Electronics and Computer Science
Faculty of Engineering and Physical Sciences
University of Southampton

Gherman Nicolisin

April 20, 2024

Folidity - Formally Verifiable Smart Contract Language

Project supervisor: Prof. Vladimiro Sassone
Second examiner: Dr. Indu Bodala

A project report submitted for the award of
BSc Computer Science

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
ELECTRONICS AND COMPUTER SCIENCE

A project report submitted for the award of BSc Computer Science

By Gherman Nicolisin

This paper addresses the long-lasting problem involving the exploits of Smart Contract vulnerabilities. There are tools, such as in the formal verification field and alternative Smart Contract languages, that attempt to address these issues. However, neither approach has managed to combine the static formal verification and the generation of runtime assertions. Furthermore, this work believes that implicit hidden state transition is the root cause of security compromises. In light of the above, we introduce Folidity, a formally verifiable Smart Contract language with a unique approach to reasoning about the modelling and development of Smart Contract systems. Folidity features explicit state transition checks, a model-first approach, and built-in formal verification compilation stage.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

- **Solang - https://github.com/hyperledger/solang**
- **Open Source libraries listed in Appendix C**

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

Acknowledgments

I want to thank my supervisor Prof. Vladimiro Sassone for the guidance throughout the project that enabled me to push myself. I also would like to thank Stefano De Angelis, who partook in proofreading of my work. Finally, I dedicate this work to my parents and siblings, who have been a massive support for me during the last few years of my degree.

Contents

1. Introduction

The concept of "smart contract" (SC) was first coined by Nick Szabo as a computerised transaction protocol [1]. He later defined smart contracts as observable, verifiable, privity-applicable, and enforceable programs [2]. In other words, smart contracts were envisioned to inherit the natural properties of traditional "paper-based" contracts.

In 2014 SCs were technically formalised at the protocol level by Dr. Gavin Wood as an arbitrary program written in some programming language (Solidity) and executed in the blockchain's virtual machine of Ethereum [3].

Ethereum Virtual Machine (EVM) iterated over the idea of Bitcoin Scripting [4], allowing developers to deploy general-purpose, Turing-Complete programs that can have their own storage, hence the state, written in Solidity [5]. This enabled sophisticated applications that grew beyond the simple fund transfers among users.

Overall, SC can be summarised as an *immutable*, *permissionless*, *deterministic* computer program that is executed as part of state transition in the blockchain system [6], [3].

After a relatively short time, SCs have come a long way and allowed users to access different online services, also known as Decentralised Applications (DApps), in a completely trustless and decentralised way. The applications have spanned financial, health, construction [7], and other sectors.

2. Security and Safety of Smart Contracts

2.1. Overview

With the increased adoption of DApps and the total value locked in them, numerous attacks have focused on extracting funds from SCs. Due to SCs' permissionless nature, the most common attack vector exploits mistakes in the SC's source code. Specifically, the attacker can not tamper with the protocol code due to consensus mechanisms. Instead, they can interact with the publicly accessible parameters and interfaces to force the SC into an unexpected state, essentially gaining partial control of it.

A notorious example is the DAO hack, when hackers exploited unprotected re-entrance calls to withdraw **$50 million worth of ETH**. This event forced the community to hard-fork the protocol to revert the transaction, provoking a debate on the soundness of the action [8].

Another less-known example is the "King of the Ether" attack, caused by the unchecked low-level Solidity `send` call to transfer funds to a contract-based wallet [9]. The "King of the Ether Throne" contract could not recognise the failed transaction on the wallet side. Instead, the contract proceeded with the operation, incorrectly mutating its internal state.

Other issues involve the *safety* and *liveness* of SCs. The term *safety* describes *functional safety* and *type safety*. *Functional safety* refers to the guarantees that the system behaves according to the specification irrespective of the input data [10], whereas *type safety* refers to the guarantees that the language provides a sound type system [11]. The two are often used interchangeably with the *security* of code as compromising the former affects the latter. When discussing *liveness*, we describe the business logic of a DApp, particularly whether it transitions into the expected new state [12]. This is particularly important for executing mission-critical software in a distributed context.

Safety and *liveness* can be compromised due to the programmer's mistakes in the source code that can result in the SC entering the terminal in an unexpected state preventing users from interacting with it [13].

2.2. Vulnerability classification

There has been an effort in both academia and industry to classify common vulnerabilities and exploits in SCs in blockchain systems [14]–[16]. Some of the work has been recycled by bug bounty platforms, growing the community of auditors and encouraging peer-review of SCs through the websites such as *Code4rena*[1], *Solodit*[2], and many others.

Analysing the abovementioned work, SC vulnerabilities can be categorised into the six general groups outlined in Table 2.1. The six categories have been defined based on an analysis of the most common vulnerabilities and how they affect SC execution. Each category represents the general scope for a specific set of vulnerabilities that should be addressed in the SC development.

2.3. Setting the scene

Even with the raised awareness for the security and safety of SCs, recent reports from *Code4rena* still show *SCV3*, *SCV4* and *SCV5* commonly present in the recent audit reports [17], [13], [18].

In particular, in [18], a relatively simple calculation mistake prevented other SC users from withdrawing their funds.

It can be seen that SC Vulnerabilities illustrated in Table 2.1 are still evident in modern SCs, resulting in opening them up to exploits of different severity levels. Looking at the mentioned reports, there is little consensus about the weight of each vulnerability. Therefore, we can not classify any particular vulnerability as more severe than the other as it solely depends on the context in the code it is present. Furthermore, it has been realised that additional tooling or alternative SCLs need to be discovered to minimise the exposure of SC code to the earlier-mentioned vulnerabilities.

[1] https://code4rena.com
[2] https://solodit.xyz

Code	Title	Summary
SCV1	Timestamp maniupulation	Timestamp used in control-flow, randomness and storage, can open an exploit due to an ability for validator to manipulate the timestamp
SCV2	Pseudo-randomness	Using block number, block hash, block timestamp are not truly randomly generated parameters, and can be manipulated by the adversary validator
SCV3	Invalidly-coded states	When coding business logic, control-flow checks can be incorrectly coded resulting the SC entering into invalid state
SCV4	Access Control exploits	This is a more broad categorisation of vulnerabilities. It occurs when an adversary calls a restricted function. This is specifically present in *upgradeability* and *deleteability* of SCs
SCV5	Arithmetic operations	SCs are suspected to the same arithmetic bugs as classic programs. Therefore, unchecked operations can result in underflow/overflow or deletion by zero
SCV6	Unchecked external calls	Unchecked re-entrant, forward, delegate calls can result in the contract entering into unexpected state

Table 2.1: Classification of SC vulnerabilities

3. Related Work

3.1. Overview

Different solutions have been presented to mitigate the consistency in the presence of vulnerabilities and programmer mistakes. We can generally categorise them into two groups: safe SCLs, which allow users to write safe and secure code, particularly described in Section 3.3, and formal verification tools used alongside traditional SCLs presented in Section 3.2.

This chapter reviews both categories of tools, allowing us to evaluate their effectiveness in correlation to usability, aiming to provide a concise framework to analyse the SC tools dedicated to producing error-proof DApps.

3.2. Formal Verification Tools

Formal verification describes the assessment of the correctness of a system concerning a formal specification [19]. The specification is usually described in terms of verifiable models using mathematical proofs. There are multiple ways to verify a program formally focused on specific parts. *Model checking* utilises propositional logic to verify the mathematical abstractions of the system [20]. *Theorem proving* involves verifying relations between the model and the statements about the system [21]. Finally, *symbolic execution* focuses on executing the program using symbolic values instead of concrete values [19].

KEVM[3] is a tool that provides executable semantics of EVM using $\mathbb{K}$ framework. It uses reachability logic to reason symbolically about the system [22]. KEVM is a powerful tool that operates at the EVM bytecode level. Specifically, SC developers are required to write a specification in a separate file that is checked against the SC's compiled EVM bytecode. Whilst this provides more fine-grained assurance of the safety and correctness, it requires specialised knowledge of the $\mathbb{K}$ framework and EVM semantics, hence significantly increasing the development time.

The other interesting tool is Dafny[4]. Dafny is a general-purpose tool that checks inputs in any language using Hoare-logic and high-level annotations. Although Dafny offers compilation to some system languages, Solidity is not yet a supported target. Notably, work in the field suggests that Dafny can be an effective and

[3]https://jellopaper.org/index.html

easy-to-use tool for producing a formal specification [23]. The syntax resembles a traditional imperative style and is substantially easier to learn and understand than KEVM.

Some tools can be used alongside Solidity code, such as Scribble[5]. Scribble enables developers to provide formal specifications of functions inside docstrings that seamlessly integrate with existing Solidity code. It offers VS Code extensions and is actively maintained by Consensys[6]. The trade-off is the limited expressiveness compared with KEVM and Dafny.

Finally, experiments have been conducted to verify SC without any formal annotations. In particular, VeriSmart focuses explicitly on ensuring arithmetic safety and preciseness in SCs [24]. However, VeriSmart fails to detect other types of errors, although an effort has been made to apply the verifier to more areas of SC [25].

Formal verification is a multi-disciplinary field offering multiple ways of reason about the systems. One of the actively researched topics is bounded model verification [26]. Developers are required to reason about the programs as finite state machines (FSM)[27]. This reasoning approach is more apparent in SC development since the state transition is at the core of blockchain execution. Bounded model checking has been achieved through only a few experimental projects, such as Solidifier [28] and Microsoft [25]. Both projects attempt to translate Solidity code to an intermediate modelling language, Boogie [29]. Boogie then leverages SMT solvers to find any assertion violations.

Overall, we can see that formal verification tools provide a robust way of ensuring the safety of SCs. While significant effort has been made in the field, it is evident that formal verification tools in SC development attempt to compensate for Solidity's implicit state transitions and lack of *implicit* safety.

3.3. Safe Smart Contract Languages

Multiple attempts have been made to address a flawed programming model of Solidity [30]. Alternative SCLs aim to provide built-in safety features in a type system, modelling, and function declaration to minimise the need for external tooling.

Some languages strive for simplicity, such as Vyper[7]. By stripping off some low-level features, Vyper minimises the developer's chances of misusing the dangerous operations. It also provides overflow checking, signed integers, and other safe arithmetic operations. However, Vyper is still immature, and the recent bug in the compiler caused a massive re-entrance exploit in the *curve.fi* AMM protocol [31].

[4]https://dafny.org/latest/DafnyRef/DafnyRef
[5]https://docs.scribble.codes
[6]https://consensys.io/diligence/scribble

Furthermore, Vyper still suffers from the same implicit state transition problem as Solidity.

Flint is an experimental language with protected calls and asset types [32]. Protected calls introduce a role-based access system where the SC developer can specify the permitted caller to a message function. Another unique feature is array-bounded loops that partially address the halting problem. Flint also addresses a state-transition problem by allowing developers to specify all possible states in the contract. The message functions need to specify the state transition, which occurs explicitly. The language provides a significant improvement in a modelling approach. However, it still lacks the modelling SC input data in terms of constraints and invariants, and explicit state transition is still an optional feature that the developer can miss in using.

Another promising SCL reasons about SC development through dependent and polymorphic types [33]. It extends Idris[8] and makes the developer model the SC as part of a state transition function by adopting a functional programming style. Dependent types provide more fine-grained control over the input and output data that flow through the SC functions. In particular, similar to Haskell, the language offers *side-effects* functionality that resembles *IO* monads in Haskell. The downside of the approach is that the syntax has become too cumbersome for other developers to learn. Thus, it has been stated that the language does not strive for simplicity and sacrifices it for safety.

3.4. Problem Statement

We can identify the positive trend in providing safety for SCs. Modern formal verification methods offer support to SC developers in ensuring that their code satisfies the requirements of the system, while proposed SCL solutions offer runtime safety, minimising the need for the former.

However, there has been no effort to combine the two approaches into a single development process. Formal verification tools focus on the validation of functional correctness and model consistency of a program at the compile time, whereas SCLs focus on data validation at the runtime. Recent work suggests that the improved optimisation of SMT solvers allows us to turn the formal model specification into the runtime assertions [34]. Furthermore, no effort has been made to minimise false negatives in SC formal modelling, even though the methods have been developed for traditional systems, such as Event-B [35].

[7]https://docs.vyperlang.org/en/latest/index.html
[8]https://www.idris-lang.org

4. Proposed Solution

4.1. Outline

In light of the above, we believe there is a need for a solution that combines two approaches to allow SC developers to reason about their program in terms of FSM models that can be verified at the compile time for functional correctness and model consistency, and enable an automatic generation of invariants and constraints to validate the data at runtime.

