NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

JOÃO MIGUEL DE ALMEIDA AFONSO

Bachelor in Computer Science

MECHANISMS FOR MODELING AND VALIDATION OF SMART CONTRACTS

MASTER THESIS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

December, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

MECHANISMS FOR MODELING AND VALIDATION OF SMART CONTRACTS

MASTER THESIS

JOÃO MIGUEL DE ALMEIDA AFONSO

Bachelor in Computer Science

Adviser: António Ravara
Associate Professor, NOVA University Lisbon

Examination Committee

Chair: Carlos Damásio
Associate Professor, NOVA University Lisbon

Rapporteurs: Emilio Tuosto
Associate Professor, Gran Sasso Science Institute

António Ravara
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
December, 2023

Mechanisms for Modeling and Validation of Smart Contracts
MASTER THESIS

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.5) [20].

Acknowledgements

I want to begin by expressing my deep gratitude to my family. It was thanks to the incredible effort they made that I was able to continue my academic journey. Without their support and continuous encouragement, I wouldn't be here today, completing a significant part of my educational journey.

I'm also grateful to my friends who have been with me from the beginning, facing challenges together and offering constant support, while also experiencing stress and hardships together.

I would like to express my sincere gratitude to all those who have contributed to the success of this project. Your support, guidance, and hard work have been invaluable, and I am truly appreciative of your efforts.

Last but not least, I would like to express my deep gratitude to my advisor, Professor António Ravara, for his valuable support, guidance, and patience with me during this thesis.

Abstract

The characterization of a Smart Contract consists of describing agreements between multiple parties involved, which can be autonomously upheld without the need of a trusted intermediary. SCs are stored and executed on the Blockchain, a decentralized, digital ledger designed to log transactions in a secure and transparent manner. It operates across a network of computers, where each node retains a copy of the ledger and transactions are validated through consensus protocols.

One of the key features of SCs and blockchains is their immutability: once they have been deployed, the contracts and blocks of the blockchain can not be changed (although it is technically possible, it necessitates extreme measures such as hard forking or obtaining cooperation of a substantial portion of the network nodes in a blockchain).

However, vulnerabilities can still exist in the code of SCs, potentially resulting in exploitable bugs that may lead to the loss of valuable assets. To address these issues domain-specific languages (DSLs) are under development to offer safety assurances to programmers. However, none of these languages can offer absolute proof of contract correctness, leaving room for improvement in the development of SCs.

This thesis aims to contribute to the process of elaborating SCs by introducing a language designed to capture their behavior. In addition to this language, suitable tool mechanisms were also developed to perform static validations on the behavior of existing Daml SCs, ensuring alignment with the intended outcomes of the contracts and ultimately bolstering the auditability of SCs.

Resumo

A caracterização de um SC consiste em descrever acordos entre várias partes envolvidas, que podem ser mantidos autonomamente sem a necessidade de um intermediário de confiança. Os SCs são armazenados e executados na Blockchain, uma digital ledger descentralizada tendo como função registar transações de maneira segura e transparente. A blockchain opera numa rede de computadores, onde cada nó mantém uma própria cópia e as transações são validadas por protocolos de consenso.

Uma das principais características dos SCs e das blockchains é a sua imutabilidade: assim que levarem deploy, os contratos e os blocos da blockchain não podem ser alterados (tecnicamente é possível, mas para tal requere-se medidas extremas, como um fork na blockchain ou a obtenção da cooperação de uma parte substancial dos nós da rede de uma blockchain).

No entanto, vulnerabilidades ainda podem existir no código dos SCs, potencialmente resultando em falhas que podem levar à perda de valores. Para lidar com estes problemas, domain-specific languages (DSLs) estão em desenvolvimento para oferecer garantias de segurança aos programadores. No entanto, nenhuma dessas linguagens pode oferecer prova absoluta da correção do contrato, deixando em aberto vários caminhos melhorias no desenvolvimento de SCs.

Esta tese tem como objetivo contribuir para o processo de elaboração de SCs, introduzindo uma linguagem cuja finalidade é capturar o comportamento destes. Além dessa linguagem, também foram desenvolvidos mecanismos de ferramentas capazes de realizar validações estáticas sobre o comportamento dos SCs existentes, garantindo alinhamento com os efeitos pretendidos do contrato , reforçando assim a auditabilidade dos SCs.

