshot of the tool, it lists the issues found with the code block on the left and it highlights what the problematic lines are.

2.3.4 Formal Verification

There also exists a myriad of tools which apply formal verification techniques. They utilise symbolic execution engines and SMT solvers to analyse Solidity contracts and provide formal guarantees to the user. However, we do not create any formal

```
truffle(develop)> let instance = await MetaCoin.deployed()  
truffle(develop)> instance
```

Figure 2.15: Deploying a contract using Truffle Console [83]

```
let balance = await instance.getBalance(accounts[0])
```

Figure 2.16: Calling a function on a contract in the Truffle Console [83]

verification tools in this project as that is a project in its own right. Thus, we do not explore these tools any further.

2.4 Contract Deployment and Interaction

An important part of development is deploying and interacting with contracts. This includes both deployment to a production blockchain and also to a local blockchain. Deploying a contract is a complicated and cumbersome process. Please refer to section 2.8.3 for more details on the process. Thus, having tools which automate this process greatly enhance the developers quality of life and will enable them to quickly set up continuous deployment pipelines.

Interactive testing is an important part of development. Developers regularly run code they are developing on their local machine to verify that it is working as intended [52]. Currently, smart contracts can only be run on a blockchain. Deploying contracts and interacting with them manually is difficult and requires in depth knowledge about how the blockchain operates. Please refer to section 2.8.4 for more details on contract deployment. The ecosystem is required to provide a way for developers to carry out this process in a quick and convenient way.

We will examine some tools in the Solidity ecosystem which help developers deploy their contracts and interact with them. The purpose of this is to see what features are offered by these tools.

2.4.1 Truffle Console

Truffle provides an interactive console which comes packaged with a local blockchain. The console enables developers to compile, deploy and interact with their contracts on their local machine. It abstracts away all the complexities of the underlying blockchain and enables developers to deploy and interact with their contracts using JavaScript. Figure 2.15 is an example of how to deploy a contract using the Truffle console. Figure 2.16 is an example of how to interact with a contract that has been deployed.

2.4.2 Truffle Deploy

Truffle comes packaged with a deployment tool that can be used to deploy contracts to a production blockchain. It can handle compilation and deployment of contracts. Also, since Solidity supports imports, the deployment tool facilitates deploying multiple contracts and linking them. To use the tool, developers specify a migrations file which describes to the tool how to deploy the contracts. If the project has multiple contracts, then the file specifies the dependencies between the contracts and how to link them. Listing 2.1 is an example of a migration file. The file specifies two contracts that need to be deployed and it specifies the dependencies between them. The user also specifies in a configuration file the URL of the blockchain they wish to deploy to. To deploy the contracts, the user types the command `truffle deploy` in the root of their project.

```
1 const ConvertLib = artifacts.require("ConvertLib");
2 const MetaCoin = artifacts.require("MetaCoin");
3
4 module.exports = function(deployer) {
5   deployer.deploy(ConvertLib);
6   deployer.link(ConvertLib, MetaCoin);
7   deployer.deploy(MetaCoin);
8 };
```

Listing 2.1: Truffle Migration File [85]

2.4.3 ZeppelinOS

ZeppelinOS is a smart contract programming platform which automates the process of deploying code to the blockchain. It also enables developers to link their smart contract to other contracts that have been deployed to the blockchain. It manages all this complexity and presents the developer a simple interface to deploy their contracts and link to existing smart contracts. It also has features which enable developers to publish their contracts as packages to be used by other smart contract developers using the platform. To deploy a contract, you first add the contract to the project using the command `npx zos add <contract name>`, you then use the `session` command to connect to a blockchain and then run `push` to deploy.

2.5 Testing

2.5.1 Importance of Testing

As contracts cannot be modified once uploaded to the blockchain, it is imperative that developers can employ mechanisms to ensure their code is correct. Correctness can be ensured in two ways, developers can utilise formal verification or some form of testing. Formal verification is a way of creating a mathematical proof which guarantees the correctness of your program, there currently exists no verification tools for Flint [66]. Formal verification can provide some absolute guarantees about the

correctness of a program. However, we argue that it is also useful to have automated unit tests when developing a contract. The main reasons for this are, developers are used to testing in this manner and it is something that has been very widely adopted in industry. Formal verification tools have limitations in what they can express and what properties they can prove. The Solidity compiler comes with an SMT solver which supports a subset of the Solidity language and is limited in what it can express [5]. Thus, we believe that a unit testing framework is an important part of the smart contract developers arsenal as it provides them with a familiar mechanism to write a variety of tests to validate the correctness of their code. For the remainder of this section, we will explore how to test Solidity smart contracts.

2.5.2 Testing Smart Contracts

```
1 contract('Counter', ([owner]) => {
2   let c;
3
4   beforeEach('setup contract for each test', async function () {
5     c = await Counter.new();
6   })
7
8   it('test increment', async () => {
9     await c.increment();
10    let val = await c.getValue();
11    asset.equal(1, val, "should be 1");
12  });
13 });
```

Listing 2.2: Truffle JavaScript test

```
1 contract TestCounter {
2
3   function testIncrement() public {
4     Counter counter = Counter(DeployedAddresses.Counter());
5
6     counter.increment();
7
8     Assert.equal(counter.getValue(), 1, "Should be 1");
9   }
10 }
```

Listing 2.3: Truffle Solidity Test

Truffle which is currently the official developer ecosystem for the Solidity programming language enables developers to write tests in two languages; Javascript and Solidity. Test written in Solidity are run directly on the blockchain, thus are subject to the same limitations as normal smart contracts. Tests cannot exceed a certain size because of block limits, there is no easy way log during a test and you cannot verify if an event has fired as this information is not made available within the blockchain. The intention of writing Solidity tests is to run your tests in the same environment that the contract is going to be executed in, this is useful when you are creating a smart contract that is going to be consumed by other smart contracts

Framework	JavaScript Tests	Solidity Tests
Waffle	Yes	No
Embark	Yes	No
Truffle	Yes	Yes
Etherlime	Yes	No
DApp	No	Yes

Table 2.1: Smart Contract Testing Frameworks Testing Support

[88]. Listing 2.3 is an example of a test written in Solidity.

Tests written in Javascript are run externally to the blockchain and make use of the web3 library to communicate with the blockchain using RPC. The intention of Javascript testing is to exercise your smart contract as an external application (decentralised application). The tests are run using node.js [88]. Listing 2.2 is an example of a test written in JavaScript.

2.5.3 Testing Language

Table 2.1 lists a selection of popular smart contract programming frameworks and details what languages they support testing in. Popular smart contract programming libraries such as OpenZeppelin write all their tests in JavaScript [55]. Additionally, we have found that many of the tutorials for smart contract development focus primarily on writing JavaScript tests.

The popularity of writing JavaScript tests is due to a few reasons. It is a very popular language amongst developers, more than 49% of developers surveyed in StackOverflow’s 2018 Developer Survey reported they use the language regularly [72]. Thus, many developers are familiar with the JavaScript and the node ecosystem.

Additionally, testing in JavaScript means you are not subject to the limitations of running code in the blockchain. A notable limitation is the fact that the tests cannot verify if events have fired because event information is not made available to code executing within the blockchain [23]. Please refer to section 2.7.1.2 for more information about blockchain events. Due to the popularity of testing in JavaScript, there exists many npm libraries which make it easier to write tests for your contract. An example of a popular library is the `truffle-assertions` library, it provides developers with the ability to test for exceptions and events. However, a fundamental issue with working with JavaScript is the language itself. The language has a few issues which make working with it difficult such as no static types, confusing equality semantics and weak typing [77].

2.5.4 Truffle Test

As mentioned before, the Truffle testing framework supports writing tests both in JavaScript and Solidity. Alongside providing support for testing of contract functions, it also supports testing for exceptions and testing Ether transactions. We discuss Ether in more detail in section 2.8.1. With support from external libraries, tests written in JavaScript can also support testing events and exceptions. We will not examine any other testing frameworks in the Solidity ecosystem, they all offer similar features.

2.6 Code coverage

Code coverage measures how much of a codebase is covered by a set of tests. It helps programmers find parts of their codebase which are untested. Untested code is dangerous code [75]. This is especially important for smart contracts because they cannot be modified once they have been deployed to the blockchain. Additionally, high code coverage can be an indicator that your source code is reliable. However, code coverage should not be the only metric to measure the reliability of a codebase [12].

2.6.1 Methods to implement code coverage

Code coverage is implemented by adding instrumentation code that tracks what has been executed, this information is then collected at the end of execution and collated into a report. The instrumentation code can be added to the generated assembly code or to the high level source code. The benefit of implementing it at the assembly level is that you can reuse the same tool for any language that targets that assembly. However, instrumenting assembly is an arduous process and some information from the high level source code is lost [10].

2.6.2 Types of code coverage

2.6.2.1 Line Coverage

Line coverage is a metric which measures how many executable lines were run over the total number of executable lines. It is a good indication of test coverage over a codebase but value of the metric is dependent on the developers coding style [3]. Consider Listing 2.4, a tool measuring line coverage would report that 50% of the executable lines are hit, this is an accurate reflection of the number of lines that are executed. Listing 2.5 is the same piece of code but written using a different style, a tool measuring line coverage would report 100% line coverage. This is not an accurate reflection of the codebase because the else block is never executed. To overcome this issue, instead of measuring line coverage, tools can measure statement coverage or block coverage. Statement coverage measures how many executable statements out of the total number of executable statements were executed. A statement is

a single piece of executable code, in Flint this would be assignment expressions & function calls. Block coverage looks at the total number of blocks that were executed over the total number of executable blocks. A block is defined as a sequence of statements which contain no branches, a branch defines the start of a new block.

```

1  contract Counter {
2      // removed
3  }
4
5  Counter :: (any) {
6      public init() {}
7
8      public func test() {
9          if (true) {
10             // do something
11          } else {
12             // do something else
13          }
14      }
15 }

```

Listing 2.4: Line Coverage Example 1

```

1  contract Counter {
2      // removed
3  }
4
5  Counter :: (any) {
6      public init() {}
7
8      public func test() {
9          if (true) { // do something } else { // do something }
10      }
11 }

```

Listing 2.5: Line Coverage Example 2

2.6.2.2 Branch Coverage

Branch coverage measures the total number of branches that were executed compared to the total number of branches that are theoretically possible in the program [2]. To illustrate what this means and how branch coverage differs from block coverage and statement coverage, we will look at a small example. Listing 2.6 is a piece of code where a branch coverage tool would report that 50% of the branches have been hit. This is because this code has two branches, there exists an implicit else branch for the if statement. Block or statement coverage would report 100% for this piece of code. Branch coverage looks at all the possible branches not just the ones that are visible in the sourcecode.

```

1
2  public func test() {
3      if (true) {
4          let x : Int = 1

```

```
5     }  
6 }
```

Listing 2.6: Branch Coverage Example 1

2.6.2.3 Condition Coverage

Condition coverage looks at all boolean expressions including boolean sub-expressions and checks if they have taken on all possible values. Consider Listing 2.7, to achieve 100% condition coverage then all four combinations of a and b need to be exercised. That is the program needs to have TT,FF,TF,FT to achieve 100% condition coverage. It differs from branch coverage in that it considers boolean sub-expressions and it also considers boolean expressions that are not part of a branching statement [13, 3].

```
1  
2 public func test(a : Bool, b : Bool) {  
3     if (a || b) {  
4         // code  
5     }  
6 }
```

Listing 2.7: Condition Coverage Example 1

2.6.2.4 Path Coverage

Path coverage is the most complex form of coverage and it works over the state machine that a program represents. Path coverage is a measure of how many execution paths were taken over all the possible execution paths in a program. It is the most complex form of coverage to implement as programs can have a huge number of paths, potentially infinite [13]. Generally, when working over a program state space you move into the formal verification space where techniques are applied to handle the large state space.

2.6.2.5 Function Coverage

Function coverage measures the total number of functions that were called over all the functions that exist in the program.

2.6.3 Existing Tools

We will now survey some tools that can be used to measure coverage in Solidity smart contracts.

2.6.3.1 Solidity-Coverage

solidity-coverage is a tool which can be used to measure the code coverage for Solidity contracts. It instruments the contract using events and generates a code coverage

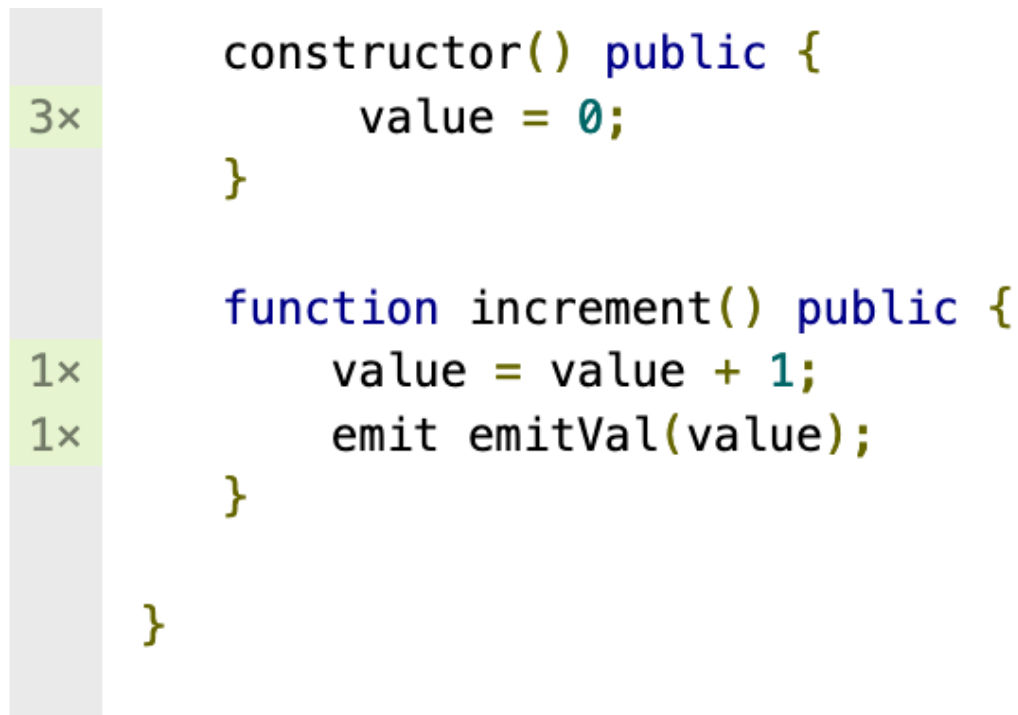


Figure 2.17: Istanbul Code Coverage Report

100% Statements 3/3 100% Branches 0/0 100% Functions 2/2 100% Lines 3/3

Figure 2.18: Istanbul Code Coverage Summary Tab

Modifier	Effect
public	Both accessor and mutator created
visible	Only accessor created (variable can be viewed by anyone)
private	Nothing is synthesised (can be manually specified)

Table 2.2: Flint Contract Variable Visibility Modifiers

report using Istanbul, which is a JavaScript code coverage tool. Please refer to sections 2.7.1.2 for more details on blockchain events. Although, Istanbul is designed to provide test coverage for Javascript files, it supports creating reports for other languages. The user can provide the tool with a JSON object containing the coverage data which it uses to produce a HTML coverage report [41, 64]. Figure 2.17 is an example of a code coverage report generated by Istanbul. The report contains statement coverage, branch coverage and function coverage. Figure 2.18 is an example of the summary tab created by Istanbul.

2.6.3.2 Sol-Coverage

Sol-Coverage is another tool that measures code coverage for Solidity contracts. Similar to *solidity-coverage* it generates an Istanbul code coverage report. However, it differs in how it collects the data to generate a code coverage report. Instead of instrumenting the source code of the contract, it monitors the transactions that are sent to the blockchain whilst the tests are running. Please refer to section 2.8.1 for more details on transactions. For each transaction, it requests a trace of that transaction from the blockchain. The trace of a transaction lists all the assembly instructions that were executed as part of that transaction. The tool then uses this data in combination with the source code of the contract to generate the coverage report [1].

2.7 Flint

Flint is a high level smart contract programming language designed with safety in mind. It has features to restrict access to functions, supports immutability and has features to prevent accidental loss of currency. It is a statically typed imperative language and takes a lot of inspiration from Swift. It natively supports Integers, Booleans, Addresses (Ethereum Addresses), Strings, Arrays (dynamic and fixed) and Dictionaries. It supports user defined types by means of Structs. Flint also supports range types which can be used to specify a range of numbers. Contracts are defined in an object oriented fashion containing fields and function declarations [31].

2.7.1 Anatomy of Flint Contract

Listing 2.10 is an example of a Flint contract. Line 1 - 4 is the contract declaration. This declares a new contract called `Counter` which contains two variable declarations and an event. We will discuss events in section 2.7.1.2. Variable declarations

in Flint need to be marked with a mutability modifier. Variables marked with `let` are immutable, they cannot be changed after they have been assigned a value. Variables marked with `var` are mutable. Contract variables can also have an optional visibility modifier. Table 2.2 lists all the visibility modifiers and their effect. The default modifier is `private`. Depending on the visibility modifiers specified, the Flint compiler can automatically synthesise accessors and mutators for contract fields.

Lines 8-21 define the contract behaviour declaration for the contract `Counter`. It is here where the contract functions are defined. Every contract must specify an `init` function, this is the constructor function for the contract. The signature of the contract behaviour declaration specifies any protections that are applied to the functions within the block. We explore the protections Flint offers in sections 2.7.2 & 2.7.3.

2.7.1.1 Functions

Functions in Flint are declared using the `func` keyword. They can have parameters and return values. By default contract functions have a `private` visibility modifier, this means the function can only be called from within the contract. However, if the function is marked with a `public` modifier then the function can be called by anyone. Functions which mutate the state of the contract must be explicitly marked with the `mutating` keyword.

Functions can also be marked with attributes using the syntax `@:<attribute name>`. Attributes provide metadata about the function. Currently, the only supported attribute is `payable`. Functions marked with this attribute can be sent Ether, functions which are marked `payable` are required to add a parameter of type `Wei` which is marked with `implicit` to their signature. The function can access the `Wei` sent to it via this parameter. The function `acceptMoney` is an example of a function which is `payable`.

A contract can also define a fallback function, this function is called when a transaction is sent to this contract but no function is specified. The syntax for a fallback function is shown in Listing 2.8. A fallback function cannot have any parameters and it cannot return any values.

```
1 public fallback() {  
2     // code  
3 }
```

Listing 2.8: Flint Fallback Function

2.7.1.2 Events

Flint contracts can define events. The event has to be declared within the contract declaration. Line 5 in Listing 2.10 is an event declaration. Events can have parameters. The keyword `emit` is used to log an event in the blockchain. Refer to section 2.8.5 for a more in depth explanation of what events are in the blockchain.

2.7.1.3 External Calls

Flint contracts can make external calls to other contracts on the blockchain. To call an external contract, the developer has to first specify an interface for the external contract. The interface is specified using Solidity types. The return values from an external call are also Solidity types and need to be casted back into Flint types. When calling functions on external contracts, the user can specify two hyper parameters: gas & value. The gas hyper parameter specifies how much gas the users wishes to allocate for this call. The value hyper parameter specifies how much Ether the user wishes to attach to this call. Ether can only be sent to functions marked with the payable attribute. Listing 2.9 is an example of how to use external calls in Flint.

```

1  // interface for external calls
2  external trait ExtContract {
3      func getNum() -> int256
4  }
5
6  contract Counter {
7  }
8
9  // calling the external calls
10 Counter :: (any) {
11     public func callExt(externalAddress: Address) {
12         // instantiate an external contract
13         let extInstance = ExtContract(address: externalAddress)
14
15         // calling method on external contract with hyper parameters
16         // result of call is casted to Flint type
17         let val : Int = (call(gas: 1000)! extInstance.getNum()) as! Int
18     }
19 }
20 }

```

Listing 2.9: Flint Contract Using External Calls

```

1  contract Counter {
2      let owner : Address
3      var value : Int = 0
4
5      event CounterVal(val : Int)
6  }
7
8  Counter :: (any) {
9      public init(owner: Address) {
10         self.owner = owner
11     }
12
13     @payable
14     public func acceptMoney(implicit w : Wei) {
15         // body
16     }
17
18     public func getValue() -> Int {
19         return value
20     }
21 }

```

```

20   }
21
22   mutating public func increment() {
23       self.value += 1
24       emit CounterVal(val: self.value)
25   }
26 }

```

Listing 2.10: Flint Counter Contract

2.7.2 Caller Protections

Flint supports caller protections, this enables developers to restrict access to functions based on the address of the caller. Listing 2.11 is an example of a Flint contract which uses caller protections. Caller protections are specified in the header of a contract behaviour declaration. Line 12 defines the caller protection `owner`, this enforces that the only address that can call this function is the one stored in `owner`. Table 2.3 lists all the ways one can specify a caller protection in Flint [32]. A contract behaviour declaration can have multiple caller protections, the calling address has to satisfy each of the caller protections. Caller protections are statically checked by the Flint compiler. However, Flint also supports dynamic checking of caller protections where the developer can delay the checking of caller protections till run time. This can be done by using the `try` keyword.

```

1  contract Counter {
2      let owner : Address
3      var value : Int = 0
4  }
5
6  Counter :: (any) {
7      public init(owner: Address) {
8          self.owner = owner
9      }
10 }
11
12 Counter :: (owner) {
13     public func increment() {
14         self.value += 1
15     }

```

Name	Flint Type	Condition
Predicate	Address -> Bool	Check if calling address satisfies a predicate
0-ary Function	() -> Bool	Satisfy 0-ary predicate
Field (Simple)	Address	Check calling address is equal to field
Field (Array)	[Address] or Address[n]	Check calling address is in list
Field (Dict)	[T : Address]	Check calling address is in values
Any	any	Always true

Table 2.3: Types Of Flint Caller Protections [32]

16 }

Listing 2.11: Flint Contract Using Caller Protections

2.7.3 Type states

Flint supports partitioning a contract into a set of states and restricting access to functions depending on the state of the contract. Listing 2.12 is an example of a contract that utilises type states. The contract declaration defines all the type states that the contract can be in. Similar to caller protections, each contract behaviour declaration defines the state protections. The `become` statement is used to change the state of a contract. Similar to caller protections, type state protection is checked statically but can also be checked dynamically. A contract behaviour declaration can specify multiple type states it is valid in.

```

1  contract Counter (State1, State2) {
2      var value : Int = 0
3  }
4
5  Counter @(State1) :: (any) {
6      public init() {
7          become State1
8      }
9
10     mutating public func s1() {
11         become State2
12     }
13 }
14 }
15
16 Counter @(State2) :: (any) {
17     mutating public func s2() {
18         become State1
19     }
20 }
21 }
```

Listing 2.12: Flint Contract Using Type States

2.7.4 Structs and Enums

Structs provide a mechanism for Flint developers to create user defined types. They can be used to group fields and methods together. Listing 2.14 is an example of a Flint struct. Methods or fields in a struct are not protected by visibility modifier as they can only be accessed by contracts within the same source file. However, if a struct method mutates the state of a struct then it needs to be marked as mutating. Structs can be function parameters but can only be passed by reference. The `init` function is used to instantiate a struct. A struct can specify multiple `init` functions. The fields in a struct have to be initialised within an `init` function or as part of the declaration for the struct. In Listing 2.14, the `country` field is declared within

the declaration of the struct and the other two fields are set within the `init` function.

Flint developers can also define enumerations. Listing 2.13 is an example of a Flint enumeration. Enumerations have an associated type with them which is used to give `rawValues` to each case within the enumeration. The `rawValues` can be automatically inferred by the compiler or specified by the developer. Currently, Flint only supports enumerations with basic types.

```
1 enum Num : Int {
2     case one
3     case two
4 }
```

Listing 2.13: Flint Enum [32]

```
1 struct Person {
2     let country : String = "UK"
3     let age : Int
4     let name : String
5
6     init(age: Int, name: String) {
7         self.age = age
8         self.name = name
9     }
10
11     func name() -> String {
12         return name
13     }
14 }
```

Listing 2.14: Flint Struct

2.7.5 Traits

Flint supports traits for structs and contracts. Traits describe the partial behaviour of the contracts and structs that conform to them. Contracts and Structs can implement multiple traits. Listing 2.15 describes how traits are implemented by contracts and structs.

```
1 contract <name> : <trait_1>, <trait_2>, ... {//code}
2
3 struct <name> : <trait_1>, <trait_2>, ... {//code}
```

Listing 2.15: Implementing traits [32]

2.7.5.1 Contract Traits

Contract traits can consist of functions and function signatures which are specified within a contract behaviour declaration. Contract traits can also contain events. Listing 2.16 is an example of a contract trait from the Flint language documentation. Any contract that implements this trait will be required to implement the

functions `getOwner` and `setOwner` with the same protections applied. Implementing contracts, will have access to the events declared in the trait and the methods `renounceOwnership` and `transferOwnership`. Listing 2.17 is an example of a contract implementing this trait from the Flint language documentation. The polymorphic type `self` represents the type of the contract that is implementing this trait.

```

1 contract trait Ownable {
2   event OwnershipRenounced(previousOwner: Address)
3   event OwnershipTransferred(previousOwner: Address, newOwner:
4     Address)
5
6   self :: (any) {
7     public func getOwner() -> Address
8   }
9
10  self :: (getOwner) {
11    func setOwner(newOwner: Address)
12
13    public func renounceOwnership() {
14      emit OwnershipRenounced(getOwner())
15      setOwner(0x000...)
16    }
17
18    public func transferOwnership(newOwner: Address) {
19      assert(newOwner != 0x000...)
20      emit OwnershipTransferred(getOwner(), newOwner)
21      setOwner(newOwner)
22    }
23  }

```

Listing 2.16: Contract Trait in Flint [32]

```

1 contract ToyWallet: Ownable {
2   visible var owner: Address // visible automatically creates
3     getOwner
4   // Skipping initialiser not relevant for this example
5 }
6
7 ToyWallet :: (getOwner) {
8   func setOwner(newOwner: Address){
9     self.owner = newOwner
10  }

```

Listing 2.17: Flint Contract Implementing Trait [32]

2.7.5.2 Struct Traits

Struct traits are declared using `struct` keyword. They currently support functions, function signatures, `init` functions and `init` function signatures.