We propose *Folidity*, a safe smart contract language. Folidity offers a model-first approach to development while featuring a functional-friendly programming style. The language intends to offer a safe and secure-by-design approach to the programming, ensuring the developer is aware of any state transitions during execution.

The list of feature requirements has been comprised based on the vulnerabilities described in Table 2.1.

1. **Provide abstraction over timestamp** in response to *SCV1*. We are interested in the limited use of timestamps in SCs in favour of the block number or another safe primitive.
2. **Provide a safe interface for randomness** in response to *SCV2*. Folidity should also provide a source of randomness through a standardised interface.
3. **Enable model-first approach in development** in response to *SCV3*. Developers should reason about the storage in terms of models and how they are updated by events. This approach is inspired by the Event-B [35] work, which can also be applied to SC development.
4. **Explicit state checks at runtime** in response to *SCV3* and *SCV6*. Similar to *Requirement 3*, SC developers should be aware of any state transitions that update the state of the model. State transitions must happen explicitly and be validated at the runtime to guarantee *liveness*.
5. **Static typing** in response to *SCV3* and *SCV5*.
6. **Polymorphic-dependent types** in response to *SCV3*. Polymorphic-dependent types should be part of a runtime assertion check during state transition, and model mutation[9].

7. **Role-based access** in response to *SCV4*. All message functions that mutate the model should be annotated with the role-access header specifying which set of accounts is allowed to call it.
8. **Checked arithmetic operations** in response to *SCV5*. All arithmetic operations should be checked by default, and the developer is responsible for explicitly specifying the behaviour during over/underflow, similar to Rust.
9. **Enforced checked recursion or bounded loops** in response to *SCV3*.

Infinite loops should not be permitted, and any loops should generally be discouraged in favour of recursion. The recursion base case should be specified explicitly with appropriate invariants. Bounded loops may be used but should be limited to list or mapping iterations.

As part of the language design, the SC building workflow is illustrated in Figure 1.

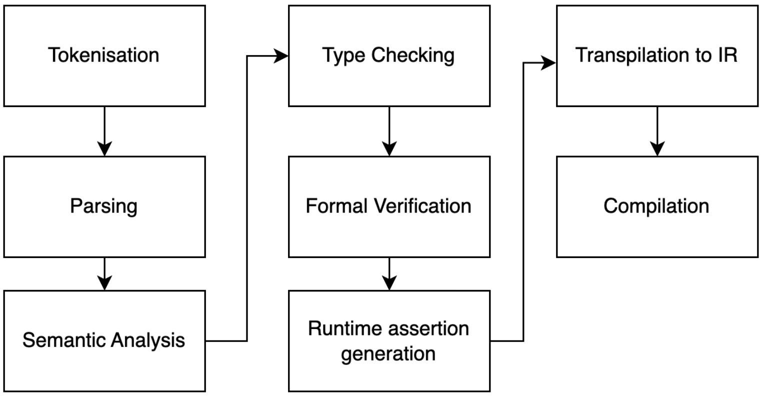


Figure 4.1: Build workflow

Formal verification is a novel addition to the workflow. After verifying the model consistency, invariants, and constraints, the program is considered safe for generating runtime assertions.

Another core feature is a pure computation context of the SC in Folidity. As illustrated in Figure 4.2, state mutations to the contract storage and the global state (e.g. account balances) happen independently. Folidity proposes a new execution model when a portion of a global state is *embedded* into the local state of the SC as shown in Figure 4.3. *Global state* refers to the overall state of the blockchain system (e.g. account balances), whereas *local state* describes the storage of an individual SC.

[9] *Model mutation* and *state transition* refer to the same process. They are used interchangeably

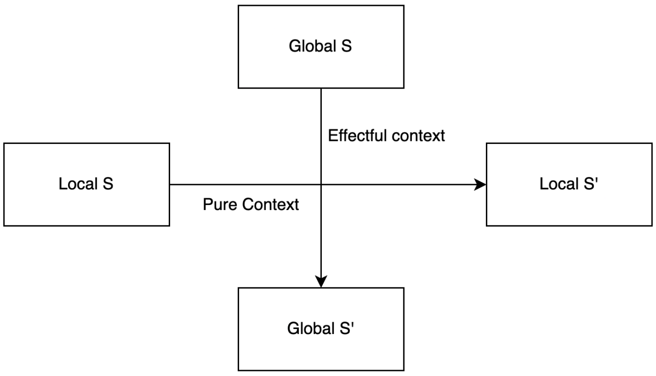


Figure 4.2: Traditional execution context

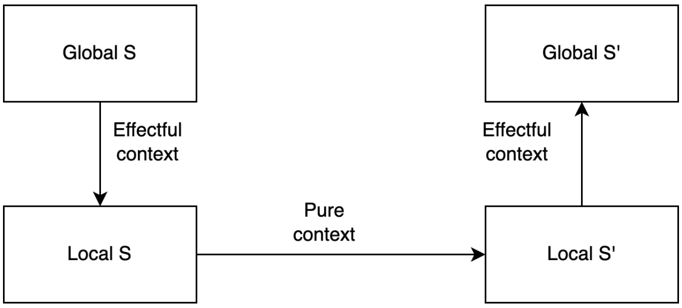


Figure 4.3: Transformed execution context

4.2. Language design

Folidity features a rich grammar that abstracts away from low-level operations while providing a high level of readability and expressivity. Certain considerations have been taken into account to reflect the desired features described in Section 4.1.

Folidity is described using LR(1)[10] grammar as outlined in Appendix B. One of the advantages of using LR(1) grammar is its expressiveness and clarity, which allow it to describe sophisticated data structures. It additionally enables easier implementation of the error-recovery [36] for reporting purposes, which lies at the core of the compiler.

4.2.1. Primitives, Expressions and Statements

Starting from primitives, Folidity provides numerous data types allowing encoding data for the domain of use cases in dApps:

- `int`, `uint`, `float` - signed, unsigned integers and floating-point numbers
- `()` - unit type, similar to rust this means no data.
- `string` - string literals, can be provided as `s"Hello World"`

[10]https://en.wikipedia.org/wiki/LR_parser

- `hex` - hexadecimal string literals, provided as `h"AB"`
- `address` - account number literal, provided as a"<address>"
- list<a>, set<a> - lists of elements of type `a`, `set` describes a list of unique elements.
- mapping<a -> b> - a mapping from type `a` to type `b` using the relation `->`
  - `->` : total relation
  - `-/>` : partial relation, can be combined with injective and surjective notations.
  - `>->` : (partial) injective relation
  - `->>` : (partial) surjective relation
  - `>->>` : bijective relation
- `char` - character, provided as `'a'`
- `bool` - boolean literals `true` or `false`

By describing the type of relations in mappings, we can combine the Event-B approach of proof obligation with symbolic execution to provide strong formal guarantees of member inclusion and member relations.

Specifically, we can define some axiom where we can have a mapping of partial injective relation between addresses (`address`) and asset ids (`uint`) assets: mapping<address >-/> int>:

$$\text{Assets: Address} \rightharpoonup \text{Int}$$

Then, for some statement S: assets = assets :> add(<a>, <b>), we can treat as a hypothesis. The compiler can then assert:

$$S, (a', a \in \text{Address}) \vdash \text{Assets}(a) \neq \text{Assets}(a')$$

Looking at the expressions, Folidity provides a standard set of operators to describe mathematical and boolean expressions (e.g. `+`, `/`, `||`, etc.) with few additions.

- `in` - inclusion operator, return `true` if for `a in A`, the $a \in A$ is true, if used in boolean context. Otherwise, it extracts an element from `A` and assigns it to `a` when used in iterators.
- `:>` - pipeline operator, heavily inspired by F# `|>` operator[11]. It allows piping the result of one function into the other one. This enables easy integration of a functional style of programming and handling of side effects of the mathematical operations, such as overflow or division by zero, hence addressing *SCV5* and *Requirement 8*.

```
let result: int = a + 1_000_000_000 :> handle((_) -> return 0);
```

[11] https://learn.microsoft.com/en-gb/dotnet/fsharp/language-reference/functions/#pipelines

Statements have close resemblances to Rust syntax and are defined in Appendix B.

```
let <var_ident>: <optional_type> = <expr>;
```

The type can be derived at the compile time from the expression.

It is worth looking at the unique additions such as struct instantiation and state transition.

Any structure-like type can instantiated using the
`<ident> : { <args>, ..<object> }` syntax, where

- `<ident>` - Identifier of the declaration.
- `<args>` - list of arguments to assign fields
- `<object>` - Object to fill the rest of the fields from if not all arguments are provided.

This expression can be combined with the state transition statement to execute the explicit change in the internal state of the SC.

```
move <state_ident> : { <args>, ..<object> };
```

4.2.2. Declarations

A typical program in Folidity consists of data structures that describe models, states, and functions that can interact with each other. Models are one of the core structures that provide the model consistency guarantee in Folidity. States can encapsulate different or the same models and describe explicit state transition or state mutations as part of program execution, and functions are the driving points in program execution. Functions declare and describe the state transitions.

Models resemble regular `struct` structures in "C-like" languages with few differences.

They describe some representation of the storage layout that is encapsulated within explicit states.

```
model MyModel {
  a: int,
  b: string,
} st [
  a > 10,
  b == s"Hello World"
]
```

Listing 4.1: Simple model with constraints

Folidity enables developers to further constrain the data that the model can accept by specifying model bounds in `st`[12] blocks. This syntax can also be used in state and function declarations as illustrated later. To support context transformation, any global state variables (e.g. block number, current caller) are injected into a model as fields and can be accessed in `st` blocks and expressions in functions. Furthermore, Folidity borrows the idea of model refinements from Event-B by allowing a model to inherit another model's fields and refine its constraints as shown in Listing 2.

```
model ParentModel {
  a: int,
} st [
  a > 10,
]

model MyModel: ParentModel {} st [
  a > 100
]
```

Listing 4.2: Model refinement

States facilitate the tracked mutation of the storage. They can encapsulate models, have raw fields in the body, or not contain any data at all. They are essentially *the* data structures that get encoded and stored as raw bytes in the contract storage.

```
model ParentModel {
  a: int,
} st [
  a > 10,
]

state StateA(ParentModel) st [
  a < 100
]

state StateB {
  b: uint
} st [
  b < 10
]

state NoState
```

Listing 4.3: States

The idea behind model encapsulation is to enable distinct states to represent identical models with their distinct constraints. Additionally, states' bounds can be further restricted by specifying the incoming state, which is the only state from which we can transition to the specified state.

[12]States for "such that"

```
state StateA from (StateB s) st [
  s.a > 10
]
```

Listing 4.4: State transition bounds

As mentioned earlier, functions facilitate the model mutation of the Folidiy SC. Functions provide a controlled interface for the user and other contracts to interact with the state of the application. Therefore, it is important to enable developers to control the execution flow of the incoming data and provide them with fine-grained control over output data and storage mutation.

Let's look at the signature of a typical function in Folidity;

```
@init
@(any)
fn (out: int) my_func(input: hex)
  where (InitialState s1) -> (FinalState s2)
  st [
    input != h"ABC",
    out > 10,
    out < 100
    s1.a == s2.a
  ] { <statements> }
```

Listing 4.5: Function signature

Starting from the top: `@init` is an optional attribute that indicates the function is used for instantiation of the contract. A developer can specify who can call the contract by using the `@(any)`. `any` is a wildcard variable indicating that anyone can call it. However, it is possible to specify a list of addresses or a specific address using data from the incoming state `@(s1.whitelist | a"<some_address>")`.

If no attributes are specified, then it is assumed that the function is private and internal.

Moving on, `(out: int)` is a return type bound by the variable `out` that can be used to specify a post-execution condition to ensure that the function's output is within the specification. It is also possible to just specify the return type, `fn int my_func(...)`. The `my_func` is an identifier of the function, followed by the list of typed parameters.

Functions in Folidity feature `where` blocks enable developers to specify state transition bounds and are also used to inject the current state into the function's execution context. Certain functions can only be executed if the input state matches the current state. After `->` we have a final state that indicates which state we transition to, this can be the same state, meaning that the function only mutates the current state and doesn't logically advance. Both input and output states can be bound by variables to specify pre and post-mutation constraints. Notice that states' variables are declared differently from other data types. This is a conscious design decision to differentiate the state mutation parts from the traditional manipulation of primitive data.

Additionally, Folidity offers a unique type of function: *view functions*. They are used exclusively for enquiring about current or previous state variables and are explicitly restricted from modifying the state of the contract.

```
view(BeginState s) fn list<address> get_voters() {
    return s.voters;
}
```

These functions are prefixed with the `view(StateName v)` tokens that indicate what state the function accesses. They do not require any attributes since they are public by default and can not be used for instantiation.

Finally, Folidity offers `struct` and `enum` declarations resembling the ones in Rust. They can be used as a type in the variable or field type annotations.

4.3. Formal Verification

Folidity's grammar is structured with first-class support for formal verification in mind. Therefore, the compiler can imply and prove certain mathematical and functional properties of the program directly from the code without the need to perform any context translations, as in the aforementioned solutions.

This chapter illustrates several examples of how model consistency and constraint satisfiability can be proven directly from the source code of a typical Folidity program.

4.3.1. Model consistency

As an example of the theory behind model consistency in SCs, let's look at role-based access. Suppose:

- $* = \{\text{All addresses}\}$
- $M = \{\text{Moderators of the system}\}$
- $A = \{\text{Admins of the system}\}$

Then, we can model a role-based access hierarchy as

$$A \subseteq M \subseteq *$$

Subsequently, given some event for the system `add_mod(a: Address)`, we can define the following invariants for the system:

$$i_0 := \text{card}(A) = 1$$
$$i_2 := \text{card}(B) = 5$$

And the invariant for the event:

$$i_2 \coloneqq c \in A$$

Where

- c - caller's address
- i_n - enumerated invariant with some boolean statement
- card(...) - cardinality of a set

For the denoted event, suppose we mutate the model by adding an address to a set of admins: $A : A \cup \{a\}$

Then, we can verify the model consistency for some state transition from an initial state S to a new state S', $S \to S'$, using propositional logic.

$$\frac{(i_0 \wedge i_1 \wedge i_2) \to A \cup \{a\}, a \in *, c \in A}{A \cup \{a\}}$$

However, as it can be seen, one of the premises violates the invariant, in particular:

$$\frac{\mathrm{card}(A) = 1 \to A \cup \{a\}, a \in *}{A \cup \{a\}}$$

In practice, the following error can be picked at the compile time by using symbolic execution of the code. The other invariant, i_2, can be picked at the runtime by generating an appropriate assertion.

4.3.2. Proving constraint satisfiability

One of the core pieces in the aforementioned workflow is the model bounds, which consist of individual boolean constraints, as shown in Listing 6. Let's break down how each of the selected techniques can be applied to the program written in Folidity. As a good starting point, we can perform a static analysis and verify that the program statements, declarations and constraints are valid and consistent.

A simple approach is to perform semantic analysis that carries out type checking and verification of the correct state transition in the function body. Specifically, if `mutate()` expects to return `StateA` but instead performs a state transition to `StateB`, we can already detect this inconsistency at compile time.

The next stage of the analysis involves verification of the consistency of the models described.

```
# Some model and its constraints
model ModelA {
  x: int,
  y: int
} st [
  x > 10,
  y < 5
]
# A state that encapsulates a model and provides its own constraints.
state StateA(ModelA) st [
  x < y
]
# A function that describes mutation.
fn () mutate(value: int) when (StateA s) -> StateA
st [
    value > 100,
    value < 100,
] { ... }
```

Listing 4.6: Simple folidity program

We can generalise the approach using the following mathematical model. Let's describe some verification system $\boldsymbol{VS}$ as

$\boldsymbol{VS} = \langle \mathbf{M}, \mathbf{E}, \boldsymbol{\Upsilon}, \Theta, \mathrm{T}_{\mathrm{M}}, \mathrm{T}_{\mathrm{E},\{\mathrm{E},\mathrm{M}\}}, \mathrm{T}_{\Upsilon,\mathrm{E}} \rangle$ where

- $\mathbf{M}$ - set of models in the system.
- $\mathbf{E}$ - set of states in the system
- $\boldsymbol{\Upsilon}$ - set of functions in the system.
- Θ - set of of constraint blocks in the system, where $\Theta[\mathbf{M}]$ corresponds to the set of constraints for models, $\Theta[\mathbf{E}]$ - state constraints and $\Theta[\boldsymbol{\Upsilon}]$ function constraints.
- T_{M} - a relation $\mathrm{T} : \mathbf{M} \rightharpoonup \mathbf{M}$ describing a model inheritance.
- $\mathrm{T}_{\mathrm{E},\{\mathrm{E},\mathrm{M}\}}$ - a relation $\mathrm{T} : \mathbf{E} \rightharpoonup \{\mathbf{E}, \mathbf{M}\}$ describing any state transition bounds and encapsulated models in states, that is some state `S'` can only be transitioned to from the specified state `S`, and state some state `S` can encapsulate some model `M`
- $\mathrm{T}_{\Upsilon,\mathrm{E}}$ - a relation $\mathrm{T} : \boldsymbol{\Upsilon} \rightharpoonup \mathbf{E}$ describing any state transition bounds for states $\mathbf{E}$ in functions $\boldsymbol{\Upsilon}$

In particular, $\forall \mu \in \mathbf{M}\ \exists \theta \in \Theta[\mu]$ where θ is a set of constraints for μ, and corresponding logic can be applied for elements of E and Υ.

Then, to verify the consistency of the system, we first need to verify the following satisfiability *Sat*:

$$\forall \mu \in \mathbf{M}$$
$$\exists \theta \in \Theta[\mu]$$
$$\text{s.t. } \theta = \{c_0, c_1, ..., c_k\}$$
$$\left(\bigwedge_i c_i\right) \Rightarrow \mathit{Sat}$$

We can define the following check by some functions $\rho(\theta) : \Theta \to \{\mathit{Sat}, \mathit{Unsat}\}$

which yields the following proof:

$$\exists \theta \in \Theta[e]$$
$$\text{s.t. } \theta = \{c_0, c_1, ..., c_k\}$$
$$\left(\bigwedge_i c_i\right) \Rightarrow \mathit{Sat\ or\ Unsat}$$

This allows to validate the next property of ***VS***

$$A = \{\mathbf{M} \cup \mathbf{E} \cup \Upsilon\}$$
$$A = \{e_0, e_1, ..., e_k\}$$
$$\left(\bigwedge_i \rho(\Theta[e_i])\right) \Rightarrow \mathit{Sat\ or\ Unsat}$$

The next stage is to verify co-dependent symbols in the system for the satisfiability of their respective constraints.

Let's look at the models $\mathbf{M}$, we want to ensure that

$$\text{if for some } m \in \mathrm{M}, m' \in \mathrm{M}$$
$$\exists (m, m') \in \mathrm{T}_{\mathrm{M}}$$
$$\text{s.t. } \rho(m) \times \rho(m') = (\mathit{Sat}, \mathit{Sat})$$
$$\text{and } \theta = \Theta[m] \cup \Theta[m']$$
$$\rho(\theta) \Rightarrow \mathit{Sat}$$

Very similar verification can applied to $\mathrm{T}_{\Upsilon,\mathrm{E}}$.

For $\mathrm{T}_{\mathrm{E},\{\mathrm{E},\mathrm{M}\}}$, the constraints can be extracted in the following way:

$$\text{if for some } \varepsilon \in \mathrm{E}, \varepsilon' \in \mathrm{E}$$
$$\exists (\varepsilon, \varepsilon') \in \mathrm{T}_{\mathrm{E},\{\mathrm{E},\mathrm{M}\}}$$
$$\text{s.t. } \rho(\varepsilon) \times \rho(\varepsilon') \times \rho(\mu) = (\mathit{Sat}, \mathit{Sat})$$
$$\text{and } \theta = \Theta[\varepsilon] \cup \Theta[\varepsilon']$$
$$\rho(\theta) \Rightarrow \mathit{Sat}$$

Similarly,

$$\text{if for some } \varepsilon \in \mathrm{E}, \mu \in \mathrm{M}$$
$$\exists(\varepsilon, \mu) \in \mathrm{T}_{\mathrm{E},\{\mathrm{E},\mathrm{M}\}}$$
$$\text{s.t. } \rho(\varepsilon) \times \rho(\mu) = (Sat, Sat)$$
$$\text{and } \theta = \Theta[\varepsilon] \cup \Theta[\mu]$$
$$\rho(\theta) \Rightarrow Sat$$

After the completing verification of T relations for consistency, we can provide a mathematical guarantee that ***VS*** has been modelled consistently.

Having verified the constraints, we can leverage them as the guards during state transitions and can apply proofs from *temporal logic* to verify that the described state transitions will take place under the described constraints.

In the final stage, we can perform the symbolic execution of instructions in the function bodies with the constraints loaded in the global context of the system. Having tracked the states of different symbols, we can verify each function for reachability for described state transitions and provide strong guarantees of functional correctness of the system described in the smart contract.

4.3.3. Other techniques

The above examples leverage the static analysis of the program to derive its mathematical properties. It is worth noting that it is possible to apply other techniques of formal verification such as symbolic execution and interface discovery [37].

In particular, we can provide even more fine-grained program validation by asserting user-defined constraints in the symbolic execution context. This enables unsatisfability detection and reachability detection at an earlier stage of execution. Traditional methods rely on composing these constraints at the runtime through the statistical discovery of the model bounds, whereas Folidity offers this information at the compile time.

In the context of multi-contract execution, which applies to EVM-compatible blockchains, instead of carrying out interface discovery through statistical methods, we can potentially encode the function signature with its model's bounds and constraints into the metadata and leverage this information at the runtime to verify the model consistency and constraint satisfiability, as illustrated earlier.

5. Implementation

5.1. Outline

The language is implemented using Rust[13] due to its memory-safety guarantees and efficiency. The compiler uses Lalrpop[14] parser-generator to streamline the development process. Folidity also requires SMT-solver for formal verification and generation of runtime assertions. To facilitate this functionality, Z3[15] will be used since it also provides Rust bindings. It was debated to use Boogie, since it provides a higher-level abstraction, but it was quickly discarded due to lack of documentation and increased development time.

As a target blockchain for the language, Algorand[16] has been selected. Algorand is a blockchain platform designed for high-performance and low-cost transactions, utilising a unique consensus algorithm called Pure Proof-of-Stake to achieve scalability, security, and decentralisation [38].

One of the potential drawbacks of Folidity is a computational overhead due to complex abstractions and additional assertions. EVM-based blockchains have varying costs for the execution, i.e. fees, that depend on the complexity of a SC. On the contrary, although Algorand has a limited execution stack, it offers fixed, low transaction fees. Additionally, Algorand execution context explicitly operates in terms of state transition, which perfectly suits the paradigm of Folidity. Finally, Algorand offers opt-in functionality and local wallet storage, allowing users to explicitly opt-in to use the SC.

The Folidity compiler emits Algoran AVM Teal[17] bytecode. It was originally planned to emit an intermediate representation in Tealish[18]. However, this option was soon invalidated due to reduced developer activity in the project and the absence of audits that may compromise the intrinsic security of the Folidity compiler.

Overall, the compilation workflow can be summarised in Figure 4

[13]https://www.rust-lang.org
[14]https://github.com/lalrpop/lalrpop
[15]https://microsoft.github.io/z3guide
[16]https://developer.algorand.org
[17]https://developer.algorand.org/docs/get-details/dapps/avm/teal/
[18]https://tealish.tinyman.org/en/latest

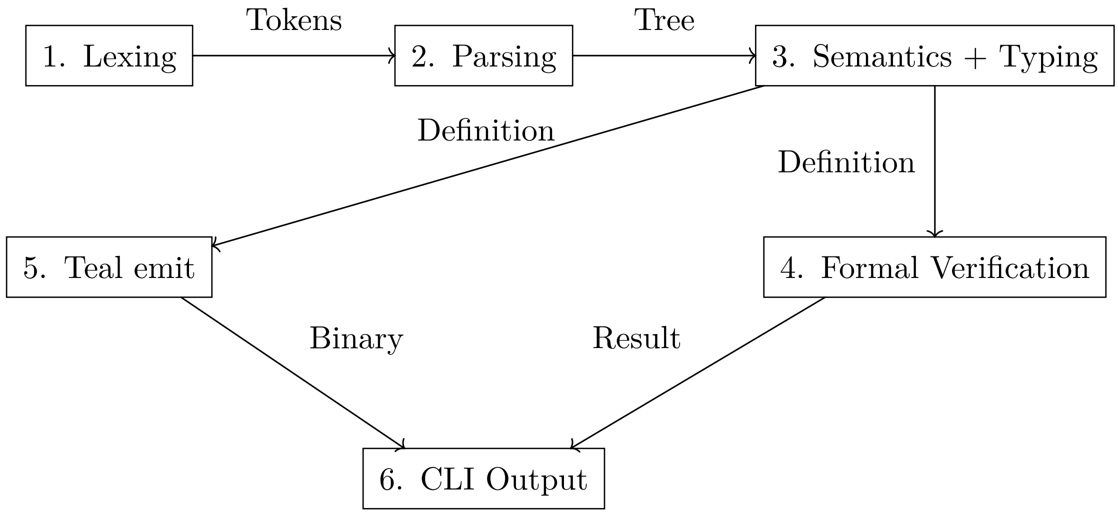


Figure 5.4: Compilation process

Step logics are composed into Rust crates (i.e. modules) for modularity and testability.