2.8 Execution of Smart Contracts

In this section, we will discuss the execution environment of smart contracts.

2.8.1 Ethereum Platform

The Ethereum platform supports the execution of decentralised programs known as smart contracts. Execution is done in a decentralised and trustless manner. The platform uses a blockchain to achieve this, the blockchain is a cryptographically secure transaction based state machine.

Users of the network can deploy and interact with smart contracts as well as other users. Both users and smart contracts are identified using addresses, which are 160-bit integers. There are two types of accounts in the Ethereum blockchain: External Owned Accounts (EOA) and Contract Accounts. EOAs are owned by users in the real world and contract accounts are associated with a smart contract. Each account has four fields associated with it:

- **nonce:** For EOAs, this is the number of transactions sent from this address. For contract accounts, this is the number of times this address has created a contract.
- **storageRoot:** Hash of the root node of the trie that represents that encodes this account's internal storage
- **balance:** Amount of Wei associated with this account.
- **codeHash:** A hash of the EVM bytecode associated with this contract.

The `storageRoot` and `codeHash` fields are only relevant to Contract Accounts. The `storageRoot` is the hash of the storage for the associated smart contract. The `codeHash` is a hash of the EVM code of the associated smart contract [94]. The `codeHash` cannot be modified, thus once a contract is deployed to the blockchain, its source code cannot be changed.

The underlying blockchain provides a mechanism for users to engage in trustless transactions. Users can call functions on a smart contract by sending a request to a particular address, similar to sending a web request, these requests are known as transactions. The transactions are processed by miners who participate in the network. Each transaction costs the user who generated it, Ethereum defines its own currency which is known as Ether. The smallest denomination of Ether is Wei.

Transactions are aggregated into blocks and each time a new block is generated, it is appended to a data structure called the blockchain. The blockchain is an append only data structure. Blocks are created by miners, when a miner creates a block, it publishes this information and all the nodes run a consensus algorithm to decide if the block is to be accepted. If it is then it is appended to the blockchain and the entire

Field	Role
gasLimit	Maximum committed spend in terms of gas for transaction
gasPrice	Price per unit of gas in Wei
init/data	Used to either initialise or communicate with a contract
value	Optional Wei amount sent to recipient
to	Recipient address
from	Source Address

Table 2.4: Transaction Fields (excluded certain fields)

Field	Role
status	0x0 if transaction was succesful, otherwise 0x1
logs	Event logs
logsBloom	Bloom filter for event logs
contractAddress	Address of newly deployed contract
to	Recipient address (0x0 if contract creation)
from	Sender address

Table 2.5: Transaction Receipt Fields (Excluding Certain Fields)

system advances in state. To control the block creation rate and prevent miners from overloading the system, blocks are formed using a proof of work algorithm. Miners have to solve a computationally hard problem to create a block. Miners compete to create blocks and the first one to be accepted by the network wins. As the Ethereum blockchain requires miners to operate, all transactions carry an associated computational cost known as gas. To submit a transaction to the blockchain, users have to pay a gas cost. The gas cost reflects the nature of the computation in the transaction, the more computation a transaction carries out then the higher the gas cost. Gas is purchased using Ether. The Ether value for the gas is rewarded to the miner who processes the transaction and has their block accepted by the network. Deploying a smart contract and interacting with a smart contract all carry an associated gas cost.

2.8.2 Transactions

Transactions are cryptographically signed instructions. They are used to deploy and interact with smart contracts. Table 2.4 describes the salient fields in a transaction. The **to** and **from** fields define the source and destination address of a transaction. Contract account addresses can never be in the **from** field as smart contracts cannot generate transactions. The **data** field is the data payload for a transaction, transactions being sent to a smart contract define the operation to be carried out on the smart contract in this field [94]. We will describe this in more detail in section 2.8.4.

As mentioned before, computation in the Ethereum blockchain cost gas. Users specify in the transaction, the amount of Ether they are willing to pay for gas using the **gasPrice** field. Users can also specify a **gasLimit** which is the maximum amount

of gas they are willing to spend on this transaction. Users have to pay upfront the amount of gas specified in `gasLimit`, any gas which is not used is refunded. The `value` field defines how much Wei is sent to the recipient [94].

When a transaction is processed by the blockchain, a transaction receipt is generated. Table 2.5 describes some of the important parts of a transaction receipt. The transaction receipt contains information about the transaction and indicates whether the transaction was successful or not. If the transaction was a contract creation transaction, it also contains the address of the newly deployed contract in the `contractAddress` field.

2.8.2.1 Cost Of Instructions

Every EVM instruction that is run on the blockchain costs a certain amount of gas to run. Depending on the instruction the amount of gas required changes. Out of the instructions that are likely to be used regularly within a smart contract, instructions that interact with contract storage are the most expensive. Storage in this case refers only to contract storage (memory) and not the stack. The `store` instruction which is used to store data in memory costs approximately 20,000 gas to run, which is around 6000x more expensive than a basic arithmetic instruction. Reading from storage costs approximately 200 gas, it is a lot cheaper than storing but still very expensive compared to other instructions. Thus, it is likely that the cost of execution for a smart contract will be dominated by instructions that interact with storage. It is in the interest of the programmer to limit how much they interact with storage. [94].

2.8.3 Contract Deployment

To deploy a contract to the blockchain, user are required to submit a contract creation transaction. To submit a contract creation transaction, users are required to submit a transaction to the blockchain where the `to` field address of the transaction is set to zero. The initialisation code for the transaction is attached to the `init` field of a transaction. Any Wei sent using the `value` field is an endowment for the newly created contract [94]. The initialisation code for a contract is a hexadecimal representation of the EVM binary for the contract. Flint, compiles down to Solidity, it uses the Solidity Compiler to generate the EVM binary for Flint contracts. [31].

Figure 2.19 illustrates how the EVM binary is partitioned. The binary is split into two sections. The creation stage and the runtime stage. The creation code is only run when the contract is being deployed i.e. when the contract creation transaction is being processed. The constructor function in Solidity is only ever called during the creation of a transaction, it is not present in the code which is deployed to the blockchain. Thus, the arguments for the constructor function are appended to the EVM binary [22]. The constructor arguments are encoded using the ABI encoding which we discuss in section 2.8.4. The runtime code is the code which is run once the contract is deployed, this is the code that other users on the Ethereum platform

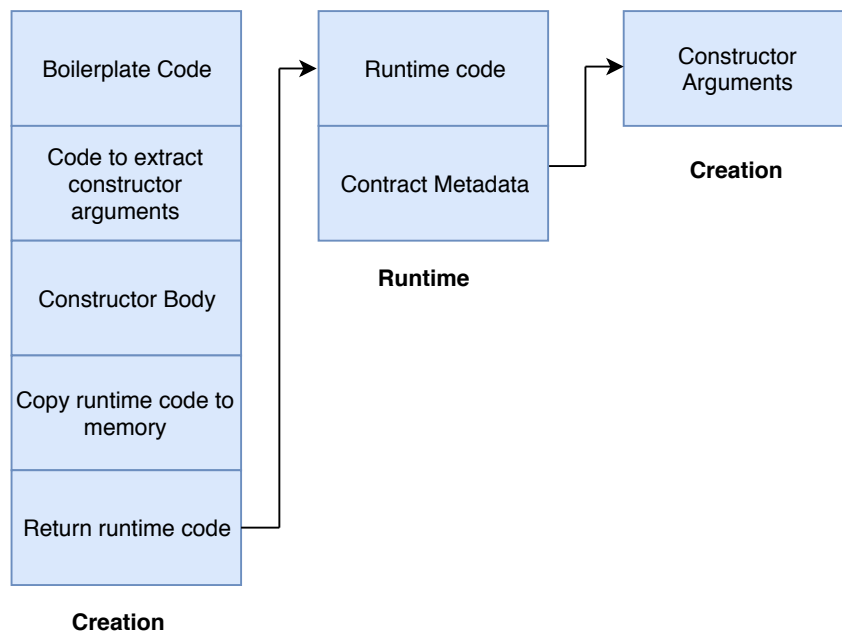


Figure 2.19: EVM Binary

interact with. It also contains some metadata which is used by the Ethereum platform to validate the contract source code.

When a contract creation transaction is submitted to the blockchain, the creation section of the binary is run. It first runs some boilerplate code which runs some basic assertions and initializes memory. It then extracts the constructor arguments, places them on the stack and runs the constructor body. The code then copies the runtime code into memory and returns it, the code is stored as part of the transaction that created the contract.

The address of the newly deployed contract is returned as part of the transaction receipt. Once a contract has been deployed, users can call functions on the contract by sending transactions to the contract address.

2.8.4 Contract Interaction

To interact with a smart contract deployed on the blockchain, users submit a transaction to the blockchain with the intended recipient being the contract address. The data field in a transaction specifies the function that is to be called on the contract. The payload is encoded using the Ethereum ABI specification [6]. The ABI (Application Binary Interface) is a protocol which defines how users can communicate with a contract.

When calling a function on a contract, the first 4 bytes of the data payload select the function to be called on the contract. Functions are selected using the Keccak hash of their signature. From the fifth byte onwards, the encoded arguments for the

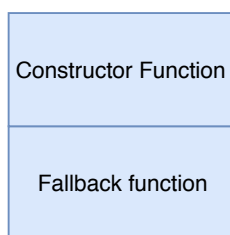


Figure 2.20: Structure Of Flint IR

function call follow [6]. If the contract function returns any values then the return values are encoded using the ABI specification. The ABI specification is also used to encode event arguments [6]. Users can also send money to a contract function by attaching Wei to the transaction using the `value` field.

2.8.5 Contract Events

The blockchain supports sending information to an external environment (outside of the network) via events. External applications can watch for events from the blockchain and react to it, this is how decentralised applications with a front end are created.

When events are emitted as part of a transaction, the EVM generates a log entry which is appended to a set of logs associated with the receipt for that transaction. Additionally, the receipt also contains a bloom filter for the log entries in that receipt. A bloom filter is a probabilistic space efficient data structure that enables you to rapidly check for the existence of an element in a set [94]. The bloom filters of all the transaction receipts in a block are combined to form a block level bloom filter [35].

The bloom filter allows for fast search of the events logs in the blockchain. Whenever an application wishes to query the blockchain for a particular event, the local node it is querying can quickly scan over all the blocks, checking the block bloom filters to see if the event logs are contained within the block. If so, then the node can search the block to extract the relevant logs. This ensures that event information can be accessed with minimal overhead.

2.8.6 Flint Code Generation

In this section, we will briefly discuss how the Flint compiler generates code and what limitations it presents on the language. Flint's IR is a Solidity contract which uses YUL assembly [31]. Figure 2.20 shows what the general structure of the generated Solidity code for a single contract looks like. The `init` function in the Flint source file is mapped to a Solidity constructor function [31]. The fallback function contains the contract functions. Once the contract is deployed, any calls to the contract are directed to the fallback function. The fallback function contains code which

calls the correct Flint contract function based on the nature of the call. Both the constructor and fallback function are wrapped in an assembly block. Each of these blocks define their own scope. This means that the `init` function in Flint cannot call any contract functions.

2.9 Conclusion

Language aware editors are an integral part of any language ecosystem, they are highly popular amongst developers and provide a myriad of features to ease the process of writing code [78]. They increase the readability of code via syntax highlighting, provide feedback from the compiler within the editor and can analyse source code to provide useful insights. Thus, the Flint ecosystem should provide the programmer with a language aware editor.

Code analysis tools feature in many language ecosystems, however they are especially prominent in the Solidity ecosystem [17]. Remix is a Solidity IDE which analyses Solidity code and helps the user enforce best practices. It warns them when they are using unsafe programming patterns, highlights potential security concerns and warns users when they write costly code [62]. SmartCheck is a smart contract auditing tool which can be used by programmers to identify issues with their contract [67]. Solgraph helps programmers visualise and identify pertinent properties of their contract [61]. These analysis tools exist to help a programmer better understand their contract and help them enforce best practices. The ecosystem should endeavour to help smart contract developers write safe and correct code as they cannot be modified once deployed. Thus, the Flint ecosystem should provide a set of analysis tools that helps programmers leverage Flint's safety features effectively. Additionally, as cost of execution is important in the blockchain programming model, the Flint ecosystem should help programmers understand the economics of their contracts.

Testing is an integral part of software development. Many programmers employ testing to increase the reliability of the software they are building [78]. Testing is especially important for the blockchain programming model because contracts cannot be modified once deployed. The Solidity ecosystem provides many test frameworks that enable programmers to test their contracts. Thus, it is important that the Flint ecosystem provides a testing framework to help developers write correct contracts.

Developers regularly run code on their local machine and interact with it. This helps them ensure that their code is working as expected. For smart contracts, this means deploying the contract to a blockchain and interacting with it. Manual deployment and interaction is long winded and inconvenient. Thus, the Solidity ecosystem provides an interactive console which enables developers to deploy and interact with their contracts in a convenient way [86]. Similarly, the Flint ecosystem should provide an interactive console which performs a similar function.

Finally, the ecosystem should provide developers with a local blockchain. Production blockchains are not suitable for development as they generally cost money to execute on.

Chapter 3

Ecosystem

We have developed an ecosystem which targets different aspects of the software engineering process. The overall goal of the ecosystem is to provide a set of tools which makes developing Flint smart contracts safer and more convenient.

Developers generally engage in writing, testing and deploying code. The ecosystem should aid the developer in all of these aspects. We have created a language server, a set of contract analysis tools, a REPL and a syntax highlighter to aid developers whilst they are writing their code. The language server provides live code validation within the editor. The contract analysis tools can run the following analyses: gas estimation, visualisation of contracts and analysis of protections used in Flint contracts. The REPL provides a convenient mechanism for developers to deploy their contracts to a local blockchain and interact with them. We have created a unit testing framework to help developers test their code. The framework can also measure test coverage of a codebase to help developers uncover untested code. We have also created a tool to help developers launch a local blockchain which is loaded with user accounts.

For the remainder of this chapter, we will discuss the syntax highlighter (flint-colour) and the tool to deploy a local blockchain (flint-block). We will also discuss how we applied the server client model when designing our tools. Chapters 4 & 5 discuss the language server and the contract analysis tools. Chapter 6 discusses the unit testing framework and chapter 7 discusses the interactive console. Chapter 8 evaluates the ecosystem and chapter 9 concludes and proposes some ideas on how to further extend the ecosystem.

3.1 Server Client Model

Each of the tools that we have designed except the syntax highlighter generate structured textual output. This output can then be consumed by another process e.g. an editor. This design philosophy has similarities with the server client model, the tools can be thought of as a server and any process which consumes the output as a client. The benefit of this approach is that it ensures our tools are not tied to any particular editor and can be ported with minimal effort between different editors. The role of

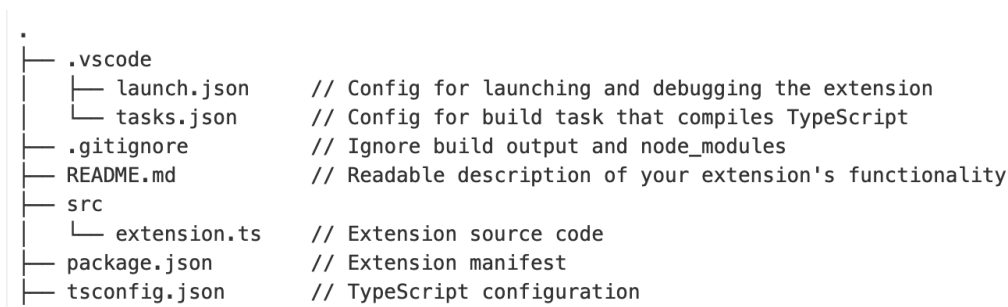


Figure 3.1: Anatomy of a VSCode extension [29]

the editor is to display the data. The majority of the processing occurs within the tools. This also ensures that we do not slow down the editor as the tools are run in a separate process.

To demonstrate features of the ecosystem which aid a developer whilst they are editing the source code we have implemented a VSCode extension. The extension interacts with the tools via the command line. We chose to use the VSCode extension framework because it is well documented and the API is highly expressive. Also, the VSCode editor supports the language server protocol. For more information on VSCode, please refer to section 2.2.

3.1.1 VSCode extension

Figure 3.1 illustrates the anatomy of a typical VSCode extension. The important files are `extension.ts` which is the source code for the extension and `package.json`, also known as the extension manifest. The extension manifest is a JSON configuration file, it defines two important fields; `activationEvents` and `contributes`. The `contributes` field defines a list of contributions, each of these extend the functionality of VSCode in some way. Using the `languages` contribution we enriched the editor with knowledge of the Flint language. This enabled us to associate `.flint` file extensions with the Flint language. Additionally, we introduced a language id for Flint, which can be used to refer to Flint in other parts of the extension. We also contributed a set of commands which were available via the VSCode command palette. These commands enabled developers to run our analysis tools from within the editor. The `activationEvents` field defines when the extension is activated by the editor, extensions can be activated in response to various events [4]. We set the `activationEvent` to be `onLanguage:flint`. This instructs the editor to launch the extension whenever a Flint file is loaded. The argument for the `onLanguage` field is the Flint language id [19]. We use the VSCode webview API to display the results of the analysis, it enables us to display HTML content to the user within the editor.

```
contract Counter {  
  var value : Int  
  event Message(msg: String)  
  let owner : Address = 0x168ffdc72126a71f3277b487a8bbdf0ccf3f28b5  
}  
  
Counter :: (any) {  
  public init(val : Int) {  
    self.value = val  
  }  
  
  mutating public func increment() {  
    self.value += 1  
  }  
  
  public func getValue() -> Int {  
    return value  
  }  
}
```

Figure 3.2: Flint Contract With Syntax Highlighting

3.2 Flint-Colour

3.2.1 Description

Syntax highlighting increases the readability of a piece of code. It accentuates language keywords and structures allowing the developer to quickly distinguish between different language features. Figure 3.2 is an example of a Flint contract with syntax highlighting on. The `mutating` keyword is colored differently to everything else, this enables a programmer to quickly see which functions mutate state. Modifiers and control keywords are coloured purple and anything that is related to storage e.g. functions, types and contracts are coloured green. This is to help programmers quickly distinguish between language structures and the properties of those structures. Literals are coloured yellow to differentiate them from other programming constructs.

3.2.2 Implementation

To implement syntax highlighting for Flint, we have created a TextMate grammar for Flint. TextMate grammars are a well established and widely adopted way of implementing syntax highlighting. Many popular editors support TextMate grammars including Sublime, Atom and VSCode [74, 20, 65]. Thus, by writing a TextMate grammar we can support syntax highlighting for Flint in many different editors [45].

TextMate grammars are composed of a series of rules, each rule specifies a regular expression which match a set of language tokens and a scope (name) for those language tokens. Listing 3.2 is an excerpt of a TextMate grammar for Flint. As you read the scope from left to right it becomes more specialized, `keyword.control` is the parent scope of `keyword.control.flint`. Scopes are used by theme files to style

language tokens. Theme files specify a series of rules which consist of a name, list of scopes that the rule is targeting and the style properties which apply to those scopes. Listing 3.1 is an excerpt of a theme file. It specifies how to colour the control keywords in flint. The scope field is used to select a list of scopes that this theme applies to, the scopes are defined in the TextMate grammar. The settings field defines a set of style properties. If a theme is not specified for a particular scope then it inherits the theme of its parent scope. When writing the grammar for the most part we used the common scopes listed in the TextMate documentation [45]. These scopes are targeted by many themes and by following these conventions then the user is free to import their own theme file and have it work with our TextMate grammar. We have written both a grammar and a theme file which has been packaged as a VSCode extension called `flint-colour`. As Flint is heavily inspired by Swift, we have taken inspiration from the Swift syntax highlighting in XCode.

```
1 {  
2   "name": "Control Keywords",  
3   "scope":  
4     ["keyword.control.flint"]  
5   },  
6   "settings": {  
7     "foreground": "#00BCD4"  
8   }  
9 }
```

Listing 3.1: Example Theme File

```
1 "patterns": {  
2   "name": "keyword.control.flint",  
3   "match": "\\b(if|while|for|return)\\b"  
4 }
```

Listing 3.2: Example Text Mate Grammar

3.2.3 Conclusion

To improve the readability of Flint code we have created a syntax highlighter. It has been designed with portability in mind thus we have chosen to implement it using a TextMate grammar. TextMate grammar's are supported by many popular editors. We have written a grammar that follows TextMate conventions thus the grammar can be targeted by many different themes.

3.3 Flint-Block

To facilitate testing, deployment and interaction with contracts on the developers machine we have created a tool which enables a developer to launch a local blockchain. This tool is called `flint-block`. The local blockchain comes pre-loaded with 5 user accounts that have a large amount of Ether. We use Geth to create the

local blockchain. Geth is a command line tool developed by the Ethereum foundation which can be used to launch a blockchain node [34].

Production blockchain such as the Ethereum blockchain carefully adjust the difficulty level to ensure that average block creation time is 12s. This is done for security reasons as it ensures that the network stays properly synchronised and makes it difficult for any adversary to maintain a fork [50]. The drawback of this is that it limits the number of transactions that can be processed per second. Additionally, on a real network the gas price is a non zero value set by the network to ensure that there is an incentive for miners to participate. For a development blockchain, these constraints must be removed to enable the developer to use the blockchain liberally. Geth supports launching a local blockchain with a custom configuration. We configured the underlying Geth blockchain so that both the initial difficulty level and gas price are set to zero.

Even though the initial difficulty is set to zero, the difficulty increases as the number of blocks increase. This happens because Geth implements the complete Ethereum protocol which automatically causes the blockchain to converge to a block creation time of 12s. This reduces the number of transactions that can be processed per second. To prevent this happening we attempted to modify the *go-ethereum* source code, but we found that we would have to change significant parts of the implementation for it to work correctly. The effect of this limitation can be mitigated if the developer periodically restarts the blockchain to prevent it from converging to a block creation time of 12s. The ideal solution would be to implement a custom blockchain node which ensures that the difficulty level is always zero. However, we did not implement this as this would require implementing a very large piece of software and was not within the scope of this MEng project. To present the reader with an idea of the size, *ganache*, the custom blockchain used by Truffle is approximately 12,000 lines of code [90].

Conclusion

Flint-block provides a quick and convenient mechanism for developers to launch a private blockchain on their local machine. It comes pre-loaded with accounts that contain Ether. It also serves as the underlying blockchain for other components in the ecosystem. This tool is an important part of the ecosystem as production block chains cost money to use, thus the ecosystem is required to provide a developer with a blockchain that is free to use.

Chapter 4

Language Server

We have created a language server to provide rich language support for Flint within the editor. In this chapter we will explore the language server in more detail, describing its implementation and explaining why it was created.

4.1 Motivation

Live code validation is a feature which is present in many language ecosystems, it usually comes as part of an IDE or can be added to an editor via an extension. Generally, this includes reporting any syntax or semantic error to the developer within the editor. Some systems also support more complex analysis of the code - they look for code smells and suggest improvements to the developer. This form of rapid feedback is convenient for the developer and improves the development experience. It saves them from having to manually compile their code to get feedback. We implemented a code validator for Flint that provides rapid feedback to the programmer within the editor as they edit code. Figure 4.1 is an example of the feedback that is presented to the user. The error message is the one outputted by the Flint compiler.

4.2 Language Server Protocol (LSP)

To develop this feature, we were presented with two options. The first was to develop the tool as an extension for a particular editor. The second was to develop a

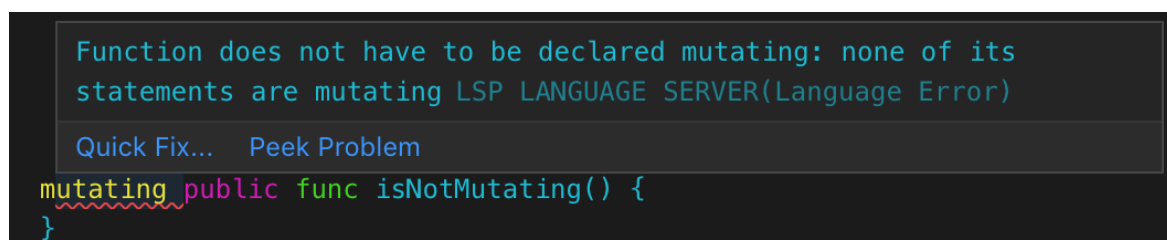


Figure 4.1: Feedback From The Code Validator

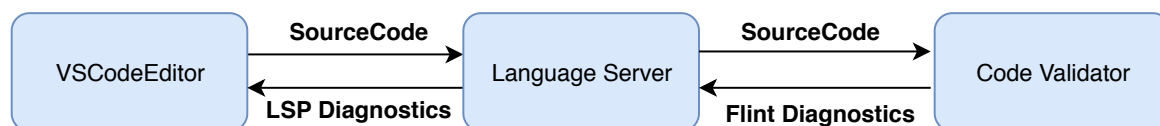


Figure 4.2: System Diagram For Live Code Validation

tool which conforms to the LSP. Please refer to section 2.2.1 for a more detailed explanation of the LSP. We chose to implement a tool which conforms to the language server protocol. We will now list the reasons why we made this decision:

- The tool can be easily ported to any editor which supports the language server protocol. There are currently over 15 editors listed on the official LSP site that support the protocol [80].
- The protocol is expansive and supports a large number of features including code navigation, code completion and live validation of code.
- The tool can be in a separate process from the editor. This means the editor will incur no performance cost from running the tool.

4.3 Implementation

Figure 4.2 is the system diagram for the code validation feature. We have implemented a VSCode extension which is triggered whenever a Flint file is loaded into the editor. The extension launches the Flint language server which then registers a callback with the editor. The callback is triggered whenever the contents of the editor is modified, the editor sends the entire source file to the language server. The server then calls the code validator with the contents of the source file. If any issues are found, the validator sends the language server a list of Flint diagnostics. The language server converts the Flint diagnostics into LSP diagnostics and sends them to the VSCode editor. The editor then displays the diagnostics to the user.

4.3.1 Code Validator

The code validator is a modified Flint compiler which augments the Flint parser to be error tolerant and also outputs Flint diagnostics as JSON for consumption by the language server [31]. The validator looks for both syntax and semantic errors. On detection of a syntax error, the validator attempts to recover and continue parsing to look for any further syntax errors. The code is checked for semantic issues only if it does not have any syntax errors. We decided not to check syntactically incorrect code for semantic issues as this can lead to misleading error messages.