Steps 1 and 2 are processed by the `folidity-parser` crate. They produce a syntax AST, which is fed to the `folidity-semantics` for semantic analysis and type checking (step 3). The resulting contract definition is then independently piped into the `folidity-verifier` crate for formal verification (step 4) and the `folidity-emitter` for the final build compilation of binary Teal code (step 5). The artefacts of the compilation and the result of verification are then supplied back to the calling the CLI crate (`solidity`) to display the result to the user and write artefacts into the file (step 6).

5.2. Scope

As part of the development process, it has been decided to limit the scope to supporting only a single SC execution. Cross-contract calls require extra consideration in design and development. Therefore, *SCV6* is only addressed in the theoretical context of this paper. Additionally, optimisation of the execution is not considered relevant at this stage in favour of safety and simplicity. Finally, Algorand offers smart signatures, a program that is delegated a signing authority[19]. As they operate in a different way from SCs, they are also outside the scope of this project.

[19] https://developer.algorand.org/docs/get-details/dapps/smart-contracts/smartsigs

5.3. Diagnostics

The `folidity-diagnostics` module is one of the core pieces of the compiler, it enables the aggregation of a list of reports across multiple crates and its presentation to the user. Folidity compiler offers `folidity-diagnostics` crate that contains `Report` structures

```rust
pub struct Report {
  /// Location of an error
  pub loc: Span,
  /// A type of error to occur.
  pub error_type: ErrorType,
  /// Level of an error.
  pub level: Level,
  /// Message of an error
  pub message: String,
  /// Additional error.
  pub additional_info: Vec<Report>,
  /// Helping note for the message.
  pub note: String,
}
```

Listing 5.7: Report structure used to contain info about the error

At each stage of the compilation, if an error occurs, then the crate composes a `Report` and adds to their respective list of errors which are then returned to the caller and displayed to the user.

5.4. Parser

Parsing has been significantly bootstrapped using Rust crates. Logos[20] is used for tokenisation. It scans strings, matches them against patterns and produces a list of `enum` tokens that can directly be referenced in Rust code. Lalrpop is a powerful parser-generator and library that allows developers to describe grammar using easy-to-use syntax and generate an AST. Its syntax is expressive and effective when managing grammar ambiguities. In addition, Lalrpop provides built-in support for error recovery producing a descriptive list of error reports. Finally, the library has been actively used in the industry by production-ready languages such as Solang[21] and Gluon[22].

[20] https://crates.io/crates/logos
[21] https://github.com/hyperledger/solang
[22] https://github.com/gluon-lang/gluon

```
AccessAttr: ast::AccessAttribute = {
  <start:@L> "@" "(" <first:Expression?> <mut memebers:("|" <Expression>)*>
")" <end:@R> => {
      let mut all = if first.is_some() { vec![first.unwrap()] } else { vec!
[] };
       all.append(&mut memebers);
       ast::AccessAttribute::new(start, end, all)
  }
}
```

Listing 5.8: Example of a Lalrpop rule.

```
pub struct AccessAttribute {
    /// Location of the token.
    pub loc: Span,
    /// Members delimited by `|`
    pub members: Vec<Expression>,
}
```

Listing 5.9: Corresponding Rust struct

As an example, Listing 8 illustrates a typical parsing rule in Lalrpop, that produces the `AccessAttribute` struct in Rust in Listing 9. `<Expression>` is another Lalrpop rule that parses expressions. This way, we can compose different rules and structures together, hence building a tree. `@L` and `@R` tokens allow tracking the location span of a token, which is heavily used in further compilation stages for reporting purposes.

Finally, we do not resolve primitives to concrete Rust types yet. Instead, they are parsed as strings and resolved to a specific type based on the context of the expression, as explained later.

5.5. Semantics & Typing

Semantic analysis is one of the largest parts of the Folidity compiler. It inspects the AST produced by the parser and produces a more concrete definition of the contract as shown in Listing 11. The Folidity uses `GlobalSymbol` and `SymbolInfo` structures to uniquely identify declarations in the codebase. They consist of a symbol's location span and index in the respective list as shown in Listing 10.

```
pub struct SymbolInfo {
    /// Locations of the global symbol.
    pub loc: Span,
    /// Index of the global symbol.
    pub i: usize,
}

pub enum GlobalSymbol {
  Struct(SymbolInfo),
  Model(SymbolInfo),
  Enum(SymbolInfo),
  State(SymbolInfo),
  Function(SymbolInfo),
}
```

Listing 5.10: Symbol structs used for identification.

```
pub struct ContractDefinition {
  /// List of all enums in the contract.
  pub enums: Vec<EnumDeclaration>,
  /// List of all structs in the contract.
  pub structs: Vec<StructDeclaration>,
  /// List of all models in the contract.
  pub models: Vec<ModelDeclaration>,
  /// List of all states in the contract.
  pub states: Vec<StateDeclaration>,
  /// list of all functions in the contract.
  pub functions: Vec<Function>,
  /// Mapping from identifiers to global declaration symbols.
  pub declaration_symbols: HashMap<String, GlobalSymbol>,
  /// Id of the next variable in the sym table.
  pub next_var_id: usize,
  /// Errors during semantic analysis.
  pub diagnostics: Vec<Report>,
}
```

Listing 5.11: Contract definition resolved by the crate.

As part of the checking, the crate first inspects that all declarations have been defined correctly by inspecting signatures, that is, there are no conflicting names. After the successful resolution, we add the structure to the respective list and Then, we can resolve fields of structs, models and states. First, we verify that no model or state is used as the type of a field. Afterwards, the module checks fields for any cycles using Solang algorithm[23]. It builds a directed graph from the fields with and uses the original index of the declaration of the index. It then finds any strongly directed components using Tarjan's algorithm[24]. If we have a simple path between the two nodes, then we have detected a cycle.

After that, we check models and states for any cycles in inheritance. We disallow model inheritance to prevent infinite-size structures, but states do not really have

[23]https://github.com/hyperledger/solang/blob/d7a875afe73f95e3c9d5112aa36c8f9eb91a6e00/src/sema/types.rs#L359
[24]https://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm

this problem since, as stated earlier, functions can transition to the same or any previous state.

Having verified storage-based declarations, we are ready to resolve functions. Each declaration that has expressions also contains a scope. Therefore, when resolving functions, models and states, a scope is created for each declaration.

```
pub struct SymTable {
    /// Variable names in the current scope.
    pub names: HashMap<String, usize>,
    // Context of variables in the given scope.
    pub context: ScopeContext,
}

pub struct Scope {
    /// Indexed map of variables
    pub vars: IndexMap<usize, VariableSym>,
    /// List of scoped symbol tables.
    pub tables: Vec<SymTable>,
    /// Index of the current scope.
    pub current: usize,
    /// What symbol this scope this belongs to.
    pub symbol: GlobalSymbol,
}
```

Listing 5.12: Symbol table and scope used in the crate.

Moving on, the function's attributes are resolved. `is_init` is stored as a boolean flag. When resolving an access attribute, we resolve the respective expressions in the attribute's body and match that the referenced fields exist in the incoming state adding it to the scope. Then, we resolve state bounds while injecting bound variables into the scope. Afterwards, the function's parameters are added to the scope as a variable. The variables are added in the order as they are described to maintain the valid stack of symbol tables that is used to control the variable access as explained later.

Having resolved the function's signature, the crate has finished resolving declaration signatures and is ready to resolve `st` blocks.

Resolving `st` blocks in declarations is done by simply resolving a list of expressions, as explained in Section 5.5.1. During each resolution stage, the scope of the declaration is provided to enable the variable lookup in expressions.

The final stage of the semantic resolution is to resolve the functions' bodies. This is done by inspecting the list of statements with injected the injected function's scope which is explained in Section 5.5.2.

If after each stage of semantic analysis no reports have been pushed, the `ContractDefinition` is returned to the caller, otherwise, the list of `Reports` is returned.

5.5.1. Expressions

Folidity features a type resolution at the compile time, similar to Rust. We define the following `enum` in Listing 13. We use this information to resolve an expression to a specific type.

```
pub enum ExpectedType {
     /// The expression is not expected to resolve to any type (e.g. a
function call)
    Empty,
    /// The expression is expected to resolve to a concrete type.
    /// e.g. `let a: int = <expr>`
    Concrete(TypeVariant),
    /// The expression can be resolved to different types and cast later.
    Dynamic(Vec<TypeVariant>),
}
```

Listing 5.13: Expected type definition.

Each enum is supplied to a function resolving an expression. If the expected type is well-known (i.e. it is declared or can be derived), then `Concrete(...)` variant is supplied. Otherwise, `Dynamic(...)` variant is supplied with the list of possible types that the expression can resolve to. As an example, in `let a: int = 10;` the `10` literal can only be resolved to a signed integer, whereas in `let a = 10;` the literal can be either signed or unsigned. Sometimes, we may not know the expected type, then we use our best effort to resolve the literal to the type it can be resolved to. If the type can not be resolved to any of the specified types, then a report is composed and added to the list of reports.

Similar to AST parsing, complex expression structures are resolved recursively and built up back to the tree of expression.

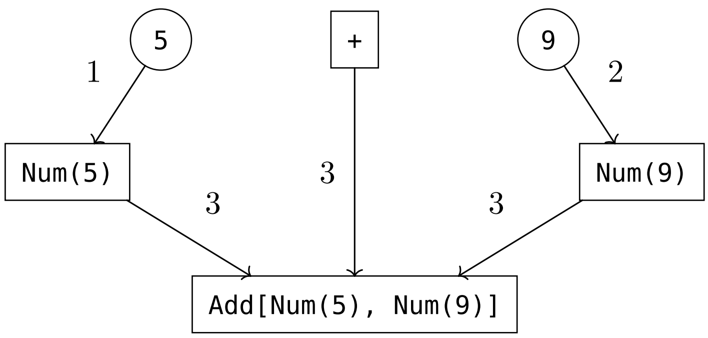


Figure 5.5: Example resolving an expression

The Figure 5 demonstrates how a simple addition is resolved. In step 1, we attempt to resolve the `5`. Assuming there are no concrete expected types, the `5` will resolve to `int`. This implies that the `9` must be resolved to `int` as well. We attempt the resolution. If it fails, we remove `int` from the list of accepted types of `5` and try again. If it succeeds, then `9` is resolved successfully and the top-level function resolving `+` has two concrete expressions. Then, in step 3, we compose expressions together and pack them into the `Add` concrete expression. The compiler additionally performs a basic optimisation of literal expression, e.g. by converting `1 + 1` into `2` literal.

Variables are resolved differently. Each scope contains multiple symbol tables depending how deep the scope goes. When the variable is used in the expression, the function looks up the variable symbol in the scope and its metadata (e.g. type, assigned expression, usage kind).

Certain variables should not be accessed in the function body and vice versa. These variables are state and return bound variables. They can only accessed in the `st` block. Similarly, function parameters should be accessed in the `st` block and function body. Therefore, when retrieving the variable, we inspect the context of the current scope, and depending on it, we either return the variable symbol or an error.

Function calls are resolved similarly by looking up the function's definition in the contract, inspecting the arguments and comparing them to the list of expressions provided as arguments. Piping (`:>`) is simply transformed into the nested function calls, where the first argument of the next function is the function call (or expression) of the previous one.

Struct initialisation is resolved similarly to function calls by comparing the list of arguments to the list of fields. The only difference is that since the models can inherit fields, we recursively retrieve the list of fields of a parent and prepend to the current list.

Finally, accessing a member (e.g. a field of a model) is done by retrieving the list of fields of the definition and checking that it contains the requested field name. The resolved expression contains the `GlobalSymbol` of the struct accessed and the position of the field.

5.5.2. Statements

Statements are resolved iteratively. Starting with the variable declaration, we first resolve the assigned expression if any, and then add it to the current symbol table in the function's scope with resolved or annotated type. Further reassignment of the variable simply updates the current entry in the table if the variable is mutable, that is, it has been annotated with the `mut` keyword.

`If-Else` blocks are resolved by first resolving the conditional expression, and resolving the list of statements in the body, then `else` statements are resolved if any. Since the `else` statements can be another `if`, we achieve `else if {}` block.

The `for` loop is handled first by resolving: a variable declaration statement, conditional expression, incrementing expression. Then the list of statements in the body is resolved. Iterators are resolved similarly. Instead, there are two expressions in the declaration: a binding variable and a list. Folidity has `skip` statement that skips the current iterator of the loop, it is only resolved if the current scope context is the loop.

State transition (`move ...;`) is resolved by first resolving the struct initialisation expression. Then the type of expression is compared to the expected final state of the function. If it mismatches, then the error is reported.

Finally, the `return` statement indicates the termination of the function's execution and any returned data, if any. Similar to the state transition, the return expression type is resolved and compared to the expected return type. Afterwards, we toggle the reachability flag, indicating that any followed expressions in the current scope are unreachable.

5.5.3. Generics

Limited support for generics has been introduced to the Folidity compiler. Although a developer can not currently use them directly in the contract's code, they are added to facilitate the support of built-in functions as part of the standard library, which is planned for future work.

Generic type has similar semantics to `ExpectedType` it contains the list of supported types that the expression can resolve to. Therefore, when `GenericType(Types)` is supplied in the `ExpectedType::Concrete(_)` it is transformed into the `ExpectedType::Dynamic(Types)` and passed for another round of type resolution.

5.6. Verifier

As mentioned earlier, Folidiy offers first-class support for verification as part of the compilation process. `folidity-verifier` heavily relies on Microsoft's work around SMT solver by leveraging their Z3 C++ library in combination with FFI wrapper, z3.rs crate[25].

5.6.1. Z3 basics

Z3 relies on propositional logic to prove the satisfiability of theorems and formulas. This essentially enables symbolic reasoning about the program code. The Z3 toolset consists of formulas comprised of quantifiers, uninterpreted functions, sets, and other Z3 AST symbols. It is followed by solvers that enable asserting formulas into the global proving context and, finally, models containing a list of concrete values assigned to symbols in formulas if they are satisfiable.

[25]https://github.com/prove-rs/z3.rs

Z3 also offers tactics and optimisation techniques which currently beyond the scope of usage of this paper.

5.6.2. Translation to Z3

Folidity compiler assumes a global proving context in the scope of the whole SC code, that is, also symbols and formulas are defined within the single proving context.

To prove the functional correctness of the program, we essentially need to translate Folidity Expression into Z3 AST types. As shown in Section 4.3.2, we want to collect the list of constraints for each declaration and prove their consistency independently of each other. However, we also want to build a graph of relationships between declarations to prove that the combination of their constraints is satisfiable as well.

```
pub struct DeclarationBounds<'ctx> {
    /// `st` location block.
    pub loc: Span,
    /// Links of others declaration.
    pub links: Vec<usize>,
    /// Constraint block of the declaration.
    pub constraints: IndexMap<u32, Constraint<'ctx>>,
    /// Scope of the local constraints.
    pub scope: Z3Scope,
}
```

Listing 5.14: Representation of bounds in declarations

We first resolve models. Since they can inherit each other and refine the constraints, they do not have any links. Consequently, we resolve states and functions, initialise them with empty links and add them to the delay for later resolution of dependencies. During the resolution of each declaration, we add respective fields and parameters to the separate Z3 scope of constants to be referenced in Z3 expressions. Constants in Z3 can be referenced by an unsigned 32-bit integer.

```
pub struct Z3Scope {
    pub consts: IndexMap<String, u32>,
}
```

Listing 5.15: Z3 scope used in the crate

After that, we resolve links in the delays by updating `links` fields with indices of the structure the current declaration depends on. Then, we are ready to transform Folidity `Expression` into the `Z3Expression`.

```rust
pub struct Z3Expression<'ctx> {
    /// Location of the expression
    pub loc: Span,
    /// Element of the expression.
    pub element: Dynamic<'ctx>,
}
```

Listing 5.16: Transformed Z3 expression

Each expression is transformed to Z3 AST type similarly to how it was resolved in semantics. Variables are transformed into the Z3 constants that are identified by integers. If we have a variable or member access that references another structure, the Z3 constant that identifies the symbol in another structure is looked up in its scope and returned. This is done to ensure that when combining two different blocks of constraints, the variables correspond to the same Z3 constants. Specifically, if `StateA` has a field named `a` which is referenced by the `k!3` constant, and some function accesses this variable via `s.a`, that `s.a` is resolved to `k!3` respectively.

Sorts in Z3 enable describing some user-defined datatype. They can be based on some concrete type (i.e. `Sort::int(...)`) or uninterpreted, that is, of some abstract type `A`. Z3 leverages the array theory by McCarthy expressing them as select-store axioms [39]. Z3 assumes that arrays are extensional over function space. Hence, since mapping in Folidity is in space of functions, we can model mapping between two types as an array with the domain of type `A` and range of type `B`.

In the verification context, it is assumed that lists and sets are both can be reduced to some user-defined `set` sort. Similarly, models, structs and states and transformed to uninterpreted types as well.

Consequently, each resolved independent expression then gets bound by a boolean constant that can be used to uniquely track that expression. This binding is happening by boolean implication.

$$k!1 \implies a > 10$$

Figure 5.6: Bound boolean formula

Therefore, if `a > 10` is unsatisfiable, then the `k!1` is unsatisfiable respectively.

The resolved and bound expressions are then packed into the `Contraint` structs with the index of the binding constant.

```
pub struct Constraint<'ctx> {
    /// Location of the constraint in the original code.
    pub loc: Span,
    /// Binding constraint symbol id to track it across contexts.
    ///
    /// e.g. `k!0 => a > 10`
    /// where `0` is the id of the symbol.
    pub binding_sym: u32,
    /// Boolean expression.
    pub expr: Bool<'ctx>,
}
```

Listing 5.17: Constraint strucutre

In the end, a list of expressions in individual `st` blocks gets transformed into the list of constraints.

5.6.3. Constraint satisfiability in individual blocks

After the transformation of expressions, the crate verifies the satisfiability of individual blocks of constraints in each declaration.

For each block, we create a solver with the global context and assert constraints. The solver then executes the verification and creates a model if successful. Otherwise, the unsatisfiability core is extracted from the solver. This core consists of the constants that contradict each other which then get reported.

As an example let's look at the following set of constraints.

$$\begin{aligned} k!0 &\implies s = \text{"Hello World"} \\ k!1 &\implies a > 10 \\ k!2 &\implies b < 5 \\ k!3 &\implies b > a \end{aligned}$$

Figure 5.7: Example of contradicting formulas

From the above set, the solver will return the unsatisfiable core of `[k!1, k!2, k!3]` that contradict each other.

5.6.4. Constraint satisfiability in joined blocks

As a final stage, lists of joined blocks of constraints based on the dependencies between declarations are produced. to eliminate the redundancy in computation, it is essential to compose lists of unique sets of constraints. To achieve that, an undirected graph of linked nodes is assembled first. Then, Tarjan's SCC algorithm is used to find any strongly connected components. In the case of an undirected graph, the strongly connected component corresponds to a set of interconnected nodes that are disjoint from other components as shown in Figure 8.

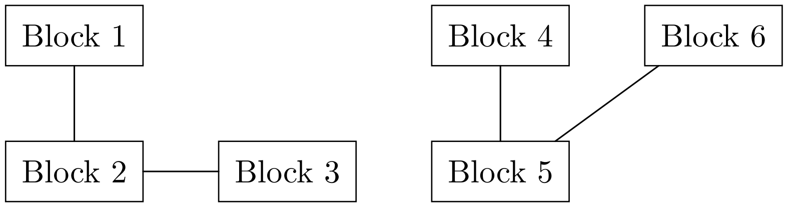


Figure 5.8: Dependency graph of linked declarations

In the example above, the algorithm will return two sets of nodes: `[1, 2, 3]` and `[4, 5, 6]`.

The constraints from these blocks and consequently composed into a single list and verified for consistency in a similar manner.

5.7. Emitter

Emission of the target code is the final stage of the compilation process. Similarly to the verifier, it also accepts `ContractDefintion` as an input. As an output, the crate returns a list of bytes of Teal instructions.

5.7.1. Teal and AVM basics

Teal is a stack-based language that is interpreted by the Alogrand Virtual Machine[26]. The typical smart contract program in Algorand is called the *approval* program. An approval can signal the successful execution by returning a positive number. The stack types consist only of `uint64` and `bytes` types. However, Teal provides a number of pseudo-opcodes that automatically encode the data into the primitive types. As an example `byte "Hello World"` encodes string literal into the byte array of UTF-8 characters.

Like in any stack-based language arguments to expression are popped from the top of a stack. Similarly, it is possible to push numerous onto the stack. As an equivalent to registers, Teal offers the *Scratch Space*. It allows to store up to 256 values during a single execution of a program. Values can be accessed or written using `load i` and `store i` commands. Additionally, Teal offers blockchain-specific opcode to access the transaction information (`txn ..`) and the global state of the consensus (`global ...`).

AVM offers different types of storage spaces, global, local and boxes. Global storage can contain up to 64 values of a total of 8K bytes and paid for by the contract's creator. Local storage requires the user to opt into the contract and can only offer 16 values, the opted user is responsible for funding it. Box storage has no limit on the number of "boxes" a contract can create, and offers up to 32K bytes of

[26]https://developer.algorand.org/docs/get-details/dapps/avm/teal/specification

storage capacity. Therefore, box storage was chosen to allow for more flexibility in the storage functionality.

5.7.2. Entrypoint code generation

First, the crate emits the boilerplate code to handle incoming arguments and direct execution flow to the respective function. If the `ApplicationID` of the current transaction is `0`, then it means that the contract is being created, and we direct execution to the constructor function. Then we handle other AVM-specific operations such as application update, deletion, etc. Afterwards, if the current transaction call is the call to the function, then we parse the arguments in order. The first argument corresponds to the function name encoded as bytes. If the function exists, then branch to the respective function blocks; otherwise, an error is returned.

The next stage of boilerplate code generation is the function call blocks. Functions can return different values, but the AVM expects the code to be returned at the end of the execution. Therefore, the emitter creates a labelled wrapper block for each function where the return value is logged before the code is returned. Any arguments that are meant to be supplied to the function are pushed onto the stack, the `callsub` opcode then redirects the execution flow to the specific label indicating the start of the function where the arguments are handled.

```
__block__incr_by:
txn ApplicationArgs 1
callsub __incr_by
pushint 1
return
```

Listing 5.18: Example fo wrapper function block

5.7.3. Function resolution

For each function, the emitter creates a mapping of concrete chunks for variables that allow access to concrete values. These chunks are then prepended whenever the variable is referenced. As a starting point, the emitter saves the current arguments to the function into the scratch and adds `load i` chunks for each respective variable to the map. A similar procedure is done for the state variables and access variables.

The emitter then resolves each statement iteratively, walking down the AST for each expression and emitting chunks of opcodes.

5.7.3.1. Expressions

Primitive types are pushed onto the stack using the standard set of opcodes, which allows the encoding of non-standard types into AVM-recognised types. For signed integers, twos-complement conversion is performed. Floating point numbers are encoded using the IEEE 754 standard. Variables are resolved by accessing concrete chunks and prepending them to the current piece of a stack.

Enums are resolved by encoding them into a byte array of 2 uint64 integers where the first value indicates the index of the enum in the contract definition, and the other one corresponds to the variant's position. Binay operations (e.g. summation, subtraction) are emitted by pushing left and right values onto the stack before pushing the operation opcode.

Moving to the complex operations, function calls are emitted similarly to the top-level call by pushing arguments onto the stack before the subroutine call. The most complex part is the handling of the storage-based operations. Currently, Folidity offers limited support for dynamically sized types (e.g. lists, strings, etc.). They are assumed to have a fixed capacity that is 512 bytes. Therefore, all types have a known size at the compile time. This is heavily used when encoding structures as a byte array that gets pushed to the storage. An example of the binary layout shown in Table 1 for the struct listed in Listing 19 demonstrates how any data structures can be encoded.

```
struct MyStruct {
  a: int,
  b: list<int>,
  c: address
}
```

Listing 5.19: An example of encoded struct

`a`: 8 bytes	`len_b`: 8 bytes	`b`: 512 bytes	`c`: 32 bytes

Table 5.1: Struct layout representation

When working with states, the layout representation is calculated from either the raw fields of the state declaration or the model's fields and its parents, that the state encapsulates.

Dynamic generation of assertions is done when any structure with an `st` block is instantiated. Firstly, the byte array is composed as described above and stored in the scratch. Secondly, each field is extracted by calculating the offset of a field in the array which is derived from the definition of the structure, the same procedure is done for the member access expression. Each field's data gets stored in the scratch, and the `load i` chunk gets added to the map of concrete vars. Then, each constraint expression is pushed onto the stack, followed by the `assert` opcode. After all expressions have been emitted, the original byte array is loaded from the scratch space.

5.7.3.2. Statements

Resolving statements is done similarly to expressions. Variable declaration and assignment result in the storage of value in the scratch and the update of the map of concrete chunks.