To implement error tolerance we have chosen to use Wirth's follow set method. Please refer to section 2.2.3 for more details on this method. We have chosen to implement Wirth's follow set method over the other methods because of the following reasons:

- It is more accurate than panic mode and exploits properties of the grammar to implement error recovery.
- It is fast and has a low memory footprint.
- It does not slow down the normal operation of the compiler i.e. when the file is syntactically valid.
- It can be used with a recursive descent parser

Wirth's follow set method works in a similar way to panic mode error recovery. When an error is detected, the parser continues to discard tokens until it reaches a token in the synchronisation set. The synchronisation set is a set of language tokens which when encountered during error recovery notify the parser that it can continue parsing normally again. In panic mode recovery these are terminator tokens i.e. tokens which terminate statements or code constructs. In Wirth's follow set method we exploit our knowledge of the grammar to have better sync tokens depending on the language structure the parser is processing.

We have implemented error tolerance for a subset of the Flint grammar. We have chosen to implement error tolerance for single line statements only. To support error recovery over multi line statements, a regional error recovery scheme would need to be implemented. This would introduce additional overhead for not much benefit. Regional and global error recovery schemes were developed during a time where computers were slow [37]. Thus, the feedback loop between the compiler and the developer took a long time. Therefore, the compiler outputting as many errors as it could find in one cycle would be extremely helpful for the developer. Current compilation speeds are extremely fast and most developers engage in a feedback loop where they fix one error at a time. Thus, we reasoned it was not necessary for the error recovery mechanism to span multiple lines as everytime the user fixes an error, we can generate fresh feedback. For the same reasons, we decided also not to implement an incremental parser. Techniques for incremental parsing were developed during a time where computers were limited in CPU and memory and are not really relevant today [7]. In the next section we will discuss in detail the rules that we encoded into our error tolerant parser.

4.3.2 Error Tolerance Rules

4.3.2.1 Variable Declarations

If the parser encounters a missing `:` in the type annotation of a variable declaration. It continues parsing until it encounters a basic type, array type or dictionary type. During the parsing process, the parser has no knowledge of user defined types so it cannot recognise these.

We can also handle invalid array and dictionary types in a type annotation. On detection of an invalid type during the parsing process, the parser syncs at the next `equal` token or at the next new line.

4.3.2.2 Expressions

On detection of invalid expressions that are not within brackets, we skip to the next new line. On detection of invalid expressions which are within a bracket then we continue parsing after the closing bracket.

4.4 Conclusion

We have created a tool that enables the developer to get rapid feedback on their code as they type. The tool reports both syntax and semantic errors. We have augmented the parser with error tolerance to improve the feedback given to the programmer. Additionally, we have implemented the tool using the LSP, which means that it can be ported with minimal effort to other editors that conform to the LSP.

Chapter 5

Contract analysis

We have created a set of analysis tools which help developers create safer and more economical contracts. The tools present insights to the users about their contracts. We analyse protections, visualise type states & contract functions and provide gas estimations. The analysis tools can be run from within the editor using VSCode commands. All the tools we have created are packaged within the `flint-ca` command line tool.

To support Flint developers, we have developed three analysis tools.

- Visualisation of the contract in meaningful ways.
- Analysis of contract protections.
- Gas estimation.

In subsequent sections, we will explore these analyses in more detail and discuss the motivation behind them and their implementation.

5.1 Caller & State Protection Analysis

5.1.1 Description of Analysis

Caller protections and state protections offer a useful safety net for Flint developers. The analysis tools we have developed infer the caller and state protections of a contract and display it to the user in a tabular format. The analysis highlights to the programmer which functions can be called in what state and by what callers. It also calculates as a % how many of the methods the programmer has specified fall under the any caller protection. Listing 5.1 is an example of a Flint contract which utilises Flint caller and state protections. Figure 5.1 is an example of running the protection analysis on this contract. The first table details all the caller protections found in the contract and the functions which can be called by each of the callers. The second table describes what methods can be called in what state. The calculation at the bottom is a measurement of how much of the contract code is under the any caller

Caller & State Analysis	
Caller	Functions
owner	reset,makeModifiable,makeRead
any	getValue,increment
State	Functions
Read	getValue,makeModifiable
Modify	reset,makeRead,increment
% of methods under any caller blocks: 40.0%	

Figure 5.1: Caller & State Protection Analysis Of Code In Listing 5.1

protection block.

The analysis can be run from a VSCode editor using the contract analysis command. The results are displayed within the editor.

5.1.2 Implementation

We implemented a VSCode extension that makes a new command available. When the command is run, it runs `flint-ca` passing in the flag `-c` to run the analysis over the contract source code. The `flint-ca` tool walks over the contract AST and infers the contract protections by examining each contract behaviour declaration. Whenever it encounters a contract behaviour declaration it examines the protections applied to it and checks what functions are within the declaration. The tool then generates a JSON object which contains the results of the analysis. The extension then takes this information, converts it to HTML and displays it to the user using the VSCode webview API.

5.1.3 Conclusion

Caller & state protections are an important safety feature in Flint. By restricting access to functions depending on states and callers, the programmer can limit the attack surface of a contract to external users. This analysis presents an easy way for programmers to see how well they are using Flint protections, this in turn will give them an intuition on how secure their contract is.

```

1 contract Counter (Read, Modify) {
2   var value: Int = 0
3   let owner : Address
4 }
5
6 Counter @(Read) :: (any) {
7   public init(owner: Address) {
```

```

8     self.owner = owner
9     become Read
10  }
11
12  public func getValue() -> Int {
13      return value
14  }
15  }
16
17  Counter @(Modify) :: (owner) {
18      mutating public func reset() {
19          self.value = 0
20      }
21  }
22
23  Counter @(Read) :: (owner) {
24      mutating public func makeModifiable() {
25          become Modify
26      }
27  }
28
29  Counter @(Modify) :: (owner) {
30      mutating public func makeRead() {
31          become Read
32      }
33  }
34
35  Counter @(Modify) :: (any) {
36      mutating public func increment() {
37          value += 1
38      }
39  }

```

Listing 5.1: Flint Contract With State and Caller Protections

5.2 Contract visualisation

5.2.1 Description of Analysis

We support two forms of visualisation. The first one is applicable to contracts that make use of type states. The analysis visualises the contract as a state diagram. The type states are states in the diagram and the transitions are contract functions. Figure 5.2 illustrates how this visualisation looks for the contract in Listing 5.1. The initial state of the contract is marked with a purple border.

Figure 5.3 is another type of visualisation we support, this one visualises all the functions in a contract and colours the states based on the properties of the function. Table 5.1 describes the scheme we use to colour the boxes. If the function has multiple properties, we apply the colour scheme in an additive manner. For example, in figure 5.3 the reset function is coloured both green and red and has a yellow border. This is because this function mutates state, sends money to an external address

Scheme	Property
Yellow Border	Mutating
Blue Border	Constant
Green Fill	Payable
Red Fill	Sends money to an external address

Table 5.1: Style Scheme For Function Analysis

and is payable.

These visualisation tools are exposed to the developer as a VSCode extension. The programmer can run it from the editor and the results of the analysis will be presented to the programmer within the editor.

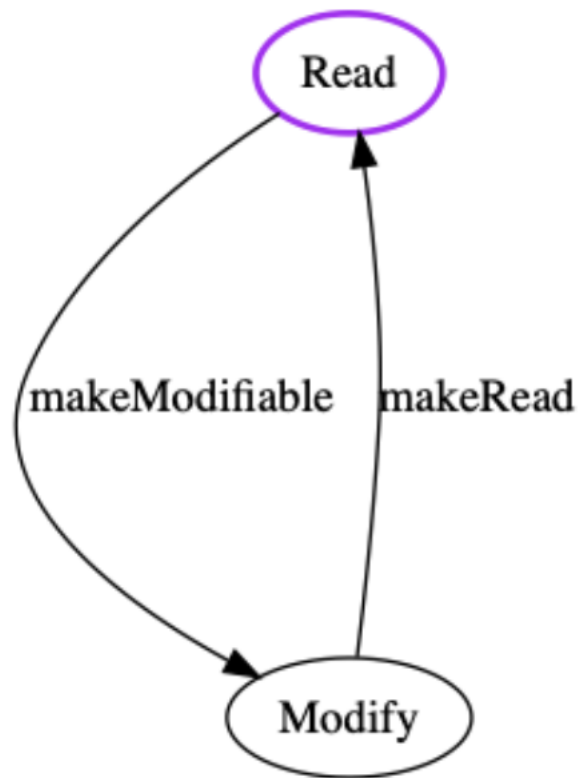


Figure 5.2: Type State Visualisation For Contract In Listing 5.1



Figure 5.3: Function Visualisation For Contract In Listing 5.1

5.2.2 Implementation

To implement the visualisation analysis, we analyse the AST of the source code and produce a dot file which represents the visualisation. We make use of GraphViz's dot program to generate the visualisation, the tool accepts a dot file, which is a textual description of a graph, and produces a graphic representing the graph [36]. Listing 5.2 is an example of a dot file. The file defines a directional graph, and lists all the transitions in the graph using the following syntax:

```
1 <starting state> -> <ending state> [<style information>]}
```

The style information is used to label transitions and theme them.

```
1 digraph {
2   graph [pad="0.5", nodesep="1", ranksep="2"];
3   Read [style=bold, color=purple]
4   Read -> Modify [label="makeModifiable"];
5   Modify -> Read [label="makeRead"];
6 }
```

Listing 5.2: Example Dot File

To create the type state visualisation, we traverse the AST of a contract to discover all of its type states. We then look for instances of the become statement and when one is encountered, we create an edge for the diagram. The starting state is the current state that the contract is in and the resulting state is the argument to the become keyword. Flint uses the become keyword to change the state of the contract. We use the initial starting state and the list of edges to generate a dot file. This dot file is then converted into a graphic using the dot tool. The generated graphic is then displayed to the user within the editor using the VSCode webview API.

To create the function visualisation, we walk the AST and examine all the functions present in the contract. We record the different properties of a function and store it in a data structure. We can infer if a function is mutating, payable or constant from the signature of the function. To check whether a function sends money to external addresses, we search the body of the function for any invocations of send. The send function is used in Flint to send money to Ethereum addresses. Upon completion of the AST walk we use the data structure that was generated to create a dot file which represents the visualisation. The dot file is then converted into a graphic using the dot tool. The generated graphic is then displayed to the user within the editor using the VSCode Webview API.

5.2.3 Conclusion

Combining the results of the two visualisations, the programmer can gain a lot of insight on the structure and properties of their smart contract. As Brett Victor mentions in his talk about learnable programming “people understand what they can see”, thus by creating tools to visualise a contract we hope to help developers understand their contracts better [47].

The function analysis enables the developer to quickly see where money leaves and enters the contract. This can be used to highlight potential security vulnerabilities e.g. money is leaving the contract in a place where it shouldn't. It also helps developers to quickly see which functions mutate state, this is a useful as functions which interact with storage are expensive. Thus, it is in the interest of the programmer to reduce the number of functions that interact with storage. Please refer to section 2.8.2.1 for a more detailed explanation on the storage costs for the blockchain.

Partitioning a contract into a set of states and restricting access to functions based on the contract state is applicable to many contract domains. It is also a very common design pattern [14, 69]. The type-state visualisation enables the developer to see their contract as a state diagram. The developer can use this to easily see the different states in their contract and how the contract moves between states. We believe this is a useful tool as it grants visibility to the developer about the behaviour of their contract. Developers can use this information to verify that the contract is working as expected.

5.3 Gas Estimation

In this section, we will discuss the motivation behind the gas estimation analysis and how it was implemented.

5.3.1 Description Of Analysis

The gas estimation analysis estimates the gas cost for each of the functions in the contract and the gas cost for deployment. Figure 5.4 is the results of the analysis when run on the contract in Listing 5.1. The table lists each function with their associated gas cost. The contract entry in the table is the amount of gas it costs to deploy the contract.

For contracts with constructor arguments, the estimate is slightly inaccurate as we do not account for constructor parameters in our estimation. The state variables of the contract are set to default values. However, this is not a significant issue as Flint only supports basic types as constructor parameters. This means that the user cannot pass in parameters which would significantly increase the cost of contract deployment. Please refer to section 2.8.3 for more information on how contracts are deployed to the blockchain. The developer can launch the gas estimation tool from the editor in a similar fashion to the other analysis tools.

Function	Gas Estimate
contract	451590
makeModifiable	42047
getValue	21683
reset	34485
makeRead	27069
increment	49673

Figure 5.4: Gas Estimates For Contract In Listing 5.1

5.3.2 Implementation

To estimate the gas of a contract, we deploy the contract and use web3 functions to estimate the gas cost of deploying the contract and calling contract functions. To estimate the gas for a contract function call, the web3 library simulates a transaction that calls the contract function. The transaction is not added to the blockchain but the simulation estimates the amount of gas it would have required for the transaction to complete. However, in order to simulate transactions, the web3 library needs to be able to call every method on the contract. This presents an issue because Flint has safety protections which restricts the callers of a function, thus to support gas estimation we pre-process the contract to remove any safety protections from the contract. This is not an issue as the contract is not reused for anything else and is only ever deployed to a local blockchain on the developers machine.

Figure 5.5 is the system diagram for the component that implements gas estimation. When the developer runs the command for gas estimation in the editor, the source code for the contract is passed onto the pre-processor. The pre-processor removes any protections from the contract and ensures that any caller can call all of the functions in the contract. We then walk over the contract AST, collect all the functions

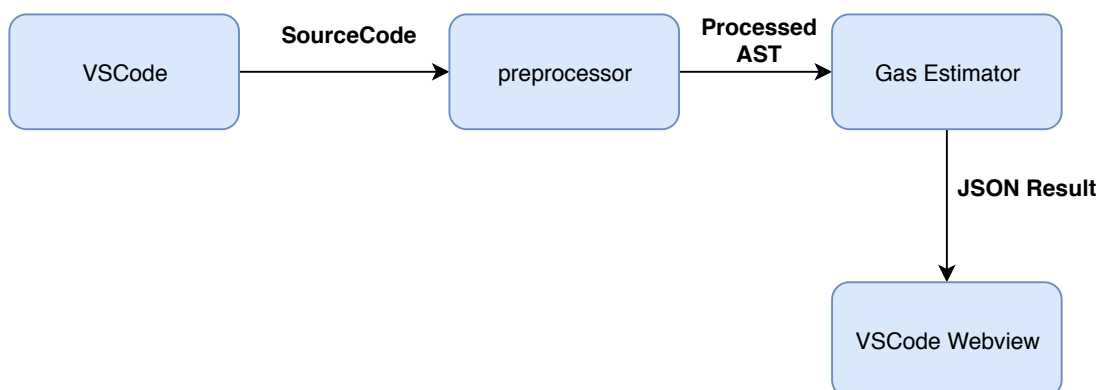


Figure 5.5: System Diagram For Gas Estimator

and then generate a JavaScript file which deploys the contract and uses web3 to estimate gas costs. The results are stored in a JSON object and passed back to the VSCode extension. The extension converts it into a HTML table and displays the results to the user using the VSCode webview API.

5.3.3 Conclusion

The gas estimation tool is a useful tool for developers as it gives them an insight on how much their contract will cost to run and deploy. Generally, it is in the interests of any developer whether they are working in industry or on a personal project to reduce the cost of deploying and operating a smart contract.

Understanding the economics of a contract also can serve as a useful guide to help a developer decide where to spend most of their time. Using the gas estimator, they can figure out what the most expensive functions in the contract are and focus their time on optimising those functions.

Chapter 6

Testing Framework

In this section we will discuss the motivations behind the testing framework, explain the rationale behind its features and describe its implementation.

6.1 Introduction

Testing is an integral part of software development. It enables developers to ensure that software they are developing is meeting the intended specification. It is especially important for smart contract development as contracts cannot be modified once deployed. Thus, developers need a way of verifying that their code is correct. Writing unit tests is one way to achieve this goal. Thus, we have created a test framework for Flint developers, the framework enables developers to write tests which exercise their contracts. The framework has been developed with Flint in mind, thus it has mechanisms to exercise all of Flint's safety features as well as events and exceptions. The test framework can also measure how well a set of tests covers a codebase. The motivation behind the coverage feature is to help developers uncover untested code. Please refer to section 2.5.2 for more information about testing smart contracts. We will discuss the code coverage feature of the testing framework later on in this chapter. Listing 6.1 is an example of a Flint test file. Figure 6.1 is the output of the test framework for this test file.

```
1 contract TestCounter {
2   let filePath : String = "<location>"
3   let contractName: String = "Counter"
4   let TestSuiteName : String = "Counter Tests"
5 }
6
7 TestCounter :: (any) {
8
9   public func test_increment_by_one() {
10    let owner : Address = newAddress()
11    let c : Counter = Counter(0, owner)
12    c.increment()
13    let val : Int = c.getValue()
14    assertEquals(1, val)
```

```
15 }
16
17 public func test_event_fires() {
18     let owner : Address = newAddress()
19     let c : Counter = Counter(0, owner)
20     c.increment()
21     assertEventFired(c.CounterVal, 1)
22 }
23
24 public func test_only_owner_can_reset() {
25     let owner : Address = newAddress()
26     let not_owner : Address = newAddress()
27     let c : Counter = Counter(0, owner)
28
29     setAddr(not_owner)
30     assertCallerUnsat(c.reset)
31     unsetAddr()
32 }
33
34
35 public func test_cant_call_increment_in_event_ready_state() {
36     let owner : Address = newAddress()
37     let c : Counter = Counter(0, owner)
38
39     setAddr(owner)
40     c.eventReady()
41     unsetAddr()
42
43     assertCantCallInThisState(c.increment)
44 }
45
46
47 public func test_exception_is_thrown() {
48     let owner : Address = newAddress()
49     let c : Counter = Counter(0, owner)
50     assertWillThrow(c.willThrow)
51 }
52 }
```

Listing 6.1: Example Flint-Test File

```
Running test suite: Counter Tests
Running test_increment_by_one
test_increment_by_one
    passed ✓
Running test_event_fires
test_event_fires
    passed ✓
Running test_only_owner_can_reset
test_only_owner_can_reset
    passed ✓
Running test_cant_call_increment_in_event_ready_state
test_cant_call_increment_in_event_ready_state
    passed ✓
Running test_exception_is_thrown
test_exception_is_thrown
    passed ✓
```

Figure 6.1: Test Output Of Framework For Listing 6.1

6.2 Anatomy Of Test File

In this section, we will cover the basic anatomy of a test file and explain at a high level how the framework operates. Listing 6.2 illustrates what a basic test file looks like. The contract declaration defines some important fields. The `filePath` variable specifies the location of the file which contains the contract that is being tested. The `contractName` field specifies the name of the contract that is being tested and the `TestSuiteName` field is used to name a group of tests.

The test functions are specified in the contract behaviour declaration. The test in listing 6.2 deploys a counter contract, stores the instance in the variable `c` and then uses this instance to interact with the contract and assert properties about it.

The test runner communicates with the blockchain over RPC using `web3.js`. Please refer to section 6.5 for more details on how the test runner operates. The backing blockchain node used by the test framework is created by `flint-block`.

```
1 contract TestCounter {
2   let filePath : String = "<contract file path>"
3   let contractName: String = "Counter"
4   let TestSuiteName : String = "Counter Tests"
5 }
6
7 TestCounter :: (any) {
8
9   public func test_counter_increment_by_one() {
10     let c : Counter = Counter()
11     c.increment()
12     let val : Int = c.getValue()
13     assertEquals(1, val);
14   }
15 }
```

Listing 6.2: Basic Flint Test File

6.3 Flint-Test DSL

We will now discuss the rationale behind choosing to create a testing framework that enables developers to write tests using a Flint-like DSL. Please refer to section 2.5.2 for more information on the languages that can be used to test smart contracts.

We made the choice to create a testing framework where the tests are written in a Flint-like DSL, which is converted into Javascript and run against the blockchain. We will now elaborate why this design decision was made. The BNF for the testing DSL can be found in the appendix (Appendix C.1). It is an adapted version of the Flint BNF. Currently, we support a subset of Flint syntax. We do not plan to support structs, enumerations, become statements, state groups, events and caller capabilities other than any as these features are not required to write tests.

Flint is strongly typed and supports immutability at the language level. We do not want to lose these benefits of Flint in our testing framework. We also believe that having the programmer write tests in the same language they use to write smart contracts is better. The programmer is not required to learn another language and it also means we can reuse tools from other parts of the ecosystem to make writing tests easier. However, as we would like to have an expressive testing framework and have the ability to test events, we decided to convert the DSL into Javascript and utilise the web3 library to interact with the blockchain. Also, by converting to Javascript we circumvent any of the limitations of running tests on the blockchain. It also means that in the future, we can integrate our test runner with the node ecosystem. This will give us access to the vast node ecosystem and allow Flint developers to use external libraries whilst testing. We believe this hybrid approach gives us the best of both worlds.

6.4 Contract Deployment

Ideally, the testing framework should enable the developer to instantiate a contract within a test and then interact with it. However, Flint does not support the instantiation of contracts as objects to be used within the code. Listing 6.3 illustrates the syntax we would like to support. We would also like to support the deployment of contracts with constructor arguments. Each test should be able to deploy its own instance of a contract. This ensures that tests are isolated from each other and do not share contract state. If this was not possible and all tests shared state then writing tests would become incredibly difficult.

```
1 let c : Counter = Counter()
```

Listing 6.3: Flint-Test DSL Deployment Syntax

A potential way to solve this issue would have been to use Flint structs as a form of indirection to contracts, this would involve pre-processing the test file to add a struct that represents the contract. This struct can then be used in place of the contract. However, the disadvantage of this approach is that we cannot accurately maintain the semantics of a contract functions in a struct e.g. we cannot support caller capabilities. The benefit of this approach would be that we could reuse the Flint compiler to process the test files.

Another approach is to create a compiler for the testing framework which is more permissible than the Flint compiler. This would enable us to support semantics within the testing framework that would not be possible in a normal Flint file. This would allow us to permit contract instantiation as part of the language. We have chosen to take this approach because it offers us the flexibility to diverge from Flint syntax and semantics when required. We use this flexibility to offer convenient syntax when testing events and instantiating contracts. This also gives us the freedom to integrate JavaScript with the testing framework thus giving developers access to the node ecosystem.

6.4.1 Constructor Arguments

The testing DSL permits a programmer to deploy a contract with constructor arguments. Please refer to section 2.8.3 for more information on how contracts are deployed to the blockchain. Due to some of the limitations of the web3 library and how Flint performs code generation, it was not possible to deploy a contract with constructor arguments using web3's contract deployment function. To overcome this limitation, the testing framework pre-processes the contract to insert a public function which can be called after the contract is deployed to initialise it with constructor arguments. Figure 6.4 is an example of how a contract looks after it has been processed. The `testFrameworkConstructor` is the function inserted by the test framework, it is called by the framework immediately after the contract is deployed. It is always placed under a contract behaviour declaration that has no protections to ensure that the test framework can always call it. Due to how Flint generates Solidity code, the Flint `init` function cannot call other contract functions. Please refer to section 2.8.6 for more information on how Flint generates code. This means we do not have to deal with the issue of functions being called in the `init` body as Flint does not support this.

We delete all the statements in the `init` body to prevent any unwanted side effects from double initialisation. As we insert the function `testFrameworkConstructor` after the semantic checks are run by the Flint compiler, we can modify immutable state variables in a function other than `init`. We argue that this transformation maintains the original intent of the programmer as we execute everything that was in the original `init` body with the exact same parameters.

```
1
2 contract Counter (s1, s2) {
3     var counter : Int
4 }
5
6 Counter @(s1, s2) :: (any) {
7     public func testFrameworkConstructor(x: Int) {
8         self.counter = x
9         become s1
10    }
11 }
12
13 Counter @(s1) :: (any) {
14     public init(x : Int) {
15         // self.counter = x (this line is deleted)
16         // become s1 (this line is deleted)
17     }
18 }
```

Listing 6.4: Processed Contract

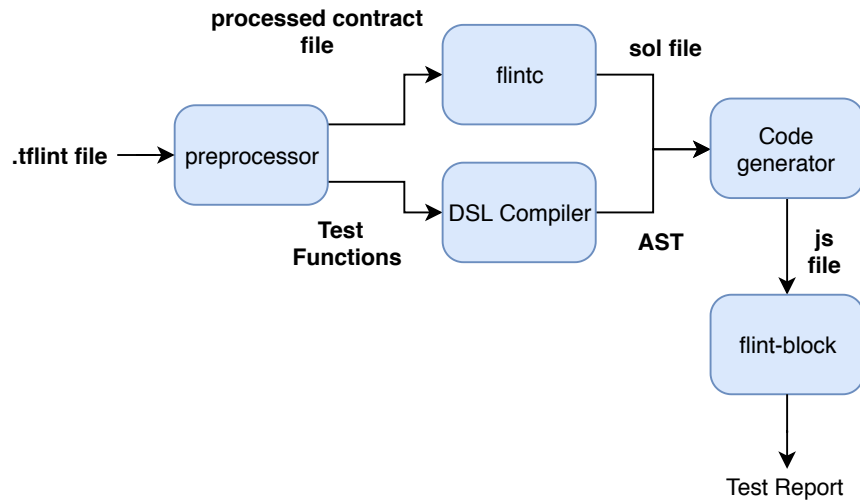


Figure 6.2: System Architecture for Flint test

6.5 System Architecture

Figure 6.2 is the system diagram for the testing framework. The framework consists of multiple stages. The pre-processor loads in the `.tflint` file which contains the tests and the location of the contract being tested. It then loads the contract being tested and processes it to insert the `testFrameworkConstructor`. If the test framework is run with coverage then the pre-processor adds instrumentation code to the contract. We will cover the coverage provider in section 6.7. The pre-processor also extracts the test functions from the contract and passes this to the DSL compiler. The compiler produces an AST which represents a JavaScript file and runs a set of semantic checks over the test functions. The semantic checks that are run are as follows:

- We check that the program types well with regards to assignment expressions.
- Scope checks, this includes checking variables are in scope, contract members (functions and events) exist and general functions are in scope.
- Modifier checks, ensuring that variables marked as `let` are not reassigned. We also check for invalid re-declarations.
- Checking that functions marked as payable have Wei attached when being called.

The Flint compiler is used to compile the processed contract described in the previous paragraph, the generated Solidity file and the AST produced by the DSL compiler is passed to the code generator. The code generation produces a JavaScript file, this file is then run against a local blockchain node created by `flint-block`. Once the file has finished executing, it reports the results of running the tests.

We use the `web3.js` library to build a set of functions which enable us to deploy contracts to a local blockchain, interact with them and query the blockchain for information about contracts e.g. event logs. The generated JavaScript file is composed of these functions. In subsequent sections, we will discuss how we used the `web3` library to deploy and communicate with contracts on the blockchain. As `web3.js` by design is asynchronous, we make heavy use of JavaScript promises to convert asynchronous calls into synchronous ones.

6.5.1 Deploying a Contract

The test runner compiles the Flint contract, extracting its the ABI and bytecode. We then use the `web3.js` library to instantiate a contract using its ABI and then send a transaction to the blockchain with the contract bytecode attached to deploy it. Please refer to section 2.8.3 for more details on how contracts are deployed to the blockchain.

6.5.2 Calling Functions

There are two ways of communicating with contracts on the blockchain, the first is using the 'call' method and the second is by sending a transaction [84].

Transactions require gas to run and change the state of the network. They are not processed immediately and require the consensus of other nodes in the network before they can be committed to the blockchain. Additionally, since they are not processed immediately, when calling a function as a transaction the blockchain does not return the result of the function. Instead it returns a transaction object which contains the hash of the transaction that executed this function. To extract return values the programmer has to make use of events [84].

Calls on the other hand do not change the state of the network. They are free to run and are processed immediately. To run calls, instead of running the function as a transaction, the local blockchain node simulates running the function but does not alter the state of the network. Since calls are simulated and processed immediately functions called as calls can return values [84].

Flint functions that contain events or are marked as mutating or payable need to be run as a transaction since they update the state of the blockchain. All other methods can be run as a call i.e. functions that are read-only. Calls are faster than transactions because you do not need to pay for the overhead of altering the state of the blockchain. Thus, it is desirable to run a function as a call when possible [84].

As it was our intention to make writing tests as easy as possible and as close to Flint syntax as possible. The test framework hides this complexity away and infers whether a function should be run as a call or a transaction. We created functions using `web3.js` to run contract functions either as calls or transactions. To handle

the asynchronous nature of transactions, we monitor the state of the blockchain and wait until transactions of interest are mined. Additionally, the test runner take care of the conversion between blockchain types and Flint types. This all culminates in an experience where the developer can write tests with Flint syntax and semantics and not be exposed to the details of the underlying blockchain.

6.5.3 Checking For Reverted Transactions

An important part of the testing framework is to check for reverted transactions. A transaction is reverted whenever a function within it calls `revert`. In the event of a revert, the blockchain undoes any effect of that transaction. However, even when a transaction fails the blockchain generates a receipt for that transaction. The receipt has a status field which is set to 0 if the transaction was reverted. We use this field to check for reverted transactions. We will discuss in subsequent sections how this was used to implement some of the features in the testing framework.

6.6 Test Framework Features

For the testing framework to be useful it needs to support deploying a contract and calling functions on it from within a test. Additionally, the testing framework should enable programmers to test all the language features of Flint: events, function protections (state and caller) and exceptions. It should also allow developers to send Ether from a test to a contract.

In this section, we will discuss the features that were implemented in the testing framework.

6.6.1 Testing Caller Protections

Caller protections are an integral safety feature in Flint. It allows the developer to explicitly restrict access to functions to specific callers. Listing 6.5 shows an example of caller protections being used in a Flint contract. The `reset` function can only be called by the owner, `increment` can be called by anyone. The testing framework supports writing tests to check caller protections. Developers can assert in a test that a function can or cannot be called depending on the address it is called with. To support this the testing framework has a mechanism to change the address transactions are sent from during a test. Listing 6.6 is an example of a set of tests which check the caller protections for the contract in Listing 6.5.

The function `test_any_can_call_increment` verifies that the `increment` function on the counter can be called by anyone. The test creates two new users, deploys a counter contract with one of the addresses as the argument to the constructor. This sets the owner of the contract as that address. The test then changes the address context of the testing framework using the `setAddr` function and then checks whether the `increment` method can be called using the `assertCallerSat` function.

```

1  contract Counter {
2      let owner : Address
3      var value : Int = 0
4  }
5
6  Counter :: (any) {
7      public init(initial_owner: Address) {
8          self.owner = initial_owner
9      }
10
11     mutating public func increment() {
12         self.value += 1
13     }
14 }
15
16 Counter :: (owner) {
17     mutating public reset() {
18         self.value = 0
19     }
20 }

```

Listing 6.5: Using Caller Capabilities in Flint

```

1  contract TestCounter {
2      let filePath : String = "<location of contract>"
3      let contractName: String = "Counter"
4      let TestSuiteName : String = "CounterTests"
5  }
6
7  TestCounter :: (any) {
8
9      public func test_any_can_call_increment() {
10         let owner : Address = newAddress()
11         let not_owner : Address = newAddress()
12         let c : Counter = Counter(owner)
13
14         setAddr(not_owner)
15         assertCallerSat(c.increment)
16         unsetAddr()
17     }
18
19     public func test_only_owner_can_call_reset() {
20         let owner : Address = newAddress()
21         let not_owner : Address = newAddress()
22         let c : Counter = Counter(owner)
23
24         setAddr(not_owner)
25         assertCallerUnsat(c.reset)
26         unsetAddr()
27     }
28 }
29
30
31 }

```

Listing 6.6: Flint test checking caller protections

6.6.1.1 Implementation

Flint implements caller protections by dynamically checking whether the calling address of a function matches any of the callers specified in the protection block. If the caller does not match any of the specified callers then the transaction is reverted. Thus, to check if a particular calling address can call a method, we call the method as a transaction and check if the transaction was successful.

6.6.2 Testing State Protections

Flint supports type states which enables developers to partition a contract into a set of states and restrict access to functions depending on the state of the contract. The testing framework allows developers to write tests to assert which functions are accessible in a particular state. Partitioning contracts into states is a common smart contract design pattern known as the state transition pattern, and applicable to many contract domains [14, 69]. Some examples of the state transition pattern being used in the real world can be found in the betting contract from Ethhorse which enables users to bet on cryptocurrencies and also the auction contract owned by PocketInns [27, 58].

This pattern also can be used to implement other smart contract design patterns e.g. the Emergency Stop design pattern. This pattern is used to provide users with the option to disable a contracts critical functionality in the event of an emergency. An example of this pattern implemented in Solidity can be found in the Pausable.sol contract in the OpenZeppelin library [54].

Programming with type-states is an important part of developing contracts using Flint. It offers programmers a way to increase the security of their contract by restricting access to functions depending on the type-state of the contract. Thus, we think it is important for the unit testing framework to support testing of state protections. This ensures that the programmer can be confident that their protections have been tested and are working correctly.

Listing 6.7 is an example of a contract which uses type-states. The contract has two states: `inc` and `get`. In the `inc` state the contract enables a user to increment the value of a counter and in the `get` state you can get the value of the counter. We would like to write tests to verify that the contract methods can only be called in the type-states that the programmer intended. Listing 6.8 is an example of a test file one could write to test the state protections offered by this contract. The first test deploys a counter contract, changes the state of the contract using the method `changeINC` and then asserts that the `increment` method can be called. The second test does a similar thing but asserts that a method cannot be called. These two tests allow a programmer to write tests to verify the state protections in this contract.

```
1 contract Counter (get, inc) {  
2     var value: Int = 0  
3 }
```

```
4
5 Counter @(get) :: (any) {
6   public init() {
7     become get
8   }
9
10  mutating public func changeINC() {
11    become inc
12  }
13
14  public func getValue() -> Int {
15    return value
16  }
17 }
18
19 Counter @(inc) :: (any) {
20
21  mutating public func increment() {
22    value += 1
23  }
24 }
```

Listing 6.7: Flint Contract Using TypeStates

```
1 contract TestCounter {
2   let filePath : String = "<location>"
3   let contractName: String = "Counter"
4   let TestSuiteName : String = "CounterTests"
5 }
6
7 TestCounter :: (any) {
8
9   public func test_state_success() {
10    let c : Counter = Counter()
11
12    c.changeINC()
13
14    assertCanCallInThisState(c.increment)
15  }
16
17  public func test_state_negative() {
18    let c : Counter = Counter()
19
20    c.changeINC()
21
22    assertCantCallInThisState(c.getValue)
23  }
24
25 }
```

Listing 6.8: Flint test file to verify state protections

6.6.2.1 Implementation

Similar to caller capabilities, Flint uses dynamic checks to verify state protections and reverts if the check fails. Thus, we have implemented the asserts for state protections in a similar manner to the ones implemented for caller protections.

6.6.3 Testing For Exceptions

Exceptions are an important part of programming, they allow programmers to react in a predictable manner when an error occurs. The blockchain supports exceptions by allowing programmers to revert a transaction, this undoes any effect of the transaction that contains the offending function. Flint exposes a function in its standard library called `fatalError` which enables a programmer to revert a transaction in the case of an error [31]. Testing exceptions is good programming practice as it ensures that the exception handling code is working correctly [28, 76].

The testing framework supports testing of exceptions by allowing programmers to assert a function call should throw an exception. Listing 6.9 shows a contract which throws an exception. Listing 6.10 contains a set of tests which check the exception handling code in this contract. The first test verifies an exception is thrown when the function is called with the argument 10. The second test verifies that the code works fine when an argument other than ten is passed in.

```
1 contract Counter {
2     var val : Int = 0
3 }
4
5 Counter :: (any) {
6
7     public init() {}
8
9     mutating public func willThrowOn10(arg : Int) {
10         if (arg == 10) {
11             fatalError()
12         }
13
14         self.val = 0
15     }
16 }
17 }
```

Listing 6.9: Flint contract That throws Exceptions

```
1 contract TestCounter {
2     let filePath : String = "<location>"
3     let contractName: String = "Counter"
4     let TestSuiteName : String = "CounterTests"
5 }
6
7 TestCounter :: (any) {
8
9     public func testExceptionThrown() {
```

```

10     let c : Counter = Counter()
11
12     assertWillThrow(c.willThrowOn10, 10)
13 }
14
15 public func testNoExceptionThrown() {
16     let c : Counter = Counter()
17
18     let val : Int = c.willThrowOn10(12)
19
20     assertEquals(12, val)
21 }
22 }

```

Listing 6.10: Flint test file to test for exceptions

6.6.3.1 Implementation

To implement testing for exceptions, we run a transaction and check if it has failed.

6.6.4 Testing Events

As mentioned before, events are the bread and butter of programming decentralised applications. JavaScript callbacks in the front end of a decentralised application can listen for these events and react appropriately e.g. update UI [18].

Decentralised applications offer programmers a way to create products that are backed by a blockchain. Thus, it is important that programmers have a way to test that the blockchain is emitting the correct events to ensure that their decentralised application works correctly.

Listing 6.11 is a contract which emits an event everytime value is incremented. This event can then be used to power a front end which displays the value of the counter. Listing 6.12 is an example of a test one could use to check that the correct event is fired with the right values when the counter is incremented. The first argument to the assert is the name of the event we are testing. The second is an argument filter which specifies that the user is only interested in events that have fired with the value 1. The filter is optional but if a filter is specified then it must be specified for all the arguments the event has. Additionally, the values for the filter must match the order of the arguments in the event declaration.

```

1 contract Counter {
2     var value : Int = 0
3     event counterVal(val: Int)
4 }
5
6
7 Counter :: (any) {
8
9     public init() {}

```

```

10
11     mutating public func increment() {
12         self.value += 1
13         emit counterVal(val: self.value)
14     }
15 }

```

Listing 6.11: Flint contract using events

```

1  contract TestCounter {
2      let filePath : String = "<location>"
3      let contractName: String = "Counter"
4      let TestSuiteName : String = "CounterTests"
5  }
6
7  TestCounter :: (any) {
8
9      public func test_event_fired() {
10         let c : Counter = Counter()
11         c.increment()
12         assertEventFired(c.counterVal, 1);
13     }
14 }

```

Listing 6.12: Flint test file checking events have fired

6.6.4.1 Implementation

To check if an event has fired we use a `web3` function that allows us to query the event logs of a contract and also apply a filter. The filter can be used to filter for events that have fired with specific values. The function returns all the logs that match this filter and then we check if any of the logs originated from transactions prior to the assert in the test. The event filter is created by the DSL compiler as part of the code generation stage. If no argument filter is passed in then we check if the event was fired with any value.

6.6.5 Testing Ether Transactions

A key part of programming on the blockchain is being able to send money to contracts. Thus, the testing framework enables you to attach money to a function call when calling a function from a test. This will enable programmers to test how their contracts react when it receives Wei.

Listing 6.13 is an example of a Flint contract with a payable function. Listing 6.14 is an example of a test which a programmer can write to verify that the contract in listing 6.13 is reacting correctly when money is sent to it. In the testing framework, to send money to a contract, the developer calls a payable function and attaches Wei to the function call using the argument label `_wei`.

```

1  contract Bribe {
2      var value : Int = 0

```

```

3  event Bribed(bribe: Bool)
4  }
5
6  Bribe :: (any) {
7      public init() {}
8
9      @payable
10     public func bribe(implicit w: Wei) {
11         let rawVal : Int = w.getRawValue()
12         var bribed: Bool = false
13         if (rawVal > 100) {
14             bribed = true
15         }
16
17         emit Bribed(bribe: bribed)
18     }
19 }

```

Listing 6.13: Flint contract with payable function

```

1  contract TestBribe {
2      let filePath : String = "<location>"
3      let contractName: String = "Bribe"
4      let TestSuiteName : String = "BribeTests"
5  }
6
7  TestBribe :: (any) {
8
9      public func test_not_bribed() {
10         let b : Bribe = Bribe()
11         b.bribe(_wei: 80)
12         assertEventFired(b.Bribed, false)
13     }
14
15     public func test_was_bribed() {
16         let b : Bribe = Bribe()
17         b.bribe(_wei: 100)
18         assertEventFired(b.Bribed, true)
19     }
20 }
21 }

```

Listing 6.14: Flint test which sends money to a contract

6.6.5.1 Implementation

The DSL compiler rejects any call to a function marked as payable which has no Wei attached. To support transactions that send money, we need to ensure that the address the transaction is being sent from has Ether available. Thus, the framework uses an address that is pre-allocated with Ether by `flint-Block`. The compiler generates code which runs the function call as a transaction and attaches money to the transaction. Any function marked as payable will always be run as a transaction as it alters the state of the blockchain.

6.7 Code coverage

In this section, we will cover the code coverage feature of the testing framework. We will discuss the motivation behind it and how it was implemented. Please refer to section 2.6 for more details on the different types of coverage and how code coverage can be implemented.

6.7.1 Implementation

To implement code coverage, we decided to instrument the Flint source code instead of the EVM assembly or the YUL assembly. This is because we do not want to lose contextual information such as event names and also because the goal of this project is not to make a universal code coverage tool but rather one that is specific for Flint. We decided it was not worth the investment in time to develop a general code coverage tool as working at the assembly level is generally more arduous and error prone.

6.7.1.1 High Level Overview

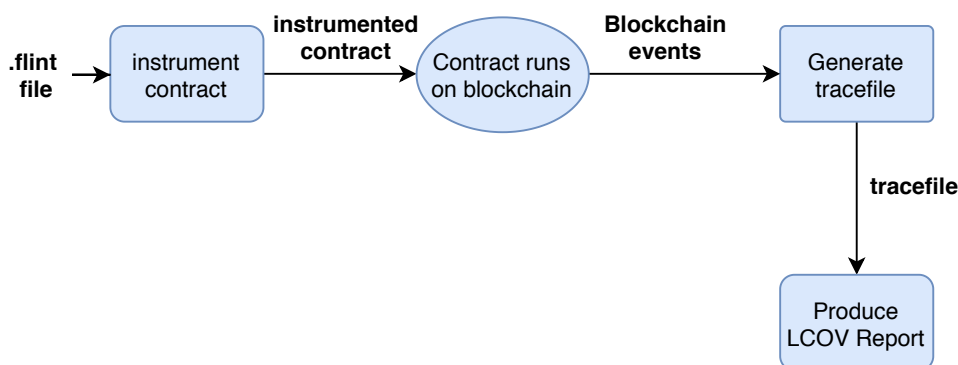


Figure 6.3: System Diagram For Code Coverage Provider

The code coverage provider is an additional module that can be plugged into the pre-processor of the testing framework. Figure 6.3 is the system diagram for the code coverage provider. The code coverage module instruments the Flint contract that is being tested, the contract is then run on the blockchain, once execution is complete the provider collects the instrumentation data and generates a report displaying the code coverage results. In subsequent sections, we will discuss each of these stages in detail.

6.7.1.2 Instrumentation

Since the only way smart contracts can send information to an external environment is via events, we use events to instrument the code. We have chosen to implement line coverage, branch coverage and function coverage. The combination of these

coverage metrics we believe will give good visibility of the codebase to the developer. Line coverage is a useful metric and is easy to understand, the drawbacks of it can be mitigated by providing a branch coverage metric. Flint constructs which can produce false results with line coverage are if blocks, do catch blocks and for loops but for all of these branch coverage will accurately report results irrespective of the developers coding style. Please refer to section 2.6 for more details on the different types of code coverage. Branch coverage is a useful metric as it gives visibility to the programmer on all the decision points in a program and which decisions have been executed. We also support function coverage which comes with an associated function counts table. The table displays the number of times each function was called. This is useful as it gives the programmer visibility on which functions are being called a lot and this can potentially help them identify bottlenecks.

The instrumentation is done on the Flint source code, we process the contract and add three events to the contract declaration. The events we add are shown in Listing 6.15. The `stmC` event is used to measure statement coverage, we add this event before each executable statement. It is important that we add the `emit` statement before each executable statement to account for `return` and `become` statements as they are both terminating statements. The `branchC` event is used to measure branch coverage, it records the line number the branch decision is on, the branch that was taken and the block number of that branch. The block number is an internal count which is used later on to generate the code coverage report. We add this event to the beginning of each branch in the program. The `funcC` is used to measure function coverage, we record the name of the function that is being executed and the line number of the declaration of that function. We add this event to the beginning of each function in the program.

Listing 6.16 is a contract before it has been instrumented and Listing 6.17 is the contract after it has been instrumented. This instrumentation is performed by the code coverage provider during the pre-processing of the contract. When we run the code coverage provider, every function call is run as a transaction as events alter the state of the blockchain thus cannot be emitted during a call.

```
1 event stmC(line: Int)
2 event branchC(line: Int, branch: Int, blockNum : Int)
3 event funcC(line: Int, fName: String)
```

Listing 6.15: Instrumentation Events

```
1 contract Counter {
2   var value : Int = 0
3 }
4
5 Counter :: (any) {
6   public init() {}
7
8   public func testCoverage() {
9     if (true) {
10       let z1 : Int = 0
11     }
12 }
```

```

12
13     let z : Int = 0
14 }
15
16 }

```

Listing 6.16: Uninstrumented Contract

```

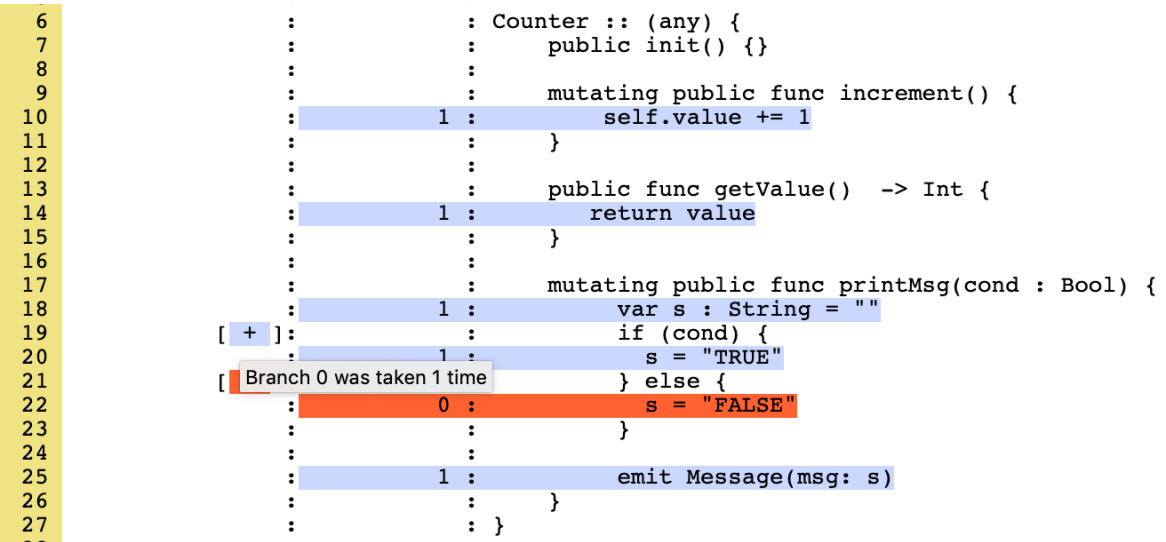
1  contract Counter1 {
2      var value : Int = 0
3      event stmC(line: Int)
4      event branchC(line: Int, branch: Int, blockNum : Int)
5      event funcC(line: Int, fName: String)
6  }
7
8  Counter1 :: (any) {
9      public init() {}
10
11     public func testCoverage() {
12         emit funcC(line: 8, fName: "testCoverage")
13         if (true) {
14             emit branchC(line: 9, branch: 0, blockNum: 2)
15             emit stmC(line: 10)
16             let z1 : Int = 0
17         } else {
18             emit branchC(line: 11, branch: 1, blockNum: 3)
19         }
20
21         emit stmC(line: 13)
22         let z : Int = 0
23     }
24
25 }

```

Listing 6.17: Instrumented Contract

6.7.1.3 LCOV Code Coverage Report

Once the program has finished executing, we gather the instrumentation events that were emitted by the blockchain and combine it with information generated by the coverage provider to produce an LCOV tracefile [48]. The tracefile is used to generate a HTML report. Figure 6.4 displays the main sections of the report. Figure 6.4a visually represents what code has been executed and what code has not. The lines highlighted blue are lines that have been executed and the number directly to the left is the number of times that line has been executed. The lines highlighted red are ones that have never been executed. The column to the far right is for the branch coverage data, it highlights what branches have been taken and how many times they were taken. Figure 6.4b is a summary of the coverage data. Figure 6.4c is the function counts table associated with this report. This table displays how many times each function was called. The Flint contract file and test file for this coverage report can be found in the appendix (Appendix D).



(a) Visual Report Of Code Coverage

	Hit	Total	Coverage
Lines:	5	6	83.3 %
Functions:	3	3	100.0 %
Branches:	1	2	50.0 %

(b) Summary Of Code Coverage

Function Name ↕	Hit count
getValue	1
increment	1
printMsg	1

(c) Function Hits table

Figure 6.4: Code Coverage Report

Field	Arguments	Coverage
DA	[line number, execution count]	Statement
BRDA	[line number, block number, branch taken, execution count]	Branch
FNDA	[execution count, function name]	Function

Table 6.1: Fields In An LCOV Record

Field	Purpose
FNF	Number of functions found
FNH	Number of functions executed
BRF	Number of branches found
BRH	Number of branches executed
LF	Number of executable lines found
LH	Number of lines executed

Table 6.2: Summary Fields In An LCOV Record

6.7.1.4 LCOV Tracefile

We will now discuss the format of an LCOV tracefile. Listing 6.18 is an example of an LCOV tracefile. A tracefile consists of multiple records, each record contains the coverage data for a single source file. You begin a record with `SF:<file location>` and end it using `end_of_record`. Table 6.1 lists the main fields that can appear in a tracefile record. The record also contains summary data which documents various totals about the coverage data. Table 6.2 lists all the entries in a tracefile record that are used to store summary data. The tracefile is processed with `genhtml` to produce a HTML coverage report [48].

```

1 SF:[redacted]
2 DA:5,1
3 DA:6,1
4 BRDA:11,4,0,1
5 FNDA:1, getValue
6 FN:13, getValue
7 BRF:1
8 BRH:1
9 FNF:1
10 FNH:1
11 LF:2
12 LH:2
13 end_of_record

```

Listing 6.18: LCOV Tracefile

6.8 Conclusion

The testing framework helps Flint developers write correct code. The framework enables developers to deploy and interact with contracts within a test. It also supports testing of all of Flint's language features: function protections, exceptions and events. Additionally, it supports sending money to a contract from within a test enabling programmers to test payable functions. The tests can be written using Flint-like syntax for the convenience of the programmer. We also provide code coverage as part of the testing framework. The framework can generate a coverage report to help programmers identify untested parts of their codebase.

Chapter 7

Interactive Console (REPL)

In this section, we will discuss the REPL and its implementation.

7.1 Introduction

Developers regularly run code they are writing on their local machine to check the code is working as expected. We expect that Flint developers will do the same. However, the process of deploying and interacting with smart contracts manually is difficult. Programmers are required to connect to a local blockchain node, compile their contract, encode the contract data into an appropriate format for the blockchain and submit a transaction to the blockchain to deploy the contract. Furthermore, to interact with the contract, programmers need to understand how to encode function calls in accordance to the ABI specification. Usually the language of choice to perform these operations is JavaScript which means the Flint developer must also be familiar with JavaScript [34]. Programmers can alleviate some of this burden by writing scripts using web3 to simplify the process. However, this is not an interactive process, places a burden on the programmer and is not beginner friendly. We have created the Flint REPL to address this issue and provide developers with a convenient and easy way to deploy and interact with contracts.

7.2 REPL Features

In this section, we will walk through how one can use the REPL. To demonstrate the important features of the REPL we will perform the following actions:

- Compile and deploy a contract.
- Call functions on a deployed contract.
- Query the event logs of a deployed contract.