`If-else` blocks are emitted with the use of branching opcodes. The emits the indexed `else` and `end` labels. `end` label is appended after the `else` block of statements. `else` label is pushed before the `else` statements. If the condition in the `if` block is satisfied, then the execution jumps to the `end`; otherwise, it goes to the `else` block.

The `For` loop is also handled with labels. The `end` label is pushed at the end. If the increment condition is `false`, then a jump to the `end` occurs; otherwise, the body of the loop is executed, followed by the incrementer instructions that are prepended with the `incr` label. The `incr` label is introduced for the use of the `skip` statement, which simply jumps to it whenever present in the loop context.

When the state transition statement `move` is present, the emitter first performs the expression resolution as explained earlier, and then, similarly to the struct constraint assertion, it asserts any function's state bounds if present. Afterwards, it saves the corresponding byte array in the box named with the state identifier.

The `return` statement is handled similarly and followed by the `retsub` opcode, which returns the execution to the subroutine caller point.

5.7.4. Final compilation

You can view a sample smart contract written in Folidity and the resulting Teal code listed in Appendix D. After all the functions have been emitted, the emitter writes them into a single string and encodes them as a byte array which gets returned to the caller.

Algorand also requires a *clear* program that contains an account opt-out logic. As this functionality is not currently used, the compiler emits a binary that returns a `0` status code.

5.8. CLI

The entry point to the Foldity compiler is the `folidity` crate that represents a CLI tool. It allows to:

- Generate a template - `folidity new ...`
- Semantically check the source code - `folidity check ...`
- Formally verify the contract - `folidity verify ...`
- Compile the contract to Teal executing previous stages as well - `folidity compile ....`

More commands and options can be displayed via `folidity help` or `folidity <command> help` commands.

It interacts with the aforementioned crates directly and retrieves the result if successful which is passed down to the next stage of compilation. If errors are returned, the crate uses `ariadne`[27] crate to nicely format and display the error and associated code with it.

An example of a failed formal verification report is shown in Figure 9.

Figure 5.9: Example of an error report

[27] https://github.com/zesterer/ariadne

6. Testing strategies

During the development test-driven development and extreme programming techniques were applied. Each crate was extensively tested using unit tests. An output is displayed in Listing 20.

Furthermore, an integration test of the example counter contract and GitHub CI workflows have been introduced to acquire architecture-independent results and better code quality.

The CI workflow stages include:

- Tests
- Check clippy and documentation
- Check formatting
- Compile and formally test examples

Finally, the `dependabot`[28] protects the compiler from supply chain attacks and outdated dependencies.

[28] https://github.com/dependabot

```
test tests::simple_exprs ... ok
test tests::test_simple_emit ... ok
test tests::test_complex_emit ... ok

...

test tests::comment_token ... ok
test tests::simple_int ... ok
test tests::strings ... ok
test tests::simple_mixed_numbers ... ok
test tests::simple_floats ... ok
test tests::test_factorial ... ok
test tests::test_lists ... ok
test tests::test_simple_func ... ok
test tests::test_structs_enums ... ok
test tests::test_factorial_tree ... ok
test tests::parse_complete_program ... ok

...

test expression::tests::test_mul ... ok
test expression::tests::test_eval ... ok
test expression::tests::test_list ... ok
test expression::tests::member_access ... ok
test expression::tests::init_struct ... ok
test expression::tests::test_var ... ok
test tests::test_first_pass ... ok
test expression::tests::pipe ... ok
test expression::tests::test_func ... ok
test tests::test_err_program ... ok
test tests::test_program ... ok

...

test tests::mul_transform ... ok
test tests::string_hex_transform ... ok
test tests::var_transform ... ok
test tests::list_transform ... ok
test tests::in_transform ... ok
test tests::test_incorrect_linked_bounds ... ok
test tests::test_incorrect_bounds ... ok
test tests::test_correct_bounds ... ok
```

Listing 6.20: Test cases

7. Limitations and Future Work

7.1. Formal verification

The theoretical model currently lacks more concrete proof of the functional correctness of the symbolic execution techniques. While this is theoretically achievable, more work and research should be done to determine how well the current version of grammar can facilitate such an approach. While it is possible to implement the symbolic execution of statements and expressions in the verifier crate. This objective was beyond the scope of this paper. Additionally, the current implementation of the verifier does not function calls and struct initialisation in expression as it requires further work on the expression transformation to be done, potentially using the *array theory*. Finally, resolving generic types in the verifier requires coercing the expression to a concrete type, which is not a trivial type, since it is only achievable with the introduction of monomorphisation.

7.2. Semantic analysis & Grammar

As mentioned before, the groundwork for generic types has been laid. However, their use is limited only to the internal code generation of standard library functions, which has not been done either. It requires further adjustments to grammar and support for monomorphisation, similar to verification.

Additionally, object destructuring and struct instantiation-by-object were originally planned but were not implemented due to the limited time available for the project. However, the current project structure allows for the introduction of these features in the future.

Since the standard library has not been introduced, it is currently not possible to handle operational side effects such as integer overflow. This partially affects the *SCV5* and *Requirement 8*. However, similar safety guarantees can be achieved by the right use of constraints. Furthermore, Algorand automatically handles integer overflows by immediately returning an error.

Finally, type-casting is currently unsupported, which may result in false negatives in type-checking. This requires further refinements of grammar by potentially introducing the `as` keyword.

7.3. Emitter

Emitter is currently at the pre-alpha stage. It produces unoptimised Teal code. This can be addressed by the introduction of control flow graphs that can efficiently eliminate redundant code. Multi-pass compilation should also be considered as an optimisation strategy. Additionally, the current iteration does not support the inclusion operator (`in`) that restricts the use of iterators and `a in list` expression in the code. This can be achieved by revising the layout generation of an array, and, similar to member access, accessing each element in the list by offset.

Finally, support for multiple targets such as EVM should be one of the priorities for future development to access a larger system of developers to acquire more developer feedback, hence, fostering the community development of the compiler.

7.4. General remarks

Due to the reduced scope of the project, multi-contract support has not been implemented. It requires the finalisation of the grammar of the language that can lead to the development of the language-specific ABI[29] metadata that can be used by CLI tools and contracts to call the contract. It also further required conducting more research in the field of interface discovery to provide evidence that it is possible to achieve a reasonable level of security across multiple SCs when the source code is not available.

Finally, the corresponding CLI tools should be developed to enable ABI metadata generation.

[29]Abstract Binary Interface

8. Project Planning

Project development did not see any major deviation from the original plan. This was achieved via rigorous planning and iterative feature development. Each module was written as a separate pull request on GitHub, which enabled me to review the existing code and test it carefully before beginning the development of the next one. I reviewed pull requests myself before merging them into the main branch, which allowed me to evaluate features I had implemented carefully and document them for future reference.

However, the development of the semantics crate took slightly more time than expected due to the realisation of the importance of the functional correctness of the crate in the scope of other crates. Therefore, more time was dedicated to testing and development which shifted the deadline for the rest of the planned work. Section F illustrates the actual time it took to complete the project and the updated objectives.

9. Final words

This paper's goal was to prove that formal verification techniques can be integrated into the development and compilation process as first-class citizens. It addresses numerous problems associated with the security and safety of typical SCs and provides theoretical and practical proof of how they can be addressed at the language level.

While not all the essential features have been implemented, this iteration of the project has laid the groundwork for the future development of the compiler and language grammar. The mathematical model proves that planned features can and should be addressed.

We have also seen recent improvements in the performance of SMT solvers such as Z3, which enable fast and efficient computation on SMT formulas. This further proves the viability of introducing a formal verification stage into the compilation process that can eliminate the mentioned vulnerabilities without any compromises in the developer experience.

However, this project does not account for the networking effect of the language's adoption. Therefore, whether SC developers are ready to adopt a new SCL when a new SCL is released every month is an open question that can only be resolved with time.

Appendix

A. Project brief

Problem Statement

With the rise of blockchain technologies, smart contracts (SC) allowed developers to create complex and resilient applications providing services to end users. However, there have been numerous instances of attacks associated with decentralized applications, involving re-entrance attacks, forced value sends, variable overflows, and incorrectly coded state checks.

This is a result of the dominating nature of procedural style in SC languages such as Solidity, Vyper, PyTeal, and others that make the state transition implicit and hidden from the developers. SC developers then need to opt in for formal verification tools such as KEVM or KAVM that prolong the development process and require specialised knowledge of formal verification.

Proposed Solution

This project proposes the development of a functional SC language with an explicit state transition and model checks. SCs run in a restricted and sandboxed environment and take part in the state transition of the blockchain state machine. This provides a pure functional context that is suitable for a functional programming language.

Folidity intends to be an SC language with a purely functional programming style that allows developers to reason about their code as a combination of state transition functions. The language also enables developers to describe storage and state transition functions using constraints and invariants that the compiler will use to formally verify the functional correctness and consistency of a storage model, inspired by Event-B model verification.

Folidity targets the Algorand Virtual Machine (AVM). AVM is famous for its stack-based assembly language, Teal, which provides fine control over execution. Programs on AVM are also reasoned in the form of stateful and stateless applications. This philosophy perfectly aligns with the programming model of Folidity. The most important benefit of opting for AVM is the fixed SC execution costs which is highly important for a prototype language.

Scope

The project involves syntax design, the development of a compiler that produces intermediate representation code in Teal, and technical analysis of a sample SC demonstrating and proving its functional safety.

Due to limited time, this project will only focus on a single-contract execution, leaving the support of cross-contract calls beyond the scope of the project.

B. Folidity Grammar

```
<program>       := <decl>+

<decl>          := <func_decl> | <model_decl> | <state_decl> | <enum_decl>
| <struct_decl>

<func_decl>     :=  `@init`? <attrs>+ <view>? `fn` <type_decl> <ident> `(`
<params>? `)` <state_bound>? <st_block>? `{` <func_body> `}`
<type_decl>     := <type> | `(` <param> `)`

<attrs>         := `@` `(` <attr_ident> `)`
<attr_ident>    := <ident> | ( <expr> `|` )*
<params>        := <param> | <param> (`,` <params>)*
<param>         := <ident> `:` <type>
<view>           := `view` `(` <state_param> `)`
<state_bound>  := `when` <state_param> <arr> ( <state_param> | <state_param>
(`,` <state_param>)*)
<func_body>     := (<statement>)*
<state_param>   := (<ident> <ident>?) | `()`

<st_block>      := `st` <expr>

<statement>      := <var> | <assign> | <if> | <for> | <foreach> | <return>
| <func_call> | <state_t> `skip` `;`
<state_t>       := `move` <struct_init>
<var>           := let `mut`? <var_ident> (`:` <type>)? (`=` <expr>)?
<var_ident>     := (<ident> | <decon>)
<decon>         := `{` <decon_list> `}`
<decon_list>    := <ident> | <ident> (`,` <decon_list>)*

<assign>        := <ident> `=` <expr>
<if>            := `if` <expr> ` `{` <statement> `}` (`else` <if>? )*
<foreach>        := `for` `(` <var_ident> `in` (<ident> | <range>) `)` `{`
<statement> `}`
<for>          := `for` `(` <var> `;` <expr> `;` <expr> `)` `{` <statement>
`}`
<return>        := `return` <expr>
<struct_init>   := <ident> : `{` <struct_args> `}`
<struct_args>   := <expr> | (`,` <expr>)* | <arg_obj>
<struct_arg>    := <ident> `:` <expr>
<arg_obj>       := `|` `..` <ident>

<model_decl>    := `model` <ident> `{` params `}` <st_block>?

<state_decl>    := `state` <ident> (`from` <ident> <ident>)?  <state_body>
```

```
<st_block>?
<state_body>    := `(` <ident> `)` |  `{` params `}`
<enum_decl>     := `enum` `{` (<ident> | <ident> (`,` <ident>)* ) `}`
<struct_decl>   := `struct` `{` <params> `}`

<type>          := `int` | `uint` | `float` | `char` | `string` | `hex`
                | `address` | `()` | `bool` | <set_type> | <list_type> |
<mapping_type>


<set_type>      := `Set` `<` <type> `>`
<list_type>     := `List` `<` <type> `>`
<mapping_type> := `Mapping` `<` <type> <mapping_rel> <type> `>`
<mapping_rel>   := (`>`)? `-` (`/`)? (`>`)? `>`

<char>          := ? '` <char>* `'`
<hex>           := `hex` `"` <char>* `"`
<address>       := `a` `"` <char>* `"`

<digit>         := [0-9]
<number>        := <digit>+

<bool>          := `true` | `false`
<rel>           := `==` | `!=` | `<` | `>` | `<=` | `>=` | `in`
<bool_op>       := `||` | '&&'

<period>        := `.`
<float>         := <number> <period> <number>?

<func_pipe>     := <expr> (`:>` <func_call>)+
<member_acc>    := <expr> (`.` <ident>)+
<func_call>     := <ident> `(` <args>? `)`
<args>          := <expr> | (<args> `,`)*


<plus>          := `+`
<minus>         := `-`
<div>           := `/`
<mul>           := `*`
<not>           := `!`
<modulo>        := `%`
<expr>          := <not>? <expr_nested>
<expr_nested>   := <term> <bool_op> <expr>
<cond>          := <expr> <rel> <expr>
<math_expr>     := <term> ( (<plus> | <minus>) <term> )*
<term>          := <factor> ( (<mul> | <div> | <modulo>) <factor> )*
<factor>          := <ident> | <constant> | <func_call> | <func_pipe> |
<member_acc> | `(` <expr> `)`
<constant>     := <number> | <float> | <bool> | <string> | <hex> | <address>
| <list>
<list>          := `[` ( <expr>? | <expr> (`,` <expr>)* ) `]`
<ident>         := <char>+
<arr>           := `->`
```

- `<ident>` - eBNF element
- `?` - optional element
- `(    )` - grouping
- `+` - one or more
- `*` - zero or more
- `ident` - literal token

C. Libraries Used

```
  logos = "0.14"
  lalrpop-util = "0.20"
  lalrpop = "0.20"
  thiserror = "1.0"
  syn = "2.0"
  synstructure = "0.13"
  proc-macro2 = "1.0"
  quote = "1.0"
  indexmap = "2.2"
  petgraph = "0.6.4"
  num-bigint = "0.4"
  num-rational = "0.4"
  num-traits = "0.2"
  algonaut_core = "0.4"
  hex = "0.4"
  regex = "1.10"
  clap = { version ="4.5", features = ["derive"]}
  ariadne = { version = "0.4", features = ["auto-color"] }
  anyhow = "1.0"
  walkdir = "2.5"
  yansi = "1.0"
  # we need to pin to commit as the crate version doesn't allow us to
detect local `z3` binary.
   z3  =   { git  = "https://github.com/prove-rs/z3.rs.git",  rev  =
"247d308f27d8b59152ad402e2d8b13d617a1a6a1" }
  derive_more = "0.99"
```

Listing 3.21: `Cargo.toml` file

D. Emit example

An example of a Folidty SC that represents counter application with some additional function to demonstrate other statements and expressions used and the compiled teal approval program.

Folidity Smart Contract

```
state CounterState {
    counter: int,
} st [
    # example bounds
    counter < 1000,
    counter > -1000
]

# This is an constructor.
@init
# Anyone can call this function.
@(any)
fn () initialise() when () -> CounterState {
    loops(5.0);
    conditionals(false, 10);
    move_state();
    move CounterState : { 0 };
}

@(any)
fn () incr_by(value: int) when (CounterState s) -> CounterState
st [
    value > 100,
] {
    let value = s.counter + value;
    move CounterState : { value };
}

@(any)
fn () decr_by(value: int) when (CounterState s) -> CounterState
st [
    value > 100,
] {
    let value = s.counter - value;
    move CounterState : { value };
}
```

```
@(any)
view(CounterState s) fn int get_value() {
    return s.counter;
}

fn () loops(value: float) {
    for (let mut i = 0; i < 10; i + 1) {
        let value = value + 123.0;
        skip;
    }
    let some_list = [-3, 4, 5];
}

fn () conditionals(cond: bool, value: int) {
    let scoped = -10;
    let mut s = s"Hello";
    s = s + s" " + s"World";
    if cond {
        let a = scoped + 3;
    } else if value > 1 {
        let b = scoped + 4;
    } else {
        let c = scoped + 5;
    }
}

fn () move_state() when (CounterState s1) -> (CounterState s2) {
    let a = a"2FMLYJHYQWRHMFKRHKTKX5UNB5DGO65U57O3YVLWUJWKRE4YYJYC2CWWBY";
    let b = [1, 2, 3];
    let c = -5;
    let d = s"Hello World";

    let counter = s1.counter;


    move CounterState : { counter };
}
```

Compiled Teal code

```
#pragma version 8

txn ApplicationID
pushint 0
==
bz create_end
b __block__initialise
create_end:
txn OnCompletion
pushint 0
==
bnz on_call
txn OnCompletion
```

```
pushint 5
==
bnz check_creator
txn OnCompletion
pushint 1
==
bnz fail
txn OnCompletion
pushint 2
==
bnz fail
txn OnCompletion
pushint 4
==
bnz check_creator
err


check_creator:
txn Sender
global CreatorAddress
==
assert
pushint 1
return


fail:
pushint 0
return


on_call:
txna ApplicationArgs 0
pushbytes "initialise"
==
bnz __block__initialise
txna ApplicationArgs 0
pushbytes "incr_by"
==
bnz __block__incr_by
txna ApplicationArgs 0
pushbytes "decr_by"
==
bnz __block__decr_by
txna ApplicationArgs 0
pushbytes "get_value"
==
bnz __block__get_value
txna ApplicationArgs 0
pushbytes "loops"
==
bnz __block__loops
```

```
txna ApplicationArgs 0
pushbytes "conditionals"
==
bnz __block__conditionals
txna ApplicationArgs 0
pushbytes "move_state"
==
bnz __block__move_state
err


__block__initialise:
callsub __initialise
pushint 1
return


__block__incr_by:
txn ApplicationArgs 1
callsub __incr_by
pushint 1
return


__block__decr_by:
txn ApplicationArgs 1
callsub __decr_by
pushint 1
return


__block__get_value:
callsub __get_value
log
pushint 1
return


__block__loops:
txn ApplicationArgs 1
callsub __loops
pushint 1
return


__block__conditionals:
txn ApplicationArgs 1
txn ApplicationArgs 2
callsub __conditionals
pushint 1
return
```

```
__block__move_state:
callsub __move_state
pushint 1
return




__initialise:


pushint 4617315517961601024
callsub __loops


pushint 0
pushint 10
callsub __conditionals


callsub __move_state


pushbytes "__CounterState"
pushint 8
bzero
store 0
pushint 0
store 1
load 0
load 1
replace 0
store 0
load 0
pushint 0
extract_uint64
pushint 1000
<
assert
load 0
pushint 0
extract_uint64
pushint 18446744073709550616
>
assert
load 0
box_put


retsub


__incr_by:
```

```
store 0
load 0
pushint 100
>
assert


pushbytes "__CounterState"
box_get
assert
store 2
load 2
pushint 0
extract_uint64
load 0
+
store 3


pushbytes "__CounterState"
pushint 8
bzero
store 4
load 3
store 5
load 4
load 5
replace 0
store 4
load 4
pushint 0
extract_uint64
pushint 1000
<
assert
load 4
pushint 0
extract_uint64
pushint 18446744073709550616
>
assert
load 4
box_put


retsub


__decr_by:
store 1
load 1
pushint 100
>
```

```
assert


pushbytes "__CounterState"
box_get
assert
store 6
load 6
pushint 0
extract_uint64
load 1
-
store 7


pushbytes "__CounterState"
pushint 8
bzero
store 8
load 7
store 9
load 8
load 9
replace 0
store 8
load 8
pushint 0
extract_uint64
pushint 1000
<
assert
load 8
pushint 0
extract_uint64
pushint 18446744073709550616
>
assert
load 8
box_put


retsub


__get_value:


pushbytes "__CounterState"
box_get
assert
store 10
load 10
pushint 0
```

```
extract_uint64
retsub


retsub


__loops:
store 2


pushint 0
store 11
load 11
pushint 10
<
bnz 0_loop_end

load 2
pushint 4638355772470722560
+
store 12


b 0_loop_incr

0_loop_incr:
load 11
pushint 1
+
0_loop_end:


pushint 18446744073709551613
pushint 4
concat
pushint 5
concat
store 13


retsub


__conditionals:
store 3
store 4


pushint 18446744073709551606
store 14
```

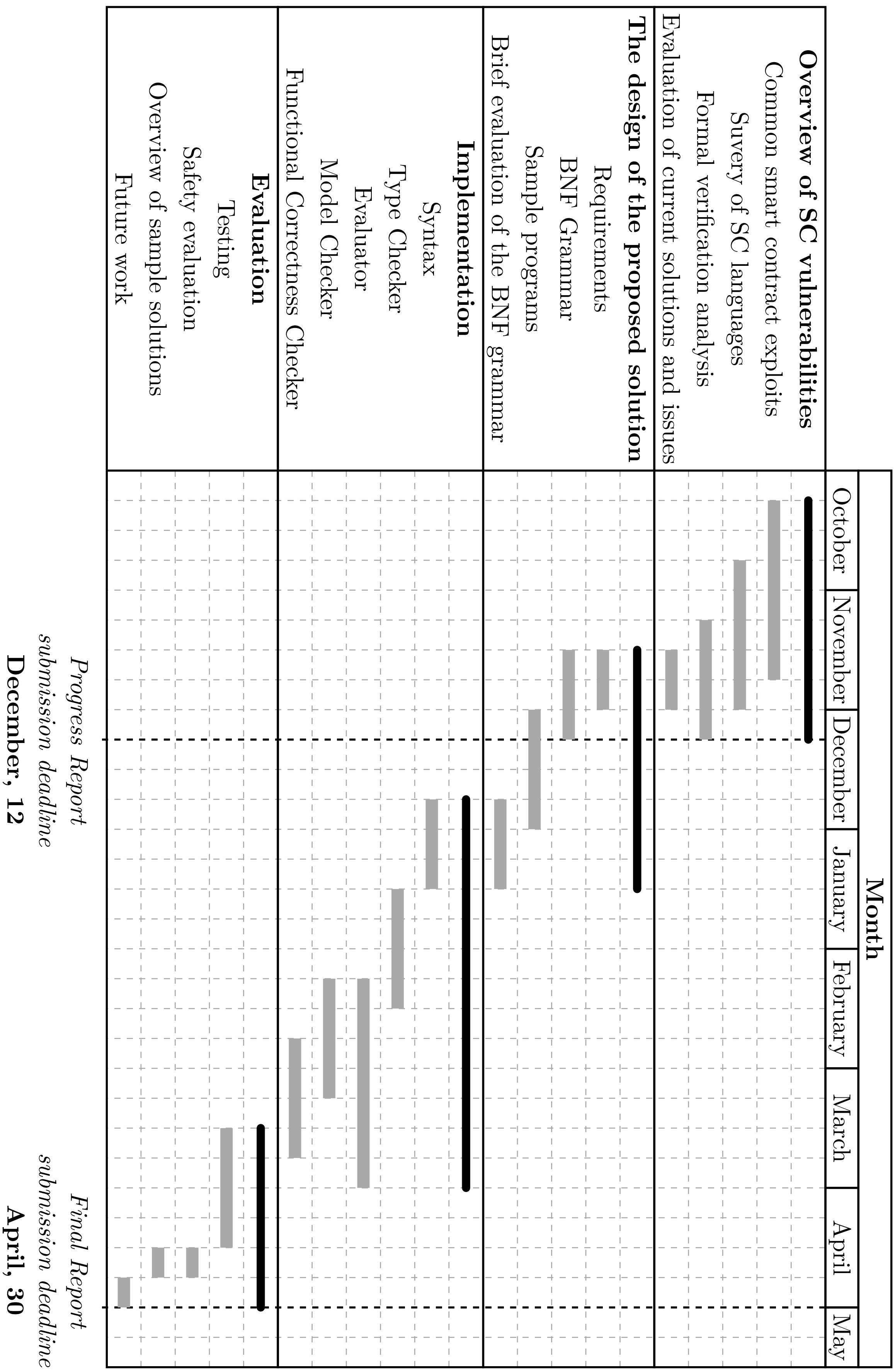
```
pushbytes "Hello"
store 15


load 15
pushbytes " "
concat
pushbytes "World"
concat
store 15


load 3
bz 5_else


load 14
pushint 3
+
store 16


b 5_if_end
5_else:

load 4
pushint 1
>
bz 6_else


load 14
pushint 4
+
store 17


b 6_if_end
6_else:


load 14
pushint 5
+
store 18


6_if_end:

5_if_end:


retsub
```

```
__move_state:

addr 2FMLYJHYQWRHMFKRHKTKX5UNB5DGO65U57O3YVLWUJWKRE4YYJYC2CWWBY
store 19

pushint 1
pushint 2
concat
pushint 3
concat
store 20

pushint 18446744073709551611
store 21

pushbytes "Hello World"
store 22

pushbytes "__CounterState"
box_get
assert
store 23
load 23
pushint 0
extract_uint64
store 24

pushbytes "__CounterState"
pushint 8
bzero
store 25
load 24
store 26
load 25
load 26
replace 0
store 25
load 25
pushint 0
extract_uint64
pushint 1000
<
assert
load 25
pushint 0
```

```
extract_uint64
pushint 18446744073709550616
>
assert
load 25
store 27
load 27
box_put


retsub
```