The REPL also supports evaluation of expressions and can support expressions which contain contract function calls as sub-expressions. Please refer to section 7.3.1.1 for more information on expressions the REPL supports.

```
demo>flint-repl repl_eval/Counter.flint

Counter:

Contract Functions:
getValue() (Constant)
bribe(_wei: Wei, val : Int) (Mutating, Payable)
increment() (Mutating)

flint>
```

Figure 7.1: Compiling Contracts With The REPL

7.2.1 Deploying a Contract

An important part of the REPL is to enable programmers to quickly compile and deploy their contracts to a local blockchain. Figure 7.1 illustrates how a user launches the REPL. Contracts are compiled when the REPL is launched. The programmer launches the REPL using the `flint-repl` tool, specifying the file containing the contracts they are interested in as an argument. The REPL compiles all of the contracts within the file and makes them available to the programmer. Additionally, for each contract the REPL displays in a user readable format the signature of all the contract functions that can be called.

Within the REPL, the programmer can create instances of a contract and interact with them. The contracts are deployed to a local blockchain. Figure 7.2 illustrates how a programmer instantiates a contract within the REPL. The REPL returns the address of the deployed contract.

7.2.2 Calling Functions

We support calling functions on a contract using Flint syntax. You call functions on a particular instance of a contract. The syntax is as follows: `<instance name>.<function name>(<params>)`. The REPL dispatches the call to the blockchain, choosing between sending a transaction or a call depending on the function. Please refer to section 6.5.2 for more information on the differences between the two. The REPL also con-

```
flint>let c : Counter = Counter()
Deploying c : Counter
Contract deployed at address: 0x5d27a8d7efba6577b316867076826978bac083df
flint>
```

Figure 7.2: Deploying Contracts In The REPL

```
flint>let x3 : Int = c.getValue()  
0  
flint>x3  
0  
flint>
```

Figure 7.3: Calling Functions On a Contract In The REPL

```
flint>c.bribe(_wei: 100, val: 100)
```

Figure 7.4: Attaching Ether To a Function Call In The REPL

verts between Flint types and blockchain types. Figure 7.3 is an example of calling a function on a contract in the REPL.

The REPL also supports calling payable functions. To call a payable function, the user calls the function as normal but alongside the normal function parameters they specify how much Wei they wish to send using the `_wei` argument label. Figure 7.4 illustrates how to call a payable function.

7.2.3 Querying Event Logs

The REPL also supports querying the event logs of a contract instance. Figure 7.5 illustrates how a user can query the event logs of a contract instance. The REPL returns the number of times the event was fired and also returns an array of logs which contains each invocation of the event. The logs are processed so that the displayed result looks like Flint. The REPL labels each of the event arguments and converts blockchain types into Flint types.

```
flint>c.increment()  
0xad25de61e5ebe21a3fdb73b6da4b57411cb93e79132b90daca67e5be046e81cd  
flint>c.CounterVal  
{  
(val: 1),  
(val: 2),  
}  
Fired: 2  
flint>
```

Figure 7.5: Querying event logs on a contract instance

7.3 Implementation

In this section, we will discuss the implementation of the REPL.

7.3.1 Overview

Figure 7.6 is the UML diagram for the Flint REPL. We have only kept the information that is necessary to explain how the system operates at a high level. The information we have removed includes classes, function parameters, return types and field variables. We have chosen to draw the UML diagram using the specification from IBM's knowledge centre [21].

The entry point of the REPL is in the REPL class, when the the REPL is launched the contracts found in the file specified by the user are loaded into the REPL. For each contract, the REPL compiles it and extracts its Application Binary Interface (ABI). The ABI is processed and displayed to the user in a readable format. The REPL also processes each of the contracts to extract information about the functions and events that exist within them. It also adds a function to each contract which it uses to support deploying of contracts with constructor arguments. We will discuss this further in section 7.3.2. As part of loading the contracts, the REPL constructs a set of REPLContract objects. These objects represent each of the contracts the REPL knows about and are responsible for deploying and managing interactions with any instance of that contract. The REPLContract object maintains a list of instances that currently exist in the REPL for the contract that it represents.

After the contracts have been loaded, the run function is called and this starts the REPL's main read-eval-print loop. The REPL uses the front-end of the Flint compiler to parse the users input and then calls the `process_expr` function on the REPLCodeProcessor to evaluate the expression. The code processor applies a set of semantic checks to ensure the statement is semantically valid. The semantic check we apply are as follows:

- We check that the program types well with regards to assignment expressions.
- Scope issues, this includes checking variables are in scope, contract members (functions and events) exist.
- Modifier checks, ensuring that variables marked as `let` are not reassigned to.
- Checking that functions marked as payable have Wei attached when being called.

The expression is then evaluated. Depending on the type of expression we either evaluate it using the blockchain or the REPLCodeProcessor. The backing blockchain for the REPL is the one provided by `flint-block`.

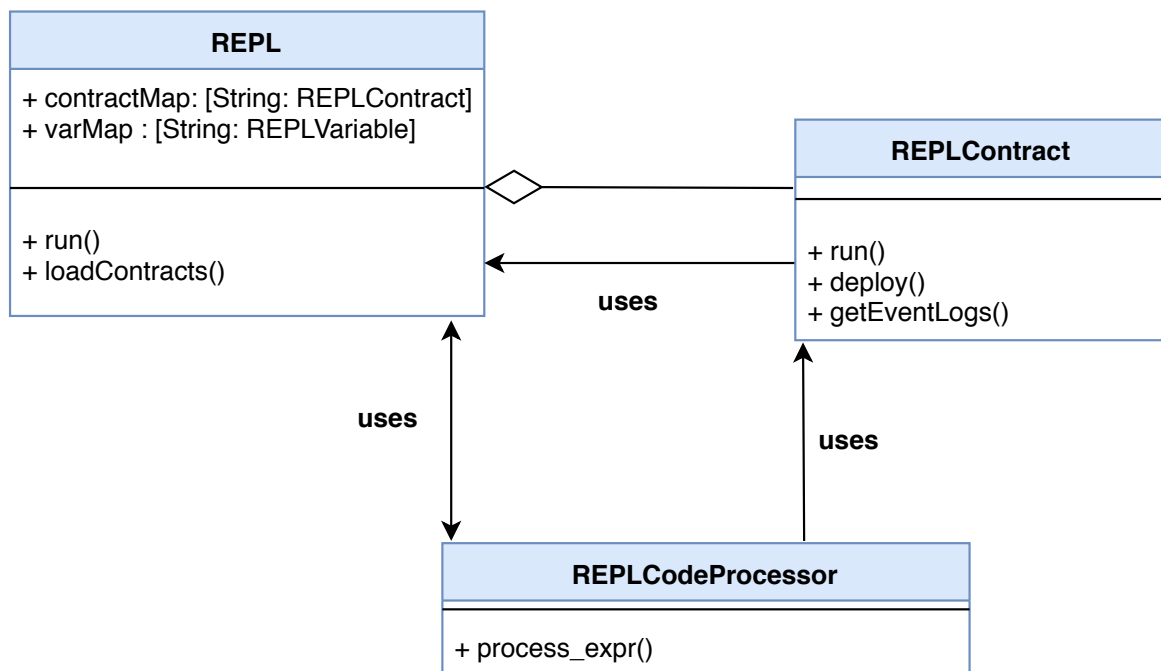


Figure 7.6: UML Diagram for REPL

7.3.1.1 Expressions the REPL supports

The BNF for the REPL can be found in the appendix (Appendix C.2). The REPL supports a subset of Flint syntax. We have decided to only support single line statements in the REPL as we do not think a REPL like environment is suited to multi line programming.

The REPL distinguishes between expressions that involve a contract and ones that do not. Expressions which involve a contract are as follows:

- Expression which deploys a contract
- Expression which calls functions on a contract
- Expression which query the event logs of a contract.

These expressions are run against the local blockchain node. We have created a set of scripts that utilise the web3 library to perform these operations on the blockchain. All other expressions are evaluated by the `REPLCodeProcessor`. In subsequent sections, we will describe in more detail how we implemented some of the key features of the REPL.

7.3.2 Deploying a Contract

To support deployment of contracts, we wrote a set of scripts which utilise the web3 and solc libraries to compile the contract, extract the ABI and then deploy the contract to the blockchain. As deployment statements need to be treated differently to

other statements, the code processor checks every assignment expression to verify whether it is a deployment expression or not. If is then it queries the REPL to get the appropriate REPLContract object and asks the REPLContract object to deploy the contract.

Contracts can be deployed with constructor arguments. To support this the REPL adds the method `replConstructor` to each contract. The REPL then calls this method after the contract has been deployed to initialise the contract with constructor arguments. We do a similar thing in the testing framework to support deployment of contracts with constructor arguments. Please refer to section 6.4.1 for more information on why we do this and how it works. Constructor arguments can be any expression supported by the REPL except for assignment expressions and contract events.

7.3.3 Contract Interaction

In this section, we will discuss how we implemented contract interaction in the REPL. We support two forms of interaction, that is calling functions on a contract and querying the event logs of a contract.

7.3.3.1 Calling Functions

Similar to the testing framework, we distinguish between calls and transactions in the REPL. The REPL infers which one to use based on the function signature. Please refer to section 6.5.2 for more detail on the difference between the two.

The REPLCodeProcessor dispatches function calls to the appropriate REPLContract object. The REPLContract selects the correct contract instance and then dispatches a call to the blockchain. It decides if the function should be called as a transaction or a call and also converts between Flint types and Blockchain types. Transactions involving currency have the appropriate amount of Ether attached.

7.3.3.2 Querying Event Logs

We support querying the event logs of a contract. This enables programmers to check how many times and event has fired and with what values. The REPLCodeProcessor delegates event queries to the appropriate REPLContract object. The REPLContract object selects the correct contract instance and then interfaces with the blockchain to collect the event data for that contract instance. The event data is then processed and displayed to the user.

7.4 Conclusion

In conclusion, we believe we have created a convenient tool that enables a developer to easily deploy and interact with their contracts. It supports calling functions on a

contract and querying its event logs. This tool can be used by developers to interact with their contracts as they are developing them. This concludes our description of the ecosystem and in the next section we will discuss the evaluation of the ecosystem.

Chapter 8

Evaluation

We will now evaluate the different tools in the ecosystem. The different tools are as follows:

1. Testing Framework
2. REPL (Interactive Console)
3. Contract Analysis Tools
4. Language Server

The evaluation will consist of both qualitative and quantitative components. The majority of the evaluation will be qualitative as the the purpose of the evaluation is to demonstrate the ecosystem adds value by helping developers create safer and better Flint contracts. Performance is only necessary to ensure that the ecosystem is usable.

8.1 Testing Framework

The goal of the testing framework is to help Flint developers write correct contracts by enabling them to write tests which exercise their contracts. It provides coverage support to help developers locate untested regions of their codebase. The evaluation will be primarily qualitative and aims to show that the testing framework makes development safer. We will do this by demonstrating how the test framework enables a programmer to detect bugs early on by writing tests for vulnerable contracts, how it helps a developer implement common smart contract design patterns safely and program reliably with events. We source vulnerable contracts from the SWC Registry (Smart Contract Weakness Classification and Test Cases) [57]. This registry documents known vulnerabilities in smart contracts and provides real world examples of the vulnerability.

We will also compare our testing framework with the Truffle testing framework. The main points of comparison will be on the expressiveness and the usability of

Feature	Truffle Test	Flint-Test
Verifying Events	10.52	18.99
Testing State Protections	10.64	22
Testing Caller Protections	10.53	26.14
Testing For Exceptions	8.11	23.93

Table 8.1: Execution Speed Of Testing Framework For Different Features

the two frameworks. Truffle currently supports writing tests in both Solidity and Javascript. For the most part we will compare with test written in JavaScript as they are more expressive and are not subject to the limitations of running on the blockchain. Please refer to section 2.5.3 for more details on the difference between Solidity and JavaScript tests. For tests that can be written in both languages, the Solidity tests are similar to their JavaScript equivalent for the most part. They differ on how they test for exceptions, we will discuss how Solidity tests exceptions in section 8.1.6.

8.1.1 Speed Of Testing Framework

As part of the evaluation, we compared the execution speed of Flint-Test to the Truffle testing framework. For each feature in Flint-Test, we wrote the test in Flint-Test and Truffle (JavaScript). Table 8.1 shows how long each of the testing frameworks took to run tests which exercised specific features. The tests we ran for this analysis are the same ones we use in our qualitative evaluation, they can be found in the remainder of this chapter. Table 8.2 details how long each of the testing frameworks took to run test suites of varying sizes. The tests we used to generate these results can be found in the appendix (Appendix E).

We have found that in all cases Truffle is significantly faster. This can be attributed to the fact that the backing blockchain we use for Flint-Test implements the complete Ethereum protocol. The Ethereum protocol seeks to control the block creation time for security reasons. It converges to a block creation time of one block every 12s. This limits the number of transactions that can be processed per second and limits the speed of Flint-Test. Truffle mitigates this issue by implementing a custom blockchain which does not have this limitation [89]. We did not implement our own custom blockchain as this is a sizeable piece of software and was not within the scope of this project. The custom blockchain used by Truffle is around 12,000 lines of code [90]. Future iterations of the Flint ecosystem should implement a custom blockchain that can be used for development and testing. Please refer to section 3.3 for more details on the limitations of using a complete Ethereum node, the reasons behind this limitation and why we made the choice of using one.

Test Suite Size	Truffle Test	Flint-test
3	7.87	26.98
5	7.66	42.84
7	7.66	83.37

Table 8.2: Execution Speed Of Testing Framework For Test Suites Of Varying Sizes

8.1.2 Testing Access Control : Caller Capabilities

8.1.2.1 How is it useful?

Flint has caller capabilities which enable developers to restrict access to functions in a smart contract to specific callers. Listing 8.1 is an example of a Flint contract which uses caller capabilities. The function `onlyOwner` is wrapped in a contract behaviour declaration which permits only the user that has the address stored in the variable `owner` to call the function `onlyOwner`.

```

1  contract Counter {
2      let owner : Address = "0XXX"
3  }
4
5  /*
6      Removing initialisation code for the sake of brevity
7  */
8
9  Counter :: (owner) {
10     public func onlyOwner() {}
11 }
```

Listing 8.1: Contract utilising caller capabilities

To support testing of caller capabilities, the testing framework exposes the following functions: `assertCallerSat`, `assertCallerUnsat`, `setAddr` and `unsetAddr`. These functions allows you to change the address that transactions are sent from during a test and assert whether a method can be called or not.

To demonstrate why it is useful to test caller capabilities, we will examine a buggy solidity contract, convert it to Flint and show how we can use the testing framework to detect the bug by writing a test. The contract we are interested in is listed under SWC-105 which is categorised as CWE-284: Improper Access Control. The contracts listed under this vulnerability have incorrect access controls which allow unauthorised users to call sensitive functions. Figure 8.2 is an example of a contract with this vulnerability. The contract was designed to implement multiple ownership over a wallet, the intended goal of the contract was to ensure that only existing owners can add/delete owners. However, the function `newOwner` function is missing the `onlyOwner` modifier meaning unauthorised users can add owners [73].

```

1
2  contract MultiOwnable {
3      address public root;
4      mapping (address => address) public owners; // owner => parent of
        owner
```

```

5
6  /**
7   * @dev The Ownable constructor sets the original 'owner' of the
      contract to the sender
8   * account.
9   */
10  constructor() public {
11      root = msg.sender;
12      owners[root] = root;
13  }
14
15  /**
16   * @dev Throws if called by any account other than the owner.
17   */
18  modifier onlyOwner() {
19      require(owners[msg.sender] != 0);
20      _;
21  }
22
23  /**
24   * @dev Adding new owners
25   * Note that the "onlyOwner" modifier is missing here.
26   */
27  function newOwner(address _owner) external returns (bool) {
28      require(_owner != 0);
29      owners[_owner] = msg.sender;
30      return true;
31  }
32
33  /**
34   * @dev Deleting owners
35   */
36  function deleteOwner(address _owner) onlyOwner external returns
      (bool) {
37      require(owners[_owner] == msg.sender || (owners[_owner] != 0 &&
38      msg.sender == root));
39      owners[_owner] = 0;
40      return true;
41  }

```

Listing 8.2: SWC-105: multiowned_vulnerable.sol [73]

Listing 8.3 shows this contract translated into Flint with the vulnerability still present. Non owners can add and delete owners. However, we argue that if the developer engages in test driven development and writes tests to check caller protections then the vulnerability can be detected during the development process instead of during production.

```

1  contract MultiOwnable {
2      var root : Address
3      var owners: [Address : Address] = [:]
4      let zero : Address = 0x0000000000000000000000000000000000000000
5  }
6
7  MultiOwnable :: caller <- (any) {

```

```

8   public init(initial_owner: Address) {
9       self.root = initial_owner
10      self.owners[root] = root
11  }
12
13  mutating public func addOwner(new_owner: Address) {
14      self.owners[new_owner] = caller
15  }
16
17
18  mutating public func deleteOwner(owner: Address) {
19      self.owners[owner] = zero
20  }
21
22  }

```

Listing 8.3: Flint contract with SWC 105 vulnerability

Listing 8.4 demonstrates how one could write tests to check only existing owners are allowed to add new owners. The test initialises a new MultiOwned contract setting the address contained in the variable owner as the initial owner. It then changes the address context of the testing environment so transactions are sent from the address stored in not_owner. It then asserts that you should not be able to call the addOwner function.

```

1  contract TestMultiOwned {
2  // Test initialisation removed
3  }
4
5  TestMultiOwned :: (any) {
6      public func test_only_owners_can_add() {
7          let owner : Address = newAddress()
8          let not_owner : Address = newAddress()
9
10         let mo : MultiOwnable = MultiOwnable(owner)
11
12         setAddr(not_owner)
13         assertCallerUnsat("addOwner", not_owner)
14         unsetAddr()
15     }
16 }

```

Listing 8.4: Flint test file for MultiOwned

The result of running the test on the vulnerable contract is shown in Figure 8.1. The test fails which indicate to the developer that the access controls are not working correctly, thus the developer has been able to detect this bug during the development process. The correct implementation of this Flint contract can be found in the Appendix (Appendix E.1).

The use of caller capabilities to restrict access to functions is more generally known as the access control design pattern. The motivation behind this pattern is to restrict the usage of functions in a smart contract to a select audience. This is a very common design pattern, an empirical study conducted on over 800 Ethereum smart contracts

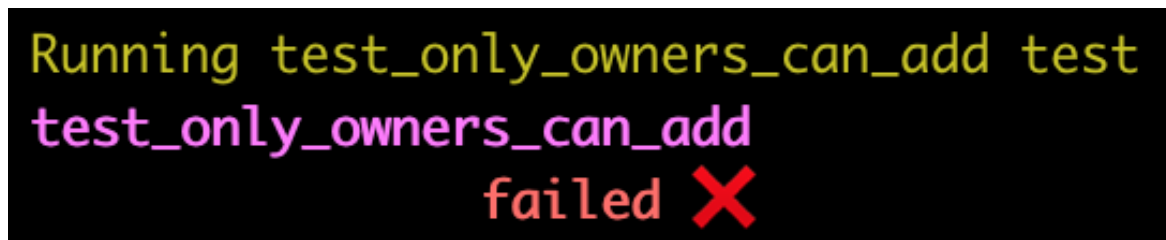


Figure 8.1: Output of running tests against vulnerable contract

reported that more than 60% of contracts used this pattern. An example of a popular contract using this pattern is the *Ownable.sol* contract which can be found in the OpenZeppelin library, which is a library for secure smart contract development [9, 15, 53, 68].

8.1.2.2 Comparison against truffle

Listing 8.5 is the same test written using the truffle testing framework.

```
1 // test initialisation removed
2
3 contract('MultiOwnable', ([owner, not_owner]) => {
4   let m0;
5
6   beforeEach('setup contract for each test', async function () {
7     m0 = await MultiOwnable.new(owner)
8   })
9
10  it('should revert when trying to add owners', async () => {
11    await truffleAssert.reverts(m0.newOwner(not_owner, {from:
12      not_owner}));
13  });
14 });
```

Listing 8.5: Truffle test for SWC 105

8.1.2.3 Conclusion

Testing caller capabilities is a useful. It enables developers to mitigate against known vulnerabilities and implement the access control pattern safely. Comparing Truffle and Flint-Test, we say that the Flint-Test test is clearer and communicates the intent of the test better. Specifically, we argue that the use of `setAddr` and `unsetAddr` is better than setting the `from` field of each transaction. It is easier to see when the address context changes, it is also less cumbersome to send multiple transactions using the same address. Additionally, the use of the specific assert means that the programmer can see quickly what is being tested, the Truffle assert is generic.

8.1.3 Testing Access Control : States

8.1.3.1 How is it useful?

Flint supports type states which enables developers to partition a contract into a set of states and restrict access to functions depending on the state of the contract. The testing framework allows developers to write tests to assert which functions are accessible in a particular state. Partitioning contracts into states is a common smart contract design pattern, known as the state transition pattern, and applicable to many contract domains [14, 69]. Some examples of the state transition pattern being used in the real world can be found in the betting contract from Ethhorse which enables users to bet on cryptocurrencies and also the auction contract owned by PocketInns [27, 58].

This pattern also can be used to implement other smart contract design patterns, an example is the Emergency Stop design pattern. This pattern is used to provide users with the option to disable a contracts critical functionality in the event of an emergency. An example of this pattern implemented in Solidity can be found in the Pausable.sol contract which is in the OpenZeppelin library [54].

To demonstrate why it is useful for the test framework to support testing of access control in contract states, we will show the first steps of developing a smart contract that implements the Emergency Stop pattern. We will use the testing framework to do this using TDD (Test Driven Development) so we can implement the contract safely and reliably.

The contract will have two states: Running and Paused, and three functions: run, pause and unpause. In the Running state, any user should be able to call the run function and the owner of the contract should be able to call the pause function. In the Paused state, the owner of the contract should be able to call the unpause function and no other function should be available. We will develop this contract incrementally starting with the run function. Before we implement the contract we will write a test to verify that the run function can only be called when the contract is in the Running state. Listing 8.6 contains the code sample used to test that the run function cannot be called in the paused state, the contract by default is in the paused state.

```
1 contract TestEmergency {
2     // initialisation removed
3 }
4
5 TestEmergency :: (any) {
6
7     public func test_cannot_run_in_paused_state() {
8         let owner : Address = newAddress()
9         let e : Emergency = Emergency(owner)
10
11         assertCantCallInThisState("run")
12     }
```



```

13
14 }

```

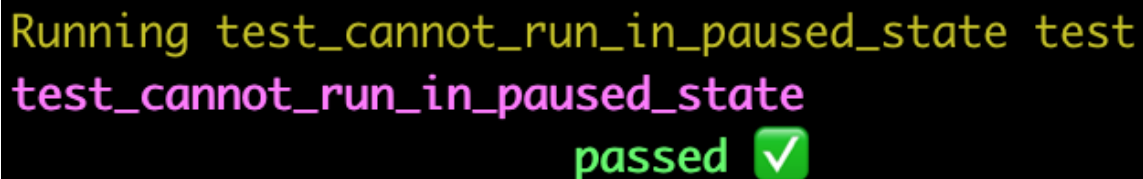
Listing 8.6: Flint test file for Emergency Contract

Listing 8.7 is the Flint contract that we are writing. The test passes which indicates to the developer that they have implemented proper access control for the run function. Figure 8.2 is the output of the test framework.

```

1  contract Emergency (Running, Paused) {
2      var owner: Address
3  }
4
5
6  Emergency @(Paused) :: (any) {
7      public init(initial_owner: Address) {
8          self.owner = initial_owner
9          become Paused
10     }
11 }
12
13 Emergency @(Running) :: (any) {
14
15     public func run() {
16         // useful function for users
17     }
18 }

```

Listing 8.7: Emergency Contract Flint


```

Running test_cannot_run_in_paused_state test
test_cannot_run_in_paused_state
passed ✓

```

Figure 8.2: Output of test framework

The development of the rest of the functions in the contract can continue in a similar way. By using the testing framework and engaging in test driven development the developer can be sure they are implementing the pattern correctly and reduce the likelihood of bugs appearing later in production.

8.1.3.2 Comparison against truffle

Listing 8.8 is the same test written for the Truffle framework. The associated solidity code is in the appendix (Appendix E.2).

```

1 // test initialisation removed
2
3 contract('Emergency', ([owner]) => {
4     let e;
5

```

```
6   beforeEach('setup contract for each test', async function () {
7       e = await Emergency.new(owner)
8   })
9
10  it('should revert when trying to call run in paused state', async
    () => {
11      await truffleAssert.reverts(e.run());
12  });
13
14  });
```

Listing 8.8: Truffle test for testing access control in states

8.1.3.3 Conclusion:

Testing access control within states enables developers to program the state transition pattern safely and other patterns that depend on it. Since this is a common pattern and applicable to many contract domains, we argue that this is useful feature to include in the testing framework and will help developers write smart contracts more safely. Comparing Flint-Test and Truffle, we say they are similar in expressiveness. However, it is clearer to see what is happening in the Flint-Test test as the contract initialisation and the creation of new users all occur within the test. The Flint-Test assertion is also more expressive and expresses clearly what the assertion is testing.

8.1.4 Testing Events

8.1.4.1 How useful is it?

The only way for the blockchain to communicate with the external world is via events. Events are integral to creating decentralised applications, external applications can monitor blockchain events and react appropriately e.g. Update UI. Effective testing of events will allow users to check that the blockchain is emitting the right information such that the front end of their decentralised applications work correctly. Thus, we believe that this feature is a useful addition to the testing framework and will help Flint developers create robust decentralised applications.