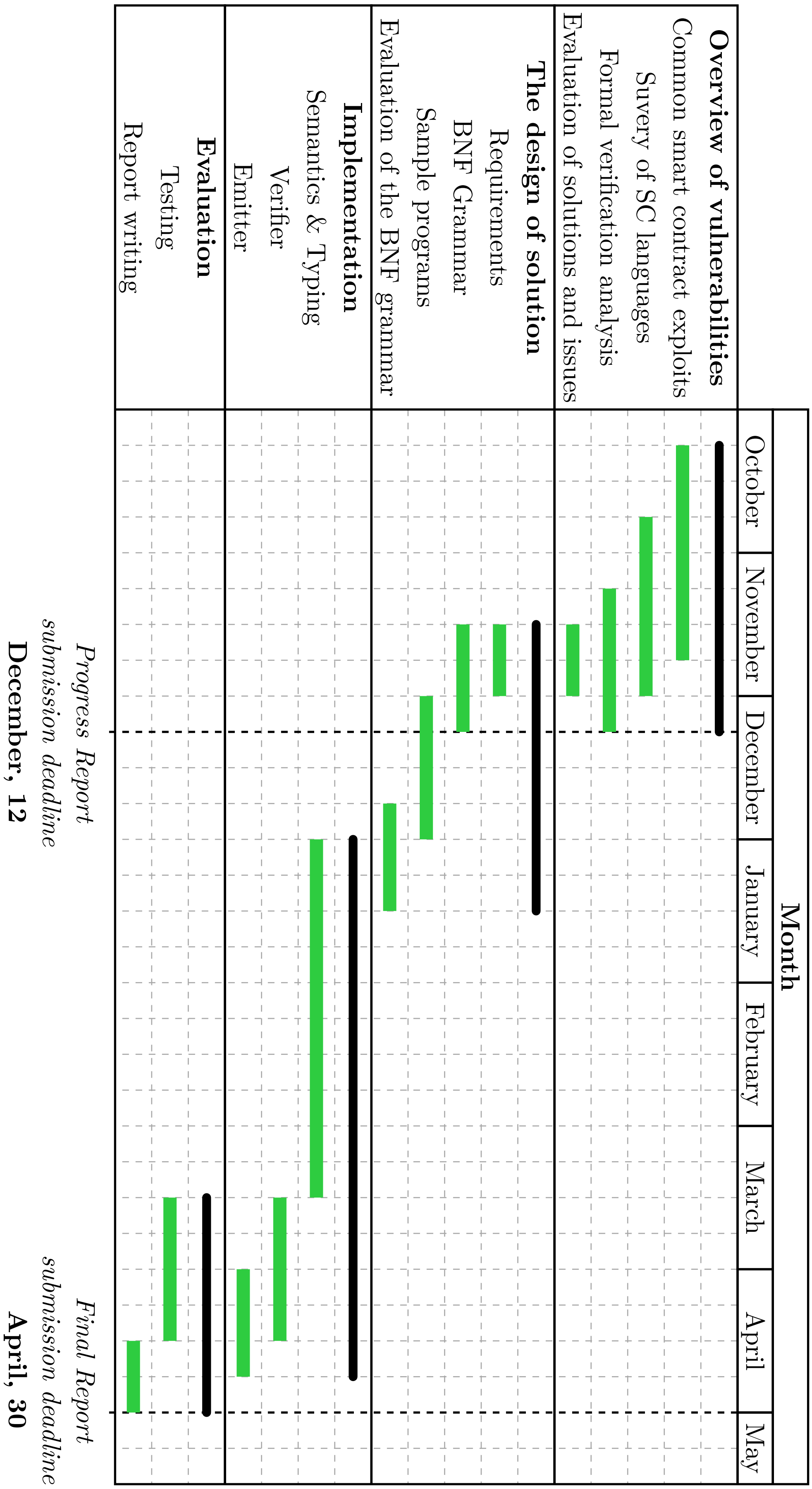
E. Old Gannt Chart

The Gannt chart that demonstrates the planned work.


Month
October
November
December
January
February
March
April
May
Overview of SC vulnerabilities
Common smart contract exploits
Suvery of SC languages
Formal verification analysis
Evaluation of current solutions and issues
The design of the proposed solution
Requirements
BNF Grammar
Sample programs
Brief evaluation of the BNF grammar
Implementation
Syntax
Type Checker
Evaluator
Model Checker
Functional Correctness Checker
Evaluation
Testing
Safety evaluation
Overview of sample solutions
Future work
Progress Report submission deadline
December, 12
Final Report submission deadline
April, 30

F. Actual Gannt Chart

The Gannt chart that demonstrates the actual work.


Month
October
November
December
January
February
March
April
May
Overview of vulnerabilities
Common smart contract exploits
Suvery of SC languages
Formal verification analysis
Evaluation of solutions and issues
The design of solution
Requirements
BNF Grammar
Sample programs
Evaluation of the BNF grammar
Implementation
Semantics & Typing
Verifier
Emitter
Evaluation
Testing
Report writing
Progress Report submission deadline
December, 12
Final Report submission deadline
April, 30

G. Archive Table of Contents

root

- `README.md` description
- `CHANGELOG.md`
- `Cargo.toml` dependencies
- `[crates]` crates locations
  - `[derive_node]` macros
  - `[diagnostics]` `folidity-diagnostics` crate
  - `[emitter]` `folidity-emitter` crate
  - `[parser]` `folidity-parser` crate
  - `[semantics]` `folidity-semantics` crate
  - `[verifier]` `folidity-verifier` crate
  - `[folidity]` `folidity` crate
- `[examples]` example contracts
  - `[counter]` Counter contract

H. Word count

The body of the report has **9476 words** counted using Typst *wordometer*[30].

[30] https://typst.app/universe/package/wordometer

Bibliography

[1] N. Szabo, "Smart Contracts," in *Nick Szabo's Papers and Concise Tutorials*, 1994. [Online]. Available: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html

[2] N. Szabo, "Smart Contracts: Building Blocks for Digital Markets," in *Nick Szabo's Papers and Concise Tutorials*, 1996. [Online]. Available: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html

[3] D. G. Wood, "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER." [Online]. Available: https://gavwood.com/paper.pdf

[4] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System." 2008.

[5] E. Foundation, "Solidity." [Online]. Available: https://docs.soliditylang.org/en/v0.8.23

[6] S. D. Levi, A. B. Lipton, Skadden, Arps, Slate, and M. & Flom LLP, "An Introduction to Smart Contracts and Their Potential and Inherent Limitations." [Online]. Available: https://corpgov.law.harvard.edu/2018/05/26/an-introduction-to-smart-contracts-and-their-potential-and-inherent-limitations

[7] T. Dounas and D. Lombardi, "A CAD-Blockchain Integration Strategy for Distributed Validated Digital Design Connecting the Blockchain," 2018, p. . doi: 10.52842/conf.ecaade.2018.1.223.

[8] X. Zhao, Z. Chen, X. Chen, Y. Wang, and C. Tang, "The DAO attack paradoxes in propositional logic," in *2017 4th International Conference on Systems and Informatics (ICSAI)*, 2017, pp. 1743–1746. doi: 10.1109/ICSAI.2017.8248566.

[9] "King of the Ether: Post-Mortem Investigation." [Online]. Available: https://www.kingoftheether.com/postmortem.html

[10] H. Gall, "Functional safety IEC 61508 / IEC 61511 the impact to certification and the user," 2008, pp. 1027–1031. doi: 10.1109/AICCSA.2008.4493673.

[11] B. C. Pierce, *Types and Programming Languages*. The MIT Press, 2002.

[12] R. van Glabbeek, “Ensuring liveness properties of distributed systems: Open problems,” *Journal of Logical and Algebraic Methods in Programming*, vol. 109, p. 100480–100481, 2019, doi: https://doi.org/10.1016/j.jlamp.2019.100480.

[13] C. C4, “Ondo Finance. Findings and Analysis Report.” [Online]. Available: https://code4rena.com/reports/2023-09-ondo

[14] J. V. Behanan, “OWASP Smart Contract Top 10.” [Online]. Available: https://owasp.org/www-project-smart-contract-top-10/

[15] S. D. Angelis, F. Lombardi, G. Zanfino, L. Aniello, and V. Sassone, “Security and dependability analysis of blockchain systems in partially synchronous networks with Byzantine faults.” [Online]. Available: https://doi.org/10.1080/17445760.2023.2272777

[16] N. Atzei, M. Bartoletti, and T. Cimoli, “A Survey of Attacks on Ethereum Smart Contracts SoK,” in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, Berlin, Heidelberg: Springer-Verlag, 2017, pp. 164–186. doi: 10.1007/978-3-662-54455-6_8.

[17] C. C4, “Arcade.xyz. Findings and Analysis Report.” [Online]. Available: https://code4rena.com/reports/2023-07-arcade

[18] C. C4, “Centrifuge. Findings and Analysis Report.” [Online]. Available: https://code4rena.com/reports/2023-09-centrifuge

[19] E. Foundation, “Formal Verification of Smart Contracts.” [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts/formal-verification/

[20] A. Souri, A. M. Rahmani, N. J. Navimipour, and R. Rezaei, “A symbolic model checking approach in formal verification of distributed systems,” *Human-centric Computing and Information Sciences*, vol. 9, no. 1, p. 4–5, doi: 10.1186/s13673-019-0165-x.

[21] J. Harrison, J. Urban, and F. Wiedijk, “History of Interactive Theorem Proving,” in *Computational Logic*, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:30345151

[22] Hildenbrandt *et al.*, “KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine,” in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 204–217. doi: 10.1109/CSF.2018.00022.

[23] F. Cassez, J. Fuller, and H. M. A. Quiles, “Deductive Verification of Smart Contracts with Dafny.” 2022.

[24] S. So, M. Lee, J. Park, H. Lee, and H. Oh, “VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts.” 2019.

[25] Y. Wang *et al.*, “Formal Specification and Verification of Smart Contracts for Azure Blockchain.” 2019.

[26] B. Beckert, T. Bormer, F. Merz, and C. Sinz, “Integration of Bounded Model Checking and Deductive Verification,” in *Formal Verification of Object-Oriented Software*, B. Beckert, F. Damiani, and D. Gurov, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 86–104.

[27] M. Nguyen, D. Stoffel, M. Wedler, and W. Kunz, “Transition-by-transition FSM traversal for reachability analysis in bounded model checking,” in *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, 2005, pp. 1068–1075. doi: 10.1109/ICCAD.2005.1560219.

[28] P. Antonino and A. W. Roscoe, “Solidifier: Bounded Model Checking Solidity Using Lazy Contract Deployment and Precise Memory Modelling,” in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, in SAC '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 1788–1797. doi: 10.1145/3412841.3442051.

[29] K. R. M. Leino, “This is Boogie 2,” Jun. 2008.

[30] M. H. Majd Soud Gísli Hjálmtýsson, “Dissecting Smart Contract Languages: A Survey.” [Online]. Available: arxiv:2310.02799v2

[31] Chainsecurity, “Curve LP Oracle Manipulation: Post Mortem.” [Online]. Available: https://chainsecurity.com/curve-lp-oracle-manipulation-post-mortem/

[32] F. Schrans, D. Hails, A. Harkness, S. Drossopoulou, and S. Eisenbach, “Flint for Safer Smart Contracts.” 2019.

[33] J. Pettersson and R. Edström, “Safer smart contracts through type-driven development.” 2016.

[34] F. Maurica, D. R. Cok, and J. Signoles, “Runtime Assertion Checking and Static Verification: Collaborative Partners,” in *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2018, pp. 75–91.

[35] T. S. Hoang, “An Introduction to the Event-B Modelling Method,” 2013, pp. 211–236.

[36] A. Sorkin and P. Donovan, “LR(1) Parser Generation System: LR(1) Error Recovery, Oracles, and Generic Tokens.” 2012.

[37] F. Howar, D. Giannakopoulou, M. Mues, and J. A. Navas, "Generating Component Interfaces by Integrating Static and Symbolic Analysis, Learning, and Runtime Monitoring," in *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2018, pp. 120–136.

[38] J. Chen and S. Micali, "Algorand." 2017.

[39] J. McCarthy, "Towards a Mathematical Science of Computation," in *Program Verification: Fundamental Issues in Computer Science*, T. R. Colburn, J. H. Fetzer, and T. L. Rankin, Eds., Dordrecht: Springer Netherlands, 1993, pp. 35–56. doi: 10.1007/978-94-011-1793-7_2.