8.1.4.2 Comparison to Truffle

Listing 8.9 is an example of a test which verifies that a contract has emitted an event. The contract being tested is a simple contract which represents a counter that can be incremented. The test is checking that whenever the counter value is incremented, an event is emitted with the current value of the counter. Listing 8.10 is the same test written for Truffle. Listing 8.11 is the Flint contract which is being tested, the equivalent Solidity contract can be found in the appendix (Appendix E.2)

```
1  contract TestCounter {
2      // test init removed
3  }
```

```

4 TestCounter :: (any) {
5
6     public func test_event_fired() {
7         let c : Counter = Counter()
8         c.increment()
9         assertEventFired(c.counterVal, val: 1)
10    }
11
12 }

```

Listing 8.9: Testing events Flint-test

```

1 // remove test init
2 contract('Counter', ([owner]) => {
3     let c;
4
5     beforeEach('setup contract for each test', async function () {
6         c = await Counter.new();
7     })
8
9     it('emit counter state when incremented', async () => {
10         let tx = await c.increment();
11         truffleAssert.eventEmitted(tx, 'emitVal', (ev) => {
12             return ev.val == 1;
13         });
14     });
15
16 });

```

Listing 8.10: Testing events Truffle

```

1 contract Counter {
2     var value: Int = 0
3     event counterVal(val: Int)
4 }
5
6 Counter :: (any) {
7     public init() {}
8
9     mutating public func increment() {
10         self.value += 1
11         emit counterVal(val: self.value)
12     }
13 }

```

Listing 8.11: Counter contract written in Flint

8.1.4.3 Conclusion

In conclusion we can say that testing for events is a useful feature as events forms a central part of building decentralised applications. The Flint-Test test for checking events have fired is more concise then the Truffle test equivalent. However, the Truffle assert for events is more expressive as it allows one to apply complex filtering

logic via the use of a closure to the events returned from the blockchain. In comparison, the Flint-Test assert only allows you to check for equality. This is a limitation of the approach I have taken as I have chosen to keep the testing DSL as close as possible to Flint syntax and semantics. Flint does not support the use of function closures or functions as types thus it is not possible for users to specify a more complex filtering logic for events. Currently, only equality on event arguments is supported. Additionally, Truffle enables developers to check events per transaction, the Flint-Test framework does not offer this level of granularity. However, we did not think it was necessary to offer this functionality as the framework automatically ensures that only the event logs of contracts deployed within the test are checked.

8.1.5 General Testing Of Contracts

8.1.5.1 How useful is it?

In this section, we will demonstrate why it is valuable for the testing framework to enable programmers to deploy contracts and call functions on them as part of a test.

Listing 8.12 is a Solidity contract with the SWC-119 vulnerability present. Listing 8.13 is a Flint contract with the same vulnerability. The function parameters are shadowing the state variables. This behaviour can be a source of bugs especially in more complex systems where the existence of the issue is not as obvious. The Solidity compiler will warn the programmer when this happens, the Flint compiler does not. However, in both languages this vulnerability can be mitigated by testing and verifying the contracts are behaving as intended. Listing 8.14 is a Flint-test file which can be used to check for this vulnerability. By engaging in TDD the programmer can detect this bug during the development process.

```

1 contract ShadowingInFunctions {
2     uint n = 2;
3
4     function test1(uint n) constant returns (uint) {
5         return n; // shadowing state variable
6     }
7 }

```

Listing 8.12: Contract with the SWC-119 Vulnerability

```

1 contract Shadow {
2     let n : Int = 2
3 }
4
5 Shadow :: (any) {
6     public init() {}
7
8     public func test1(n :Int) -> Int {
9         return n // shadowing contract variable
10    }
11
12 }

```

Listing 8.13: Flint contract with vulnerability SWC-119

```

1  contract TestCounter {
2      // test init removed
3  }
4
5  TestCounter :: (any) {
6      public func test_test1() {
7          let s : Shadow = Shadow()
8          let x: Int = s.test1(1)
9          assertEquals(x, 2)
10     }
11 }

```

Listing 8.14: Flint test for vulnerability SWC-119

8.1.5.2 Comparison to Truffle

Listing 8.15 is an example of a truffle test to check for vulnerability SWC-119.

```

1  // remove test init
2
3  contract('Shadow', ([owner]) => {
4      let s;
5
6      beforeEach('setup contract for each test', async function () {
7          s = await Shadow.new();
8      })
9
10     it('emit counter state when incremented', async () => {
11         let bigNumVal = await s.test1(1);
12         let value = bigNumVal.toNumber();
13         assert.equal(2, value, "should be equal to 2");
14     });
15
16 });

```

Listing 8.15: Truffle test for SWC-119

8.1.5.3 Conclusion

In terms of expressiveness and conciseness of tests for the SWC-119 vulnerability, we say the Flint-Test test is more concise. Specifically, the Truffle test requires you to convert values returned back from the blockchain to an appropriate Javascript type for further use in the test. This is not required with Flint-Test as the test runner takes care of the conversion between blockchain types and Flint types. This removes the burden from the developer leading to shorter and more concise tests. We argue from a usability perspective, writing tests in the Flint-Test framework is easier in that regard. In terms of expressiveness, for general testing, the two testing frameworks are similar.

8.1.6 Testing For Exceptions

8.1.6.1 How useful is it?

Exceptions are an important of programming, they allow programmers to handle errors gracefully and in a predictable manner. As such it is useful to be able to test that the exception handling code is working correctly. Flint-Test supports the testing of exceptions by enabling programmers to write asserts which check that an exception is thrown.

8.1.6.2 Comparison To Truffle

Listing 8.16 is an example of a test written using Flint-Test which checks exception handling code. Listing 8.17 is the same test written in JavaScript for the Truffle framework. Listing 8.18 is the same test written in Solidity for the Truffle Framework. The contract which is being tested is shown in Listing 8.19 (Flint Version).

```

1  contract TestCounter {
2      // test init removed
3  }
4  TestCounter :: (any) {
5
6      public func testExceptionThrown() {
7          let c : Counter = Counter()
8          assertWillThrow(c.willThrow)
9      }
10 }
```

Listing 8.16: Tests Which Check Exception Handling Code (Flint-Test)

```

1  // test init removed
2  contract('Counter', ([owner]) => {
3      let c;
4
5      beforeEach('setup contract for each test', async function () {
6          c = await Counter.new();
7      })
8
9      it('throw exception', async () => {
10         await truffleAssert.reverts(c.willThrow());
11     });
12
13 });
```

Listing 8.17: Test Which Check Exception Handling Code (Truffle JavaScript)

```

1  // test init removed
2  contract TestCounter {
3
4      function testExceptionThrown() public {
5          Counter counter = Counter(DeployedAddresses.Counter());
6          bool statusOfCall;
7      }
```

```

8      (statusOfCall, ) =
        address(this).call(abi.encodePacked(counter.willThrow.selector));
9
10     Assert.IsFalse(statusOfCall, "Exception was not thrown");
11
12 }
13 }

```

Listing 8.18: Test Which Check Exception Handling Code (Truffle Solidity)

```

1 contract Counter {
2     var val : Int = 0
3 }
4
5 Counter :: (any) {
6     public init() {}
7
8     public func willThrow() {
9         fatalError()
10    }
11 }

```

Listing 8.19: Flint Contract Which Throws Exceptions

8.1.6.3 Conclusion

Testing for exceptions is good programming practice. Exception handling code is code that may be run, thus it should be tested to ensure it is working correctly. Comparing to Truffle, we can see that in terms of expressiveness and conciseness the JavaScript test is similar to the test written for Flint-Test. To test for exceptions in Solidity, one has to first encode the function they are going to call using the ABI encoder and then send a call to the function. The call returns a boolean which indicates if the call has succeeded or not. We argue that the approach used by Flint-Test is better as it is less verbose and it is easier to see what the intent of the test is. It is not clear what the Solidity test is trying to do, the programmer would most likely have to leave accompanying comments.

8.1.7 Testing Ether Transactions

To support testing of functions which accept money, the framework enables developers to send Wei as part of a function call.

8.1.7.1 Comparison To Truffle

Listing 8.20 is an example of a Flint-Test test which sends money. Listing 8.21 is an example of a Truffle test which sends money. Listing 8.22 is the contract which is being tested (Flint Version).

```

1 contract TestCounter {
2     // test init removed
3 }

```

```

4
5 TestCounter :: (any) {
6
7     public func test_was_bribed() {
8         let b : Bribe = Bribe()
9         b.bribe(_wei: 100)
10        assertEventFired(b.Bribed, true)
11    }
12
13 }

```

Listing 8.20: Flint-Test Which Sends Money To A Function

```

1 // test init removed
2 contract('Bribe', ([owner]) => {
3     let b;
4
5     beforeEach('setup contract for each test', async function () {
6         b = await Bribe.new();
7     })
8
9     it('test which sends money', async () => {
10        let tx = await b.bribe({value:150});
11        truffleAssert.eventEmitted(tx, 'Bribed', (ev) => {
12            // if we reach here, then event was fired
13            return true;
14        });
15    });
16
17 });

```

Listing 8.21: Truffle JavaScript Test Which Sends Money To A Function

```

1 contract Bribe {
2     var value : Int = 0
3     event Bribed(bribe: Bool)
4 }
5
6 Bribe :: (any) {
7     public init() {}
8
9     @payable
10    public func bribe(implicit w: Wei) {
11        let rawVal : Int = w.getRawValue()
12        var bribed: Bool = false
13        if (rawVal > 100) {
14            bribed = true
15        }
16
17        emit Bribed(bribe: bribed)
18    }
19 }

```

Listing 8.22: Flint Contract Which Accepts Money

8.1.7.2 Conclusion

Exchanging currency between accounts (users & contracts) is an important component of programming smart contracts. Thus, for the testing framework to be effective and be relevant to a large class of contracts, it needs to be able to support sending money to contracts that are being tested. Comparing to Truffle, we can see that both frameworks support sending Ether in a similar way. The programmer attaches the amount of Ether they wish to send as part of the function call. However, we argue that Flint-Test is clearer as the argument is labelled with the denomination of Ether that is being sent. Whereas the argument is labelled with `value` in the Truffle test. It is not immediately clear what denomination of Ether is being sent.

8.1.8 Code Coverage

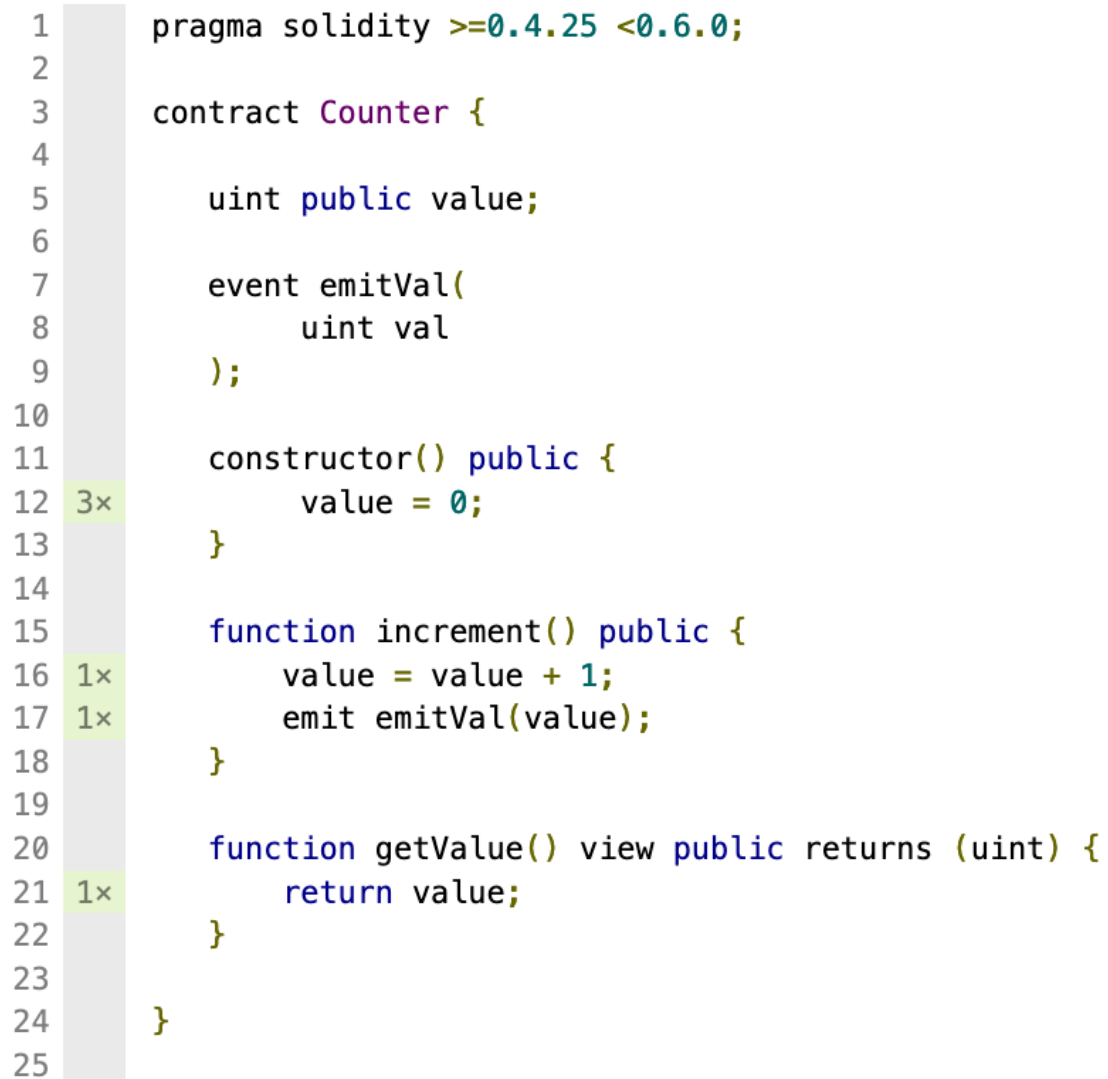
Truffle does not support code coverage by default. However, with the use of external libraries Solidity developers can measure code coverage. *Solidity-Coverage* & *Sol-Coverage* are popular tools which can measure code coverage. The tools use Istanbul to generate a code coverage report [64, 1]. We will now compare the report generated by these tools to the one generated by Flint-Test. We used *Solidity-Coverage* to generate the report. We did not generate one using *Sol-Coverage* as they both use Istanbul to generate the report. All three tools measure function, line and branch coverage. Figure 8.3 is an example of a report generated by *solidity-coverage*. Figure 8.4 is an example of a coverage report generated by Flint-Test. The reports present the information to the programmer in a similar way. Apart from style differences there is no significant difference between the two reports. Both tools also provide a summary table which summarises the coverage data in the report. Additionally, our tool provides the programmer with a table that documents how many times a function was called. The other two tools do not. Figure 8.5 is an example of such a table.

8.1.8.1 Strengths:

- Provides visibility on the codebase, uncovers untested code.
- Measures three types of coverage: branch, line and function coverage.
- Easy to run, comes as part of the testing framework. All you need to do is pass in a command line flag.
- Generates an LCOV report which presents the coverage data to the user as an easy to read HTML report which includes a table that reports how many times each function was called.

8.1.8.2 Limitations:

- We do not measure condition coverage or path coverage.



```
1  pragma solidity >=0.4.25 <0.6.0;
2
3  contract Counter {
4
5      uint public value;
6
7      event emitVal(
8          uint val
9      );
10
11     constructor() public {
12         value = 0;
13     }
14
15     function increment() public {
16         value = value + 1;
17         emit emitVal(value);
18     }
19
20     function getValue() view public returns (uint) {
21         return value;
22     }
23
24 }
25
```

Figure 8.3: Coverage Report Generated By *solidity-coverage* [64]

	Branch data	Line data	Source code
1		:	: contract Counter {
2		:	: var value : Int = 0
3		:	: event counterVal(val: Int)
4		:	: }
5		:	:
6		:	:
7		:	: Counter :: (any) {
8		:	:
9		:	: public init() {}
10		:	:
11		:	: public func getValue() -> Int {
12		: 1 :	: return value
13		:	: }
14		:	:
15		:	: mutating public func increment() {
16		: 1 :	: self.value += 1
17		: 1 :	: emit counterVal(val: self.value)
18		:	: }
19		:	: }

Figure 8.4: Coverage Report Generated By Flint-Test

Function Name ↕	Hit count
getValue	1
increment	1

Figure 8.5: Function Hits Table

- We use events to instrument the contract, thus if a transaction reverts then we lose coverage data as the blockchain does not store events from reverted transactions. *Solidity-Coverage* has the same limitation because it uses events for instrumentation. However, this limitation can be overcome if transaction traces are used instead of events to generate the coverage data. The blockchain can provide transaction traces for reverted transactions. *Sol-Cover* uses transaction traces. Please refer to section 2.6.3.2 for more information on how to instrument contracts using transaction traces.

8.1.9 Feature Comparison

In this section, we will do a feature comparison between Flint-Test and the Truffle testing framework. For the Truffle testing framework, we will compare with both the Javascript and Solidity versions.

Clean Room Environment (Per Test)

Truffle Javascript, Truffle Solidity and Flint-Test all support a clean room environment per test. This means that each unit test is run in isolation and tests do not share state.

Basic Testing Of Contracts

Truffle Javascript, Truffle Solidity and Flint-Test all enable you to deploy and interact with contracts.

Inbuilt Assertion Library

Truffle Javascript, Truffle Solidity and Flint-Test all come with a basic assertion library.

Testing Caller Protections

Truffle Javascript, Truffle Solidity and Flint-Test all have the ability to test caller protections.

Testing State Protections

Truffle Javascript, Truffle Solidity and Flint-Test all have the ability to state protections.

Access To Node Ecosystem And Libraries

Only Truffle Javascript supports access to the node ecosystem and libraries.

Testing For Exceptions

Truffle Javascript, Truffle Solidity and Flint-Test all have the ability to test for exceptions.

Testing Ether Transactions

Truffle Javascript, Truffle Solidity and Flint-Test all have the ability to send money to contracts during a test.

Code coverage

Only the Flint-Test comes with inbuilt code coverage support. The Truffle testing framework does not have any support for code coverage, however it can be added via the use of external libraries.

Stack Traces

The blockchain does not return any stack traces when a transaction fails. This is in part due to the fact that when a transaction fails, the blockchain undoes any effect of the transaction. However, certain Ethereum clients support the ability to trace transactions which have failed. The trace returns back all the op-codes that were executed as part of this transaction. *SolTrace* is a library which can take Solidity

source files and a trace from a blockchain and use them to create a stack trace [70]. Thus, both Truffle Javascript and Truffle Solidity can both support stack traces with external library support. Flint-Test cannot do this yet as the the Flint compiler cannot yet generate mappings between the high Flint code and its EVM assembly representation.

Debugging Capacity

The Truffle debugger can be used to debug failing transactions, thus developers can use this to debug failing tests by debugging the transactions they produce. The Truffle debugger works with Solidity source files. There does not currently exist a debugger for Flint.

Optimised For Speed

Truffle makes use of a custom blockchain called Ganache to run automated unit tests really fast [89]. Currently, Flint-Test runs tests on a blockchain created using Geth, this limits the speed of the test framework. Please refer to section 8.1.1 for more details on this.

8.1.10 Summary

In this section, we will summarise the key strengths and limitations of the testing framework.

Strengths:

1. Testing framework enables you to mitigate against known smart contract vulnerabilities and implement common smart contract design patterns safely.
2. Enables developers to test caller protections, state protections, exceptions and events. It also supports sending Ether to contracts during a test.
3. Each test is run in a clean room environment, tests do not share contract state.
4. Flint tests are generally more expressive than the equivalent Truffle tests. The assertions are more expressive
5. Code coverage is inbuilt.

Limitations:

1. The Flint testing framework is slow compared to Truffle. This limitation exists because the backing blockchain for Flint-Test implements the complete Ethereum protocol. It limits the number of transactions that can be processed per second. Please refer to section 8.1.1 for more details on why this limitation exists and how it can be overcome.

2. As Truffle allows you to write tests in Javascript, developers have access to the vast node ecosystem and a familiar development environment (node). Flint-Test does not have any libraries other than the ones provided by the framework and has no integration with Javascript.
3. The Flint-Test DSL is small thus is limited in what it can express e.g. we do not yet support for loops. However, with time we hope the DSL will grow and become more expressive.

8.2 REPL (Interactive Console)

The primary goal of the REPL is to provide a mechanism for developers to deploy their contracts to a local blockchain and interact with them in a quick and convenient manner. To evaluate the REPL, we will assess how easy it is to interact with contracts using the REPL. We will also compare our REPL with the Truffle REPL. To perform this evaluation, we will go through a process of deploying a contract using the REPL and interacting with it. We will evaluate each step by executing the same step in Truffle and then comparing. We will also list the strengths and limitations of the REPL.

8.2.1 Deploying A Contract

In this section, we will examine how the two REPL's enable developers to compile and deploy contracts to a local blockchain.

8.2.1.1 Flint REPL:

Compiling and deploying a contract is a two step process. The first step is to launch the REPL specifying the contract file path as an argument to the CLI. The REPL then compiles the contract and makes it available for deployment. Additionally, it also shows in a readable format the functions that can be called on that contract and their attributes. Figure 8.6 illustrates how a user launches the REPL, it also shows the information that is presented to the user when a contract is loaded. Figure 8.7 illustrates how a user would deploy a contract in the Flint REPL, it also shows what is returned to the user once a contract has been deployed.

8.2.1.2 Truffle REPL:

Compiling and deploying a contract is a multi step process in Truffle. The user is first required to run *truffle develop*, then run the command *compile* and then *deploy* the contract. Figure 8.8 is a screenshot of running *compile* in the Truffle REPL. Figure 8.9 illustrates how to deploy a contract in the Truffle REPL. Printing the contract instance, prints the entire contract object representing this contract. The screenshot is too large to include in this report.

```
demo>flint-repl repl_eval/Counter.flint

Counter:

Contract Functions:
getValue() (Constant)
bribe(_wei: Wei, val : Int) (Mutating, Payable)
increment() (Mutating)

flint>
```

Figure 8.6: Launching a Flint REPL with a contract

```
flint>let c : Counter = Counter()
Deploying c : Counter
Contract deployed at address: 0x5d27a8d7efba6577b316867076826978bac083df
flint>
```

Figure 8.7: Deploying a contract in the Flint REPL

```
truffle(develop)> compile

Compiling your contracts...
=====
> Compiling ./contracts/Counter.sol
> Compiling ./contracts/Migrations.sol
> Artifacts written to /Users/Zubair/Documents/Imperial/Thesis/Code/metacoins/build/contracts
> Compiled successfully using:
   - solc: 0.5.0+commit.1d4f565a.Emscripten.clang

truffle(develop)>
```

Figure 8.8: Running compile using Truffle REPL

```
truffle(develop)> var counter;
undefined
truffle(develop)> Counter.new().then(function(instance) {counter = instance;});
```

Figure 8.9: Deploying a contract using Truffle REPL

8.2.1.3 Evaluation

It requires more steps to compile and deploy contracts using Truffle REPL in comparison to Flint-REPL. Also, the Truffle console requires more verbose syntax to deploy the contract and the syntax is different from what is used to deploy contracts in test files. The Flint-REPL enables developers to deploy contracts using Flint syntax which is the same as what would be used in test files. It also presents more useful information to the programmer about the contracts that the REPL is aware of. Thus, we argue that it is easier to compile and deploy contracts using the Flint-REPL.

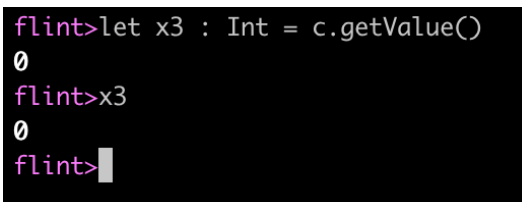
The Truffle REPL supports loading additional contracts whilst the REPL is running, the Flint REPL does not support this. The Flint-REPL has to be relaunched if the developer wishes to deploy different contracts.

8.2.2 Calling Functions

In this section, we will examine how the two REPLs enable developers to call functions on a contract.

8.2.2.1 Flint REPL:

Figure 8.10 illustrates how one would call a function which reads the state of the blockchain i.e. the value of the counter in the Flint-REPL. The value returned is an integer.



```
flint>let x3 : Int = c.getValue()  
0  
flint>x3  
0  
flint>
```

Figure 8.10: Calling a Function On a Contract (constant function) Using The Flint REPL

8.2.2.2 Truffle REPL:

Figure 8.11 illustrates how one would call the same function using Truffle. The REPL returns a `BigNumber` object which is displayed in the screenshot. The programmer is required to manually convert the value from `BigNumber` to `Integer`.

8.2.2.3 Evaluation

In terms of syntax, both calls are very similar. The Flint REPL automatically converts the values returned from the blockchain back into types which are familiar to the user. However, in the case of integers the Truffle REPL returns a `BigNumber` object instead of an integer. It is not entirely obvious what number the object represents


```
truffle(develop)> let x3 = counter.getValue()
undefined
truffle(develop)> x3
BN {
  negative: 0,
  words: [ 1, <1 empty item> ],
  length: 1,
  red: null }
truffle(develop)> |
```

Figure 8.11: Calling a function on a contract (constant function) using Truffle REPL

and the programmer will have to convert it into an integer. It also means that the programmer is limited in what they can do with functions that return a big number, at the moment the Truffle REPL does not support the addition of big numbers. Thus, there is an additional overhead to the programmer if they wish to construct complex expressions using multiple function calls. Figure 8.13 is an example of an expression where the sub-expressions are function calls in the Flint-REPL. The Flint-REPL can process each of the calls and perform the expression evaluation. Figure 8.12 shows the same calculation being performed in the Truffle REPL. The Truffle REPL currently does not support arithmetic expressions where the sub-expressions are contract calls. The developer is required to process each of the sub calls separately. We argue that the Flint REPL is more usable and puts less of a burden on the programmer.

```
truffle(develop)> let x5 = counter.getValue() + counter.getValue()
undefined
truffle(develop)> x5
'[object Promise][object Promise]'
```

Figure 8.12: Expressions using Truffle REPL

```
flint>let x : Int = c.getValue() + c.getValue()
2
flint> |
```

Figure 8.13: Expressions using Flint REPL

8.2.3 Sending Money To A Contract

In this section, we will examine how the two REPL's enable developers to send money to a contract.

8.2.3.1 Flint REPL:

Figure 8.14 illustrates how one would call a function and send money to a contract in the Flint-REPL. The `_wei` parameter specifies how much Wei will be sent.

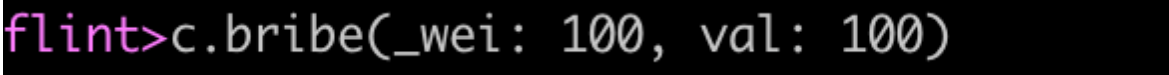
A screenshot of the Flint REPL interface showing a command to send money to a contract. The text is `flint>c.bribe(_wei: 100, val: 100)` in a monospaced font on a dark background.

Figure 8.14: Sending money to a contract using Flint REPL

8.2.3.2 Truffle REPL:

Figure 8.15 is an example of how a user can send money to a contract using the Truffle-REPL. The JavaScript object is used to specify hyper parameters for the function call, the value mapping it is used to specify the amount of Ether that should be sent. The remaining arguments are function arguments.

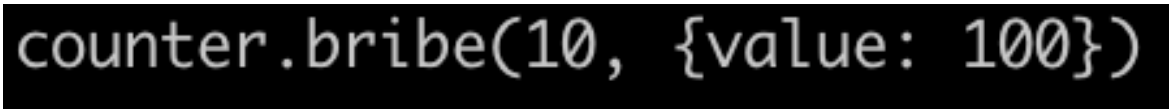
A screenshot of the Truffle REPL interface showing a command to send money to a contract. The text is `counter.bribe(10, {value: 100})` in a monospaced font on a dark background.

Figure 8.15: Sending money to a contract using Truffle REPL

8.2.3.3 Evaluation

The command for sending money in both Truffle and Flint are very similar. The return values for each of the calls differ. Truffle returns a transaction object whereas Flint returns a transaction hash. Sending currency to a contract is a central part of programming on the blockchain. Thus, we believe that this is a necessary feature to include in the REPL and is something that will be useful for smart contract developers. In terms of usability and expressiveness, we match the Truffle REPL. We cannot see any significant way to improve how the programmer sends money to the contract. However, we argue that the Flint-REPL is slightly better than the Truffle REPL as the use of the label `_wei` makes it clear what denomination of Ether is being sent.

8.2.4 Querying Event Logs

In this section, we will examine how the two REPL's enable developers to query the event logs of a contract.

8.2.4.1 Flint REPL:

The Flint REPL allows you to query event logs using dot syntax. The REPL returns the number of times the event was fired and also returns an array of logs which contains each invocation of the event. The logs are processed so that the displayed result looks like Flint. The REPL labels each of the event arguments and converts blockchain types into Flint types.

```
flint>c.increment()  
0xad25de61e5ebe21a3fdb73b6da4b57411cb93e79132b90daca67e5be046e81cd  
flint>c.CounterVal  
{  
(val: 1),  
(val: 2),  
}  
Fired: 2  
flint>
```

Figure 8.16: Querying a contract event in the Flint REPL

8.2.4.2 Truffle REPL:

You cannot query the event logs of a contract using the Truffle REPL. However, Truffle does support querying the event logs of a transaction. Figure 8.17 is an example of using the Truffle REPL to view the events that were fired as part of a transaction.

8.2.4.3 Evaluation

Comparing the two REPL's, we can see that the presentation of events in the Flint REPL is clearer and more concise. The Truffle REPL prints the underlying log object straight from the blockchain and does not convert any of the blockchain types into types that the programmer uses in the source file. It also does not name the arguments and we argue it presents a lot of superfluous information such as the block number which is not useful to the programmer when they are interested in the events that have been fired. The Flint REPL removes this complexity and converts the underlying logs into something which is similar to the event declaration syntax in Flint. Each of the arguments are converted into types that the programmer is used to. In terms of the syntax, a direct comparison is not possible as the two REPL's offer slightly different functionality. Truffle enables a programmer to check which events have fired per transaction whereas the Flint REPL enables you to check which events have fired since the beginning of the contract. We argue that the approach the Flint-REPL takes is more useful as it enables a programmer to see what events have fired per transaction but also all of the events which have been fired since the beginning of the contract.

```
truffle(develop)> let result = await counter.increment()
undefined
truffle(develop)> result.logs[0]
{ logIndex: 0,
  transactionIndex: 0,
  transactionHash:
    '0x799dfdcc9d686cad326de29df676a1b14d261f29955948c5e2ddf67823c4829d',
  blockHash:
    '0xc7445aaf49616bbda292fe61831b7ef8865d08ada5bfeb3ff5bc3090d86d8ca0',
  blockNumber: 11,
  address: '0x54E0E7FfB5af530Fd1515dC0A6f316E14Cb3D9f3',
  type: 'mined',
  id: 'log_025d6d22',
  event: 'CounterVal',
  args:
    Result {
      '0': BN { negative: 0, words: [Array], length: 1, red: null },
      __length__: 1,
      _value: BN { negative: 0, words: [Array], length: 1, red: null } } }
truffle(develop)> █
```

Figure 8.17: Querying a contract event in the Truffle REPL

8.2.5 Summary

In this section, we will summarise the above evaluation of the REPL and list the key strengths and limitations.

Strengths:

- Abstracts complexity of underlying blockchain and presents information to the user in a format that is close to Flint syntax.
- Enables a programmer to deploy and interact with a contract. This includes calling functions, querying event logs and sending money to the contract.
- Permits programmers to construct arithmetic expressions where the sub-expressions are function calls. Truffle does not support this.
- It is easier to deploy and compile contracts in the Flint-REPL (compared to the Truffle REPL).

Limitations:

- Does not support loading of additional contracts once the REPL is running. The REPL has to be relaunched with a new file.
- Currently only supports loading a single Flint file.

- The Flint REPL hides away all the complexity of the underlying blockchain. This can be seen as both a strength and a weakness. It is a limitation because it does not offer the programmer the flexibility to observe or modify the underlying blockchain via the REPL. Truffle REPL supports this via a complex contract abstraction which enables users to modify block properties such as block timeout and block limit.
- Requires the developer to be running a local blockchain which the REPL connects to. Truffle does not require a local blockchain to be running, the REPL comes packaged with a blockchain.

8.2.6 Conclusion

Overall, the Flint REPL enables developers to deploy and interact with smart contracts in a user friendly manner. It has useful features enabling a programmer to call functions on the contract and query events. The main limitations are it does not support dynamic loading of contracts and does not offer the programmer the ability to observe or modify the underlying blockchain.

8.3 Language Server & Contract Analysis

In this section, we will evaluate the language server and the contract analysis tools. There exist some tools in the Flint ecosystem which target this aspect of development but they are limited in scope. Thus, we will not be comparing our implementation against them. Instead, we will compare with tools from the Solidity ecosystem as they are more developed. However, since Flint is different from Solidity with regards to syntax and semantics, the tools are not directly comparable. For this evaluation, we will compare when possible. As it is not possible to compare with every single tool in the Solidity ecosystem, we have chosen a set of tools that we will compare with. We have chosen these tools based on their popularity and the functionality that they offer. We use the following Solidity tools as points of comparison throughout the evaluation. Remix IDE which is a popular browser based Solidity IDE released by the Ethereum foundation [62]. Yakindu Solidity Tools which is a set of Solidity Tools maintained and developed by Yakindu [95]. They have created a Solidity IDE with many features to aid Solidity development. The Solidity VSCode plugin by Juan Banco which is a highly popular Solidity plugin found in the VSCode extension market place [44]. Solgraph which is a Solidity contract visualiser that can be used to visualise the functions of a Solidity contract and highlight potential security vulnerabilities [61]. The purpose of this evaluation will be to assess the strengths and limitations of the language server and the contract analysers.

8.3.1 Language Server

We currently implement a language server and a visual studio code extension to support development of Flint contracts in VS Code. The extension supports syntax

highlighting and semantic & syntax error detection. The language server makes use of a heuristic error tolerant parser which enables it to detect multiple syntax errors. For this evaluation, we will only consider the Yakindu, Remix IDE and solidity VSCode tools. Solgraph does not provide any editor features thus will not be used in this section of the evaluation.

Strengths:

- Provides live validation for Flint syntax and semantic errors. Remix IDE does not support live validation of Solidity code. Yakindu and Solidity VSCode both do.
- Includes an error tolerant parser thus it can detect multiple syntax errors.
- Syntax Highlighting. All three Solidity tools being considered support syntax highlighting.
- Implemented using the language server protocol thus it can be easily ported to any editor which implements the language server protocol. Only the Yakindu Solidity Tools currently implement the language server protocol.

Limitations:

- We do not yet support code completion. However, this is a planned feature and will be implemented if time permits.
- We do not yet support quick fixes. Yakindu supports quick fixes, the other two do not.
- Easy code navigation, only Yakindu supports quick code navigation.
- No code formatting tools.

8.3.1.1 Conclusion

Overall, we believe that the language server is the first step in creating an IDE like environment for Flint developers. It currently supports providing live feedback from the compiler as the user edits code. However, for the system to be truly useful - it will need to support code completion, quick fixes and easy code navigation. This set of features will make it easy for the developer to write code and navigate around the codebase. The language server protocol supports all the aforementioned features thus they can be implemented as part of the Flint language server. Overtime, we also anticipate that Flint will have an associated style guide and a set of best practices which can be encoded into a Linter.

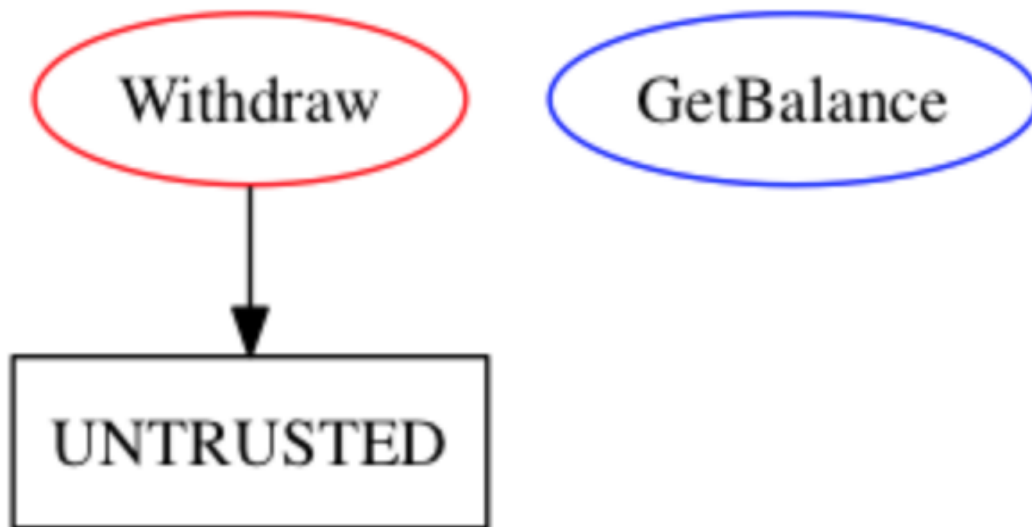


Figure 8.18: Solgraph Visualisation [61]



Figure 8.19: Flint-CA Function Visualisation [61]

8.3.2 Contract Analysis

In this section we will evaluate the contract analysis tools. The current analysis tools we provide are gas estimation, type state visualisation, function visualisation and protection analysis. For this section of the evaluation we will compare against the Remix IDE, Yakinu toolset and Solgraph when possible.

Strengths:

- Gas estimation for all the contract functions. None of the other tools we are considering have this feature. This is a useful feature because it gives visibility to the programmer on which functions are costly and helps the programmer understand the economics of their contract.
- Protection analysis. Provides information to the programmer in a tabular format about the protections their contract utilises. It also provides a metric on how much of their code is under the any caller protection. This is a useful analysis as it helps a programmer understand how well they are using Flint's

protection features. They can quickly see how the functions in the contract are protected and how much of their contract is exposed to all Ethereum users. We have found no comparable tool in the Solidity ecosystem for this analysis as Solidity does not require programmers to have protections.

- Visualising contract type states as a state diagram. Similar to the protection analysis, this tool offers a useful way for a programmer to understand their contract. It helps them see what states their contract can be in and how to move between the states. This can be used by the programmer to check that they are implementing their contract properly and it is behaving as expected with respect to type states. None of the other tools currently support visualisation of a contract as a state diagram. However, Yakindu are currently building a set of tools to enable programmers to visually construct smart contracts by creating a state diagram representing the contract which is then turned into a Solidity smart contract.
- We visualise contract functions. This visualisation enables a programmer to quickly see where money enters and leaves the contract and what functions mutate state. This is useful as the programmer can use this information to check for any potential security vulnerabilities e.g. money leaving the contract in an unexpected location. It can also be used to see how many functions in the contract interact with storage, instructions that interact with storage are expensive. It is in the interests of the programmer to minimise how much they interact with contract storage. Solgraph performs a similar function for Solidity smart contracts. Figure 8.18 is an example of a visualisation created by Solgraph. Figure 8.19 is the visualisation created by our tool for a similar contract. We say that the visualisations are very similar and there is no significant difference between the two tools. For more information on the colour schemes used. Please refer to sections 5 & 2.3.2.

Limitations:

- The gas estimation analysis may not be an accurate reflection of how much the contract may cost in a production setting. Our current approach involves simulating a transaction and seeing how much it costs to calculate gas estimates. For functions that make use of blockchain properties may cost different amounts of gas depending on the blockchain they are run on. This means if the test blockchain does not match the production blockchain then the estimates could be wrong. However, this is currently not an issue with Flint as it does not permit programmers to use features of the underlying blockchain.
- We have no formal verification tools. There are many formal verification tools in the Solidity ecosystem. The Solidity compiler has an inbuilt formal verification tool [66].
- We do not provide tools which can be used to monitor smart contracts which have been deployed to a blockchain. However, we argue that this is not neces-

sary because the tools in the Solidity ecosystem that perform this function can be reused for Flint as both Flint and Solidity target the Ethereum blockchain and they both compile down to the same assembly. The tools generally interface with the Blockchain and work at the EVM assembly level [17].

- We do not yet support analysis of Flint code to find problematic patterns. The main reason for this is because Flint is a very new language and as such there are no established patterns or anti patterns yet.

8.3.2.1 Conclusion

Overall, we believe we have created a set of analysis tools that enable developers to see how well they are leveraging Flint's safety features, help them understand the economics of their contracts and visualise the contract in meaningful ways.

8.4 Entire Ecosystem

In this section, we will evaluate the ecosystem as a whole.

8.4.1 Strengths & Limitations

Strengths:

- Contract analysis tools which help a programmer enforce Flint best practices, understand the economics of their contract and understand their contract in general.
- Live compiler feedback whilst the programmer is editing Flint source code in VSCode. The tool we implemented conforms to the LSP thus is easily portable to other editors. The extension also comes with a syntax highlighter.
- Unit testing framework which can exercise Flint's novel features and has inbuilt code coverage.
- REPL which facilitates easy deployment of contracts to test network and enables programmers to interact with the contract. This includes calling contract functions, querying contract events and sending money to a contract.
- We provide a tool that enables developers to launch a local blockchain.

Limitations:

- Lack of debugging tools and a profiler.
- Lack of formal verification tools.
- Language server does not support code completion and quick code navigation.

8.4.2 Conclusion

Overall, we believe that we have created a development ecosystem that targets the different aspects of development. We have tools that ease the writing and understanding of source code. A tool to make deploying and interacting with your contract easy. A testing framework with coverage to help the developer write smart contracts safely and reliably. Each of these tools have been developed with Flint features in mind and have been developed to help the developer utilise, test and leverage Flint's core features. We will not discuss these tools any further as we have evaluated them in previous sections. Compared to what existed before, the process of developing Flint contracts has been greatly improved. Developers can now write contracts, test them and deploy them to a local blockchain and interact with them. The ecosystem is currently missing tools for deployment and debugging. Additionally, the current tools especially the language server can be improved to make the developer experience better.

The Solidity ecosystem has a larger variety of tools to support the development of Solidity contracts. However, the tools in the ecosystem are developed by many different people and organisations. As such, there are many tools that perform the same function with varying degrees of success and there is not a consistent user interface between the tools. Additionally, the tools are generally not designed with interoperability in mind as they are developed by different parties. In comparison, the Flint ecosystem is a lot smaller but the ecosystem is a lot more coherent. All the tool present a similar interface to the user and we do not have an issue of multiplicity, each tool has a defined job. We hope as the ecosystem grows, this coherency is maintained.

Chapter 9

Conclusion

Modern development ecosystems provide a myriad of features that help programmers write, run, test and deploy code. The ecosystems help developers enforce best practices, write safer and more reliable code and make the process of development more convenient. The Ethereum platform provides a mechanism for developers to write programs (smart contracts) which run in a decentralised manner and provide complete transparency with their execution [94]. Flint is a safety oriented language designed to help programmers write safe smart contracts [31].

We designed a set of tools to aid Flint developers and help them write better, safer and more secure smart contracts. We developed a Flint language server that provides live code validation as the developer types, it reports both syntax and semantic errors. The language server is designed to provide rapid feedback to the developer as they type and saves them the hassle of manually compiling code to get feedback. The language server conforms to the language server protocol enabling it to be ported to multiple editors with minimal effort. We developed a set of contract analysis tools which provide insight to the programmer about their contracts. The analysis tools visualise Flint contracts that use type states as a state diagram, visualise contract functions, analyse the use of caller and state protections and provide gas estimates for each of the functions in the contract. Combined, these analyses enables developers to understand how well they are leveraging Flint's safety features and helps them understand the economics of their contracts. We have also developed a unit testing framework which enables developers to write tests to check the correctness of their contracts. The testing framework has mechanisms to help developers exercise all of Flint's safety features and also test for events and exceptions. The framework can generate a code coverage report to help programmers find untested parts of their codebase. We have also developed an interactive console which enables programmers to deploy their contracts to a local blockchain. This console allows programmers to interact with their contracts i.e. call functions and query event logs. We have also created a syntax highlighter to make it easier to read Flint code. Finally, we have created a tool which enables developers to launch a local blockchain.

9.1 Future Work

The current iteration of the ecosystem presents tools that help with the writing, running and testing of smart contract code. We have not implemented any debugging tools. We propose that an extension to this ecosystem would be to develop a full fledged debugger for Flint. The debugger should allow developers to step through a Flint contract, inspect state and evaluate expressions whilst debugging.

Ethereum nodes such as Geth permit a developer to trace transactions. Transaction traces contain valuable information about the execution of a transaction. They list all of the EVM assembly instructions that were executed as part of that transaction and for each instruction the trace also contains the contents of memory at the time the instruction was executed [81]. By using trace information and a mapping between Flint code and its EVM assembly representation, one can implement a debugger for Flint.

9.2 Challenges

- **Developing a set of disjoint tools.** The design of the ecosystem involved a notable amount of design and implementation. The implementation required over 5000 lines of code and used multiple languages (JavaScript, Swift & bash). Each tool had to be designed to be robust and extensible so that it is easy to build on this ecosystem. To ensure the tools we developed were relevant and useful we explored the Solidity ecosystem and researched software engineering practices to find out what tools developers required to develop smart contracts effectively.
- **Working with the Ethereum ecosystem.** This project required working extensively with the Ethereum ecosystem to develop the tools. The ecosystem is new and as such the documentation is not always up to date and the tools have bugs. Additionally, working with the blockchain proved to be a challenge because it provided little visibility when something went wrong, there are limited debugging tools and the primary language of the ecosystem is JavaScript.
- **Evaluating the ecosystem.** It was not immediately obvious how to evaluate the ecosystem. Especially because Flint is a very new language and as such there are not many developers working with it. However, we evaluated the ecosystem by demonstrating how it adds value for the developer and also compared how well we fare against the Solidity ecosystem.

Bibliography

- [1] *Ox Documentation*. URL: <https://0x.org/docs/sol-coverage%7B%5C#%7Dusage> (visited on 06/12/2019).
- [2] *1.3.Branch Coverage*. URL: <https://www.cs.odu.edu/%7B~%7Dcs252/Book/branchcov.html> (visited on 06/04/2019).
- [3] *About Code Coverage*. URL: <https://doc.froglogic.com/squish-coco/3.4/codecoverage.html%7B%5C#%7Dsec%7B%5C%7D3Abasic-block-coverage> (visited on 06/04/2019).
- [4] *Activation Events — Visual Studio Code Extension API*. URL: <https://code.visualstudio.com/api/references/activation-events> (visited on 06/08/2019).
- [5] Leonardo Alt and Christian Reitwiessner. “SMT-based Compile-time Verification of Safety Properties for Smart Contracts Work in progress”. In: (), pp. 1–9.
- [6] *Application Binary Interface Specification - Solidity 0.4.24 documentation*. URL: <https://solidity.readthedocs.io/en/v0.4.24/abi-spec.html%7B%5C#%7Dfunction-selector-and-argument-encoding> (visited on 06/08/2019).
- [7] Y. N. SRIKANT’ ARVIND M. MURCHING Y. V. PRASAD. “INCREMENTAL RECURSIVE DESCENT”. In: (1990).
- [8] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. “A Survey of Attacks on Ethereum Smart Contracts (SoK) BT - Principles of Security and Trust”. In: *Principles of Security and Trust - 6th International Conference, {POST} 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, {ETAPS} 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings* 10204.July (2017), pp. 164–186.
- [9] Massimo Bartoletti and Livio Pompianu. “An Empirical analysis of smart contracts: Platforms, applications, and design patterns”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10323 LNCS (2017), pp. 494–509. ISSN: 16113349. DOI: 10.1007/978-3-319-70278-0_31. arXiv: arXiv:1703.06322v1.
- [10] Ira D Baxter. “Branch Coverage for Arbitrary Languages Made Easy”. In: i (), pp. 2–7.
- [11] *chalk/chalk: Terminal string styling done right*. URL: <https://github.com/chalk/chalk> (visited on 06/15/2019).

- [12] *Code Coverage Metrics — SeaLights*. URL: <https://www.sealights.io/test-metrics/code-coverage-metrics/> (visited on 06/03/2019).
- [13] *Code Coverage Tutorial: Branch, Statement, Decision, FSM*. URL: <https://www.guru99.com/code-coverage.html> (visited on 06/04/2019).
- [14] *Common Patterns Solidity 0.4.21 documentation*. URL: <https://solidity.readthedocs.io/en/v0.4.21/common-patterns.html%7B%5C%7Dstate-machine> (visited on 05/30/2019).
- [15] *Common Patterns Solidity 0.4.24 documentation*. URL: <https://solidity.readthedocs.io/en/v0.4.24/common-patterns.html> (visited on 05/30/2019).
- [16] *Compilers — Stanford Lagunita*. URL: <https://lagunita.stanford.edu/courses/Engineering/Compilers/Fall2014/about> (visited on 01/22/2019).
- [17] *ConsenSys/ethereum-developer-tools-list: A guide to available tools and platforms for developing on Ethereum*. URL: <https://github.com/ConsenSys/ethereum-developer-tools-list%7B%5C%7Dmonitoring> (visited on 06/13/2019).
- [18] *Contracts - Solidity 0.4.21 documentation*. URL: <https://solidity.readthedocs.io/en/v0.4.21/contracts.html%7B%5C%7Devents> (visited on 06/02/2019).
- [19] *Contribution Points — Visual Studio Code Extension API*. URL: <https://code.visualstudio.com/api/references/contribution-points> (visited on 06/08/2019).
- [20] *Creating a Legacy TextMate Grammar*. URL: <https://flight-manual.atom.io/hacking-atom/sections/creating-a-legacy-textmate-grammar/> (visited on 06/08/2019).
- [21] *Creating models and diagrams*. URL: <https://www.ibm.com/support/knowledgecenter/en/SS5JSH%7B%5C%7D9.5.0/com.ibm.xttools.modeler.doc/topics/t%7B%5C%7Dcreatemoddiags.html> (visited on 06/05/2019).
- [22] *Deconstructing a Solidity Contract - Part II: Creation vs. Runtime*. URL: <https://blog.zeppelin.solutions/deconstructing-a-solidity-contract-part-ii-creation-vs-runtime-6b9d60ecb44c> (visited on 06/08/2019).
- [23] *Deep dive into Ethereum logs - codeburst*. URL: <https://codeburst.io/deep-dive-into-ethereum-logs-a8d2047c7371> (visited on 05/30/2019).
- [24] *DJRHails/language-flint*. URL: <https://github.com/DJRHails/language-flint> (visited on 06/15/2019).
- [25] *etheratom*. URL: <https://atom.io/packages/etheratom> (visited on 06/11/2019).
- [26] *ethereum/solc-js: Javascript bindings for the Solidity compiler*. URL: <https://github.com/ethereum/solc-js%7B%5C%7Dreadme> (visited on 06/15/2019).
- [27] *ethorse-core/Betting.sol*. URL: <https://github.com/ethorse/ethorse-core/blob/master/contracts/Betting.sol> (visited on 05/30/2019).
- [28] *Exception testing - junit-team/junit4 Wiki*. URL: <https://github.com/junit-team/junit4/wiki/exception-testing> (visited on 06/02/2019).

- [29] *Extension Anatomy — Visual Studio Code Extension API*. URL: <https://code.visualstudio.com/api/get-started/extension-anatomy> (visited on 06/07/2019).
- [30] *File Explorer - Remix, Ethereum-IDE 1 documentation*. URL: https://remix.readthedocs.io/en/latest/file%7B%5C_%7Dexplorer.html%7B%5C_%7Dconnect-your-filesystem-to-remix (visited on 01/22/2019).
- [31] *flintlang/flint: The Flint Programming Language for Smart Contracts*. URL: <https://github.com/flintlang/flint> (visited on 05/31/2019).
- [32] *flint/language_guide.md*. URL: https://github.com/flintlang/flint/blob/master/docs/language%7B%5C_%7Dguide.md%7B%5C_%7Drange-types (visited on 06/09/2019).
- [33] *flintlang/vscode-flint*. URL: <https://github.com/flintlang/vscode-flint> (visited on 06/15/2019).
- [34] *Geth - ethereum/go-ethereum Wiki*. URL: <https://github.com/ethereum/go-ethereum/wiki/geth> (visited on 06/05/2019).
- [35] *go-ethereum/bloom9.go (Bloom Filter)*. URL: https://github.com/ethereum/go-ethereum/blob/213b8f9af41aee0f0888818bceb500374409733d/core/types/bloom9.go%7B%5C_%7DL88 (visited on 06/08/2019).
- [36] *Graphviz - Graph Visualization Software*. URL: <https://www.graphviz.org/> (visited on 06/06/2019).
- [37] K. Hammond and V. J. Rayward-Smith. “A survey on syntactic error recovery and repair”. In: *Computer Languages* 9.1 (1984), pp. 51–67. ISSN: 00960551. DOI: 10.1016/0096-0551(84)90012-2.
- [38] Fergus Henderson. “Software Engineering at Google”. In: (2017). arXiv: 1702.01715. URL: <http://arxiv.org/abs/1702.01715>.
- [39] *IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains*. URL: <https://www.jetbrains.com/idea/> (visited on 06/05/2019).
- [40] *IntelliJ IDEA Ultimate - Plugins — JetBrains*. URL: <https://plugins.jetbrains.com/idea> (visited on 06/09/2019).
- [41] *Istanbul, a JavaScript test coverage tool*. URL: <https://istanbul.js.org/> (visited on 06/12/2019).
- [42] *JavaScript API (web3) - ethereum/wiki Wiki*. URL: <https://github.com/ethereum/wiki/wiki/JavaScript-API> (visited on 06/15/2019).
- [43] *Jira Service Desk — IT Service Desk & Ticketing*. URL: <https://www.atlassian.com/software/jira/service-desk> (visited on 06/09/2019).
- [44] Juan Blanco. *solidity - Visual Studio Marketplace*. 2019. URL: <https://marketplace.visualstudio.com/items?itemName=JuanBlanco.solidity> (visited on 05/29/2019).
- [45] *Language Grammars - TextMate 1.x Manual*. URL: https://macromates.com/manual/en/language%7B%5C_%7Dgrammars (visited on 06/08/2019).

- [46] *Language Server Extension Guide — Visual Studio Code Extension API*. URL: <https://code.visualstudio.com/api/language-extensions/language-server-extension-guide> (visited on 01/22/2019).
- [47] *Learnable Programming*. URL: <http://worrydream.com/LearnableProgramming/> (visited on 01/24/2019).
- [48] *Linux Test Project - Coverage lcov*. URL: <http://ltp.sourceforge.net/coverage/lcov.php> (visited on 06/05/2019).
- [49] Robert C Martin. *The Code Cleaner*. 2011. ISBN: 9780137081073. URL: <http://scholar.google.com/scholar?hl=en%7B%5C%7DbtnG=Search%7B%5C%7Dq=intitle:The+Clean+Coder+a+code+of+conduct+for+professional+programmers%7B%5C%7D0>.
- [50] *Mining - ethereum/wiki Wiki*. URL: <https://github.com/ethereum/wiki/wiki/Mining> (visited on 06/08/2019).
- [51] *Node.js*. URL: <https://nodejs.org/en/> (visited on 06/05/2019).
- [52] K Nørmark. "Systematic unit testing in a read-eval-print loop". In: *Journal of Universal Computer Science* 16.2 (2010), pp. 296–314. URL: <http://www.scopus.com/inward/record.url?eid=2-s2.0-77950890748%7B%5C%7DpartnerID=40%7B%5C%7Dmd5=>.
- [53] *openzeppelin-solidity/Ownable.sol*. URL: <https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/ownership/Ownable.sol> (visited on 05/30/2019).
- [54] *openzeppelin-solidity/Pausable.sol*. URL: <https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/lifecycle/Pausable.sol> (visited on 05/30/2019).
- [55] *openzeppelin-solidity/TESTING.md*. URL: <https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/test/TESTING.md> (visited on 05/30/2019).
- [56] *Overview*. URL: <https://microsoft.github.io/language-server-protocol/overview> (visited on 01/22/2019).
- [57] *Overview - Smart Contract Weakness Classification and Test Cases*. URL: <https://smartcontractsecurity.github.io/SWC-registry/> (visited on 05/30/2019).
- [58] *PocketinnsContracts/DutchAuction.sol*. URL: <https://github.com/pocketinns/PocketinnsContracts/blob/master/DutchAuction.sol> (visited on 05/30/2019).
- [59] Simone Porru et al. "Blockchain-oriented software engineering: Challenges and new directions". In: *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017* (2017), pp. 169–171. DOI: 10.1109/ICSE-C.2017.142.
- [60] *Quick Start using the JavaScript VM*. URL: https://remix.readthedocs.io/en/latest/quickstart_javascript_vm.html. 'Date Accessed: 22/01/2019'.
- [61] *raineorshine/solgraph: Visualize Solidity control flow for smart contract security analysis*. URL: <https://github.com/raineorshine/solgraph> (visited on 05/29/2019).

- [62] *Remix - Ethereum IDE*. URL: <https://remix.ethereum.org> (visited on 05/29/2019).
- [63] *Running transactions*. URL: https://remix.readthedocs.io/en/latest/run_tab.html. 'Date Accessed: 22/01/2019'.
- [64] *sc-forks/solidity-coverage: Code coverage for Solidity smart-contracts*. URL: <https://github.com/sc-forks/solidity-coverage> (visited on 06/12/2019).
- [65] *Scope Naming - Sublime Text 3 Documentation*. URL: https://www.sublimetext.com/docs/3/scope%7B%5C_%7Dnaming.html (visited on 06/08/2019).
- [66] *Security Considerations - Solidity 0.5.9 documentation*. URL: https://solidity.readthedocs.io/en/v0.5.9/security-considerations.html%7B%5C_%7Dabstraction-and-false-positives (visited on 05/30/2019).
- [67] *SmartCheck*. URL: <https://tool.smartdec.net/> (visited on 06/11/2019).
- [68] *solidity-patterns/access_restriction.md*. URL: https://github.com/fravoll/solidity-patterns/blob/master/docs/access%7B%5C_%7Drestriction.md (visited on 05/30/2019).
- [69] *solidity-patterns/state_machine.md*. URL: https://github.com/fravoll/solidity-patterns/blob/master/docs/state%7B%5C_%7Dmachine.md (visited on 05/30/2019).
- [70] *sol-trace*. URL: <https://sol-trace.com/> (visited on 05/30/2019).
- [71] *Specification*. URL: https://microsoft.github.io/language-server-protocol/specification%7B%5C_%7DtextDocument%7B%5C_%7DpublishDiagnostics (visited on 01/22/2019).
- [72] *Stack Overflow Developer Survey 2018*. URL: https://insights.stackoverflow.com/survey/2018%7B%5C_%7Dtechnology (visited on 05/30/2019).
- [73] *SWC-105 - Overview*. URL: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-105%7B%5C_%7Dmultiowned-vulnerablesol (visited on 05/30/2019).
- [74] *Syntax Highlight Guide — Visual Studio Code Extension API*. URL: <https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide> (visited on 06/08/2019).
- [75] *TestCoverage*. URL: <https://martinfowler.com/bliki/TestCoverage.html> (visited on 06/03/2019).
- [76] *Testing for Exceptions — Unit Testing in C# — Treehouse*. URL: <https://teamtreehouse.com/library/testing-for-exceptions> (visited on 06/02/2019).
- [77] *The JavaScript Problem - HaskellWiki*. URL: https://wiki.haskell.org/The%7B%5C_%7DJavaScript%7B%5C_%7DProblem (visited on 05/30/2019).
- [78] *The State of Developer Ecosystem 2018 - Infographic — JetBrains*. URL: <https://www.jetbrains.com/research/devecosystem-2018/> (visited on 01/19/2019).
- [79] *Theia - Cloud and Desktop IDE*. URL: <https://www.theia-ide.org/> (visited on 06/09/2019).

- [80] *Tools supporting the LSP*. URL: <https://microsoft.github.io/language-server-protocol/implementors/tools/> (visited on 06/07/2019).
- [81] *Tracing: Introduction - ethereum/go-ethereum Wiki*. URL: <https://github.com/ethereum/go-ethereum/wiki/Tracing:-Introduction> (visited on 06/12/2019).
- [82] *Trello*. URL: <https://trello.com/en-GB> (visited on 06/15/2019).
- [83] *Truffle Suite — Documentation — Truffle — Interacting with Your Contracts*. URL: <https://truffleframework.com/docs/truffle/getting-started/interacting-with-your-contracts> (visited on 06/05/2019).
- [84] *Truffle Suite — Documentation — Truffle — Interacting with Your Contracts*. URL: <https://truffleframework.com/docs/truffle/getting-started/interacting-with-your-contracts> (visited on 06/01/2019).
- [85] *Truffle Suite — Documentation — Truffle — Networks and App Deployment*. URL: <https://www.trufflesuite.com/docs/truffle/advanced/networks-and-app-deployment> (visited on 06/12/2019).
- [86] *Truffle Suite — Documentation — Truffle — Overview*. URL: <https://www.trufflesuite.com/docs/truffle/overview> (visited on 01/18/2019).
- [87] *Truffle Suite — Documentation — Truffle — Package Management via EthPM*. URL: <https://www.trufflesuite.com/docs/truffle/getting-started/package-management-via-ethpm> (visited on 06/09/2019).
- [88] *Truffle Suite — Documentation — Truffle — Testing Your Contracts*. URL: <https://truffleframework.com/docs/truffle/testing/testing-your-contracts> (visited on 05/30/2019).
- [89] *Truffle Suite — Documentation — Truffle — Testing Your Contracts*. URL: <https://truffleframework.com/docs/truffle/testing/testing-your-contracts> (visited on 05/30/2019).
- [90] *trufflesuite/ganache: Personal blockchain for Ethereum development*. URL: <https://github.com/trufflesuite/ganache> (visited on 06/14/2019).
- [91] *Unit Testing*. URL: https://remix.readthedocs.io/en/latest/unittesting_tab.html. 'Date Acsessed: 22/01/2019'.
- [92] *Welcome to Python.org*. URL: <https://www.python.org/> (visited on 06/05/2019).
- [93] *What is Ethereum? - Ethereum Homestead 0.1 documentation*. URL: <http://www.ethdocs.org/en/latest/introduction/what-is-ethereum.html> (visited on 01/23/2019).
- [94] Gavin Wood and Vitalik Buterin. "Ethereum: A Secure Decentralised Generalised Transaction Ledger". In: *Ethereum Project Yellow Paper* (2014). ISSN: 02533820. DOI: 10.1017/CB09781107415324.004. arXiv: arXiv:1011.1669v3. URL: <http://www.cryptopapers.net/papers/ethereum-yellowpaper.pdf>.

-
- [95] *YAKINDU Solidity Tools — The free to use, open source YAKINDU Solidity Tools provide an integrated development environment for Ethereum / Solidity based smart contracts.* URL: <https://yakindu.github.io/solidity-ide/> (visited on 05/29/2019).

Appendix A

User Manual

User manual on using the tools as VSCode extensions or as command line programs. The source code for these tools will be available as part of the flint language repository. They have not yet been merged in.

A.1 VSCode Extensions

The language server and analysis tools will be released as part of a VSCode extension. This will simplify the usage of these tools.

A.2 Flint Language Server

A.2.1 VSCode Extension

To run the language server, the user should install the VSCode extension and load a Flint contract into the editor. This will automatically trigger the language extension to start running.

A.2.2 Command Line Interface


To run the language server from the command line. The user should use the following command:

```
flint-lsp <source code> <file name>
```

The command line tool outputs JSON.

A.3 Analysis Tools

To run the analysis tools, the user should install the VSCode extension and load a Flint contract into the editor. This will trigger the extension. To run the analysis, the



Contract Insights: Estimate Gas
Contract Insights: Analyse

Figure A.1: Commands added to VSCode command palette.

Flag	Feature	Output
g	Gas Estimation	JSON
c	Protection Analysis	JSON
t	Type State Visualisation	Dot Graph
f	Function Visualisation	Dot Graph

Table A.1: Flags For Command Line Analysis Tool

user should load the command palette for VSCode. On Mac, the command to open the palette is shift-command-P. The commands we have added to palette are shown in figure A.1. The Analyse command can be used to run the protection analysis, type state visualisation and function visualisation. The Estimate Gas Command can be used to run the gas estimate analysis.

A.3.0.1 Command Line Interface

The syntax for running the analysis tools from the command line is as follows:

```
flint-ca <flag> <source code> <file name>
```

Table A.1 list all the different flags the user can pass in. It also lists the type of output the tool produces.

A.4 Flint-Test

The user should look at the main body of the report to see how to write a Flint-Test file and to see what features it support. Section 6 discusses the testing framework. To run the test framework the user uses the following command:

```
flint-test [-c] <flint-test file>
```

The flag toggles coverage.

A.5 Flint-REPL (Interactive Console)

To launch the console, the user uses the following command:

```
flint-repl <Flint file>
```

Please refer to section 7 for more information on what the console can do.

Appendix B

Third Party Libraries

We use the following third party libraries to build the Flint developer ecosystem.

- web3.js, by Ethereum, to interface with the blockchain. [42]
- Flint, by Franklin Schrans, to parse and compile Flint contracts [31].
- Geth, by Ethereum, to launch a local blockchain [34].
- Chalk, found in npm, to print colourful terminal output in the test framework [11].
- Solc-js, by Ethereum, to compile Solidity contracts from JavaScript [26].

Appendix C

BNF's

C.1 Flint-Test DSL BNF

```
1 ; FLINT-TEST DSL GRAMMAR
2
3 ; TOP LEVEL
4 testFile = contractDeclaration contractBehaviourDeclaration
5
6 ; CONTRACTS
7 contractDeclaration = %s"contract" SP identifier SP "{" *(WSP
    variableDeclaration CRLF) "}";
8
9 ; VARIABLES
10 variableDeclaration = [*(modifier SP)] WSP (%s"var" / %s"let") SP
    identifier typeAnnotation WSP "=" WSP expression;
11
12 ; TYPES
13 typeAnnotation = ":" WSP type;
14
15 type = identifier
16       / basicType
17
18 basicType = %s"Bool"
19            / %s"Int"
20            / %s"String"
21            / %s"Address";
22
23 ; BEHAVIOUR
24 contractBehaviourDeclaration = identifier WSP "::<" WSP (any) WSP "{"
    *(WSP contractBehaviourMember CRLF) "}";
25
26 contractBehaviourMember = functionDeclaration
27
28 ; FUNCTIONS + INITIALIZER
29 functionSignatureDeclaration = functionHead SP identifier"()"
30 functionDeclaration         = functionSignatureDeclaration
    codeBlock;
31
32 functionHead = [*(modifier SP)] %s"func";
```



```

33
34 modifier = %s"public"
35
36 ; STATEMENTS
37 codeBlock = "{" [CRLF] *(WSP statement CRLF) WSP "}";
38 statement = expression
39
40 ; EXPRESSIONS
41 expression = identifier
42             / binaryExpression
43             / functionCall
44             / literal
45             / variableDeclaration
46
47 binaryOp = "="
48           / ".";
49
50 binaryExpression = expression WSP binaryOp WSP expression;
51
52 ; FUNCTION CALLS
53 functionCall = identifier "(" [expression *( "," WSP expression )]
54               ")" ;
55
56 ; LITERALS
57 identifier = ( ALPHA / "_" ) *( ALPHA / DIGIT / "$" / "_" );
58 literal = numericLiteral
59          / stringLiteral
60          / booleanLiteral
61          / addressLiteral;
62
63 number = 1 * DIGIT;
64 numericLiteral = decimalLiteral;
65 decimalLiteral = number
66
67 addressLiteral = %s"0x" 40HEXDIG;
68
69 booleanLiteral = %s"true" / %s"false";
70 stringLiteral = "" identifier "";

```

Listing C.1: Flint-Test BNF [31]

C.2 Flint-REPL BNF

```

1 ; VARIABLES
2 variableDeclaration = (%s"var" / %s"let") SP identifier
3                     typeAnnotation [WSP "=" WSP expression];
4
5 ; TYPES
6 typeAnnotation = ":" WSP type;
7
8 basicType = %s"Bool"
9            / %s"Int"
10           / %s"String"
11           / %s"Address";

```

```
12 type = identifier
13     / basicType
14
15
16 ; EXPRESSIONS
17 expression = identifier
18             / binaryExpression
19             / functionCall
20             / literal
21             / variableDeclaration
22
23 binaryOp = "+" / "-" / "*" / "/" /
24           / "="
25           / "||" / "&&"
26           / ".";
27
28 binaryExpression = expression WSP binaryOp WSP expression;
29
30 ; FUNCTION CALLS
31 functionCall = identifier "(" [expression] *( "," WSP expression )
32              "(");
33
34 ; LITERALS
35 literal      = numericLiteral
36              / stringLiteral
37              / booleanLiteral
38              / addressLiteral;
39
40 number        = 1*DIGIT;
41 numericLiteral = decimalLiteral;
42 decimalLiteral = number
43               / number "." number;
44
45 addressLiteral = %s"0x" 40HEXDIG;
46
47 booleanLiteral = %s"true" / %s"false";
48 stringLiteral  = "" identifier "";
```

Listing C.2: REPL BNF [31]

Appendix D

Flint Contracts Used In Main Body Of Report

```
1 contract Counter {
2     var value : Int = 0
3     event Message(msg: String)
4 }
5
6 Counter :: (any) {
7
8     public init() {}
9
10    public func getValue() -> Int {
11        return value
12    }
13
14    mutating public func increment() {
15        self.value += 1
16    }
17
18    mutating public func printMsg(cond: Bool) {
19        var s : String = ""
20        if (cond) {
21            s = "TRUE"
22        } else {
23            s = "FALSE"
24        }
25
26        emit Message(msg: s)
27    }
28 }
```

Listing D.1: Flint Contract Used To Generate Coverage Report

```
1 contract TestCounter {
2     let filePath : String = "[redacted]"
3     let contractName: String = "Counter"
4     let TestSuiteName : String = "CounterTests"
5 }
6
```

```
7 TestCounter :: (any) {  
8  
9     public func test_coverage() {  
10         let c : Counter = Counter()  
11         c.increment()  
12         let x : Int = c.getValue()  
13         c.printMsg(true)  
14     }  
15  
16 }
```

Listing D.2: Flint Test File Used To Generate Coverage Report

Appendix E

Contracts Used In Evaluation

E.1 Flint

```
1 contract TestCounter {
2   let filePath : String = "[redacted]"
3   let contractName: String = "Counter"
4   let TestSuiteName : String = "Counter Tests"
5 }
6
7 TestCounter :: (any) {
8
9   public func test_event_fired() {
10    let c : Counter = Counter()
11    c.increment()
12    assertEventFired(c.counterVal, val: 1);
13  }
14
15 }
```

Listing E.1: Flint Test Which Tests Events

```
1 contract Counter {
2   var value: Int = 0
3   event counterVal(val: Int)
4 }
5
6 Counter :: (any) {
7   public init() {}
8
9   mutating public func increment() {
10    self.value += 1
11    emit counterVal(val: self.value)
12  }
13 }
```

Listing E.2: Flint Contract Which Uses Events

```
1 contract TestMultiOwned {
2   let filePath : String = "[redacted]"
3   let contractName : String = "MultiOwnable"
```

```

4  let TestSuiteName : String = "MultiOwnable Tests"
5  }
6
7  TestMultiOwned :: (any) {
8      public func test_only_owners_can_add() {
9          let owner : Address = newAddress()
10         let not_owner : Address = newAddress()
11
12         let mo : MultiOwnable = MultiOwnable(owner)
13
14         setAddr(not_owner)
15         assertCallerUnsat("addOwner", not_owner)
16         unsetAddr()
17     }
18 }

```

Listing E.3: Flint Test Used To Verify Caller Protections

```

1  contract MultiOwnable {
2      var root : Address
3      var owners: [Address : Address] = [:]
4      let zero : Address = 0x0000000000000000000000000000000000000000000000000000000000000000
5  }
6
7  MultiOwnable :: (any) {
8      public init(initial_owner: Address) {
9          self.root = initial_owner
10         self.owners[root] = root
11     }
12 }
13
14 MultiOwnable :: caller <- (checkIfOwner) {
15
16     func checkIfOwner(address : Address) -> Bool {
17         let parent : Address = owners[address]
18         return parent != zero
19     }
20
21     mutating public func addOwner(new_owner: Address) {
22         self.owners[new_owner] = caller
23     }
24
25     mutating public func deleteOwner(owner: Address) {
26         self.owners[owner] = zero
27     }
28 }
29 }

```

Listing E.4: MultiOwnable Contract Implemented in Flint

```

1  contract Emergency (Running, Paused) {
2      var owner: Address
3  }
4
5
6  Emergency @(Paused) :: (any) {

```

```

7   public init(initial_owner: Address) {
8       self.owner = initial_owner
9       become Paused
10  }
11 }
12
13 Emergency @(Paused) :: (owner) {
14
15     mutating public func unpause() {
16         become Running
17     }
18 }
19
20 Emergency @(Running) :: (owner) {
21
22     mutating public func pause() {
23         become Paused
24     }
25 }
26 }
27
28 Emergency @(Running) :: (any) {
29
30     public func run() {
31         // useful function for users
32     }
33 }

```

Listing E.5: Flint Contract Which Implements the Emergency Pattern

```

1  contract TestEmergency {
2      let filePath : String = "[redacted]"
3      let contractName: String = "Emergency"
4      let TestSuiteName : String = "Emergency Tests"
5  }
6
7  TestEmergency :: (any) {
8
9      public func test_cannot_run_in_paused_state() {
10         let owner : Address = newAddress()
11         let e : Emergency = Emergency(owner)
12
13         assertCantCallInThisState("run")
14     }
15 }
16 }

```

Listing E.6: Flint Test File Which Tests State Protections

```

1  contract TestCounter {
2      let filePath : String = "[redacted]"
3      let contractName: String = "Counter"
4      let TestSuiteName : String = "CounterTests"
5  }
6
7  TestCounter :: (any) {

```

```

8
9   public func testExceptionThrown() {
10       let c : Counter = Counter()
11
12       assertWillThrow(c.willThrow)
13   }
14 }

```

Listing E.7: Flint Test File Which Tests For Exceptions

```

1 contract Counter {
2 }
3
4 Counter :: (any) {
5
6     public init() {}
7
8     public func willThrow() {
9         fatalError()
10    }
11
12 }

```

Listing E.8: Flint Contract Which Throws an Exception

E.2 Solidity

```

1 pragma solidity >=0.4.25 <0.6.0;
2
3 contract Emergency {
4     enum States {
5         Running,
6         Paused
7     }
8
9     address public owner;
10    States public state ;
11
12    constructor(address _owner) public {
13        owner = _owner;
14        state = States.Running;
15    }
16
17
18    modifier atStage(States _state) {
19        require(state == _state);
20        _;
21    }
22
23    modifier onlOwner() {
24        require(msg.sender == owner);
25        _;
26    }
27
28    function run() public atStage(States.Running) {

```



```

29     // function will do something useful
30 }
31
32 }

```

Listing E.9: Solidity Contract Implementing the Emergency Pattern

```

1  pragma solidity >=0.4.25 <0.6.0;
2
3  contract MultiOwnable {
4      address public root;
5      mapping (address => address) public owners; // owner => parent of
        owner
6
7      /**
8       * @dev The Ownable constructor sets the original 'owner' of the
9       * contract to the sender
10      * account.
11      */
12     constructor(address _root) public {
13         root = _root;
14         owners[root] = root;
15     }
16
17     /**
18      * @dev Throws if called by any account other than the owner.
19      */
20     modifier onlyOwner() {
21         require(owners[msg.sender] != address(0));
22         _;
23     }
24
25     /**
26      * @dev Adding new owners
27      * Note that the "onlyOwner" modifier is missing here.
28      */
29     function newOwner(address _owner) external onlyOwner returns
        (bool) {
30         owners[_owner] = msg.sender;
31         return true;
32     }
33
34     /**
35      * @dev Deleting owners
36      */
37     function deleteOwner(address _owner) onlyOwner external returns
        (bool) {
38         owners[_owner] = address(0);
39         return true;
40     }
41 }

```

Listing E.10: Solidity Contract Implementing the MultiOwnable Pattern [73]

```

1  const truffleAssert = require("truffle-assertions");

```

```
2 const Emergency = artifacts.require("Emergency");
3
4 contract('Emergency', ([owner]) => {
5   let e;
6
7   beforeEach('setup contract for each test', async function () {
8     e = await Emergency.new(owner)
9   })
10
11   it('should revert when trying to call run in paused state', async
12     () => {
13       await truffleAssert.reverts(e.run());
14     });
15 });
```

Listing E.11: Truffle Test Checking State Protections

```
1 const MultiOwnable = artifacts.require("MultiOwnable");
2 const truffleAssert = require("truffle-assertions");
3
4 contract('MultiOwnable', ([owner, not_owner]) => {
5   let m0;
6
7   beforeEach('setup contract for each test', async function () {
8     m0 = await MultiOwnable.new(owner)
9   })
10
11   it('should revert when trying to add owners', async () => {
12     await truffleAssert.reverts(m0.newOwner(not_owner, {from:
13       not_owner}));
14   });
15 });
```

Listing E.12: Truffle Test Checking Caller Protections

```
1 const Counter = artifacts.require("Counter");
2 const truffleAssert = require("truffle-assertions");
3
4 contract('Counter', ([owner]) => {
5   let c;
6
7   beforeEach('setup contract for each test', async function () {
8     c = await Counter.new();
9   })
10
11   it('emit counter state when incremented', async () => {
12     let tx = await c.increment();
13     truffleAssert.eventEmitted(tx, 'emitVal', (ev) => {
14       return ev.val == 1;
15     });
16   });
17
18 });
```

Listing E.13: Truffle Tests Which Tests Events

```
1 pragma solidity >=0.4.25 <0.6.0;
2
3 contract Counter {
4
5     uint public value;
6
7     event emitVal(
8         uint val
9     );
10
11     constructor() public {
12         value = 0;
13     }
14
15     function willThrow() public {
16         revert();
17     }
18
19     function increment() public {
20         value = value + 1;
21         emit emitVal(value);
22     }
23
24 }
```

Listing E.14: Solidity Contract Which Emits an Event And Throws An Exception

```
1 pragma solidity >=0.4.25 <0.6.0;
2 contract Shadow {
3     uint n = 2;
4
5     constructor() public {}
6
7     function test1(uint n) pure public returns (uint) {
8         return n; // shadowing state variable to be
9     }
10 }
```

Listing E.15: Solidity Contract Which Has The Shadow State Variable Bug

```
1 const Shadow = artifacts.require("Shadow");
2 const truffleAssert = require("truffle-assertions");
3 Shadow.numberFormat = "BigNumber";
4
5
6 contract('Shadow', ([owner]) => {
7     let s;
8
9     beforeEach('setup contract for each test', async function () {
10         s = await Shadow.new();
11     })
12
13     it('emit counter state when incremented', async () => {
14         let bigNumVal = await s.test1(1);
15         let value = bigNumVal.toNumber();
16         assert.equal(2, value, "should be equal to 2");
17     });
18 })
```

```
17     });  
18  
19 });
```

Listing E.16: Truffle Test To Check For Shadowing State Variable Bug