Contents

List of Figures

List of Tables

Listings

Glossary

Smart Contract Programs stored on a blockchain that run when predetermined conditions are met. *(p. iii)*

1 Introduction

1.1 Context

Blockchains represent a relatively recent technological innovation that has seen widespread adoption in both the public and private sectors. With the surge in popularity driven by cryptocurrencies and smart contracts, not only have blockchains become more accessible but they have also become more susceptible to vulnerabilities.

SCs have significantly expanded the flexibility of transactions within the blockchain, a capability that was absent in its early days. Leveraging the immutability of the blockchain, a key attribute aiding in fraud prevention, and their autonomy from third-party validation, SCs offer distinct advantages. Once deployed, they remain impervious to modification, facilitating the automation of processes among involved parties to execute vital agreements. This not only saves time but also streamlines the costs for specific transactions. Hence, it becomes imperative to ensure the accuracy of these programs. Therefore, it is crucial to check if these programs are accurate. However, it is still a difficult undertaking in part because the programming languages used to developed them often are not as easy to comprehend, and can also contain a few bugs/vulnerabilities, as a few are still in their early years. As Edsger W. Dijkstra said:

> *Program testing can be used to show the presence of bugs, but never to show their absence!*

This task can be challenging, primarily because the programming languages used in their development can be complex and may contain occasional bugs or vulnerabilities, especially as some are still in their nascent stages.

1.2 Problem

As SCs handle valuable assets, ranging from simple fungible tokens like fiat currencies (e.g., the dollar) to non-fungible tokens (e.g., digital art), they involve interactions with multiple parties, being users or even other contracts, increasing the significance for proper

contract auditability. The immutability of deployed contracts underscores the need for thorough auditing, as immutable code can result in irreversible bugs leading to asset losses. A notable case in point is the DAO attack of 2016, which resulted in a theft of approximately 70 million USD due to the exploitation of vulnerabilities within a SC [30]. To make sure SC "do not go wrong", we aim to ensure:

- operations are called by the right participants at the correct time.
- valid logical restrictions and effects of SCs.
- allowed interactions between multiple SCs.

1.3 Goals

In certain scenarios, a contract's behavior can become unpredictable, giving rise to silent errors that only become apparent after deployment. Unfortunately, there may be no clear simple-to-use mechanisms for developers to proactively prevent such situations. Such that in this work, we aim to better support the development of SCs with fewer vulnerabilities. Concretely, equipping contracts with modelling mechanisms to express purpose and tools to validate their behaviour. To achieve that we look for an appropriate notion of automaton to capture a SC intended behaviour, we statically verify SCs actions with reference to their respective automaton ensuring that any logical condition, being either a constraint or an effect of any operation is valid and also generate correct-by-construction SC code from the automata.

1.4 Contributions

We separate our contributions into two categories: (i) general ones, where a person can express a SC using the presented language; (ii) more focused ones to Daml, a DSL that we enhance. In particular, our contributions are the following:

- An Appropriate notion of automata to support the modelling of interacting SCs, as well as their restrictions and effects.
- Use the notion proposed to statically verify Daml code. Generate Daml script code from an automaton.

2

Background

SCs are stored in Blockchains, these can be seen as a distributed system, where each node is a computational device running a specific consensus protocol in order to ensure consistency among them. This chapter introduces these concepts.

2.1 Distributed Systems

A Distributed System is composed of a collection of independent components, which can be either computers, processors, or processes, more commonly referred to as nodes of the system. To the user, they appear as a single coherent system. The nodes are interconnected through a network, where they achieve some sort of cooperation among themselves, (by communicating and coordinating their actions) through the use of messages, to achieve a common goal.

2.1.1 Cap Properties

The CAP theorem initiates the conversation regarding the various different trade-offs of a distributed system by stating that any distributed system can not simultaneously have three of its properties, (consistency, availability, and partition tolerance), and rather can only have two out of the three.

- Consistency is the guarantee that every node returns the same, most recent, successful write. It ultimately refers to every client having the same view of the data.
- Availability means that every non-failing node returns a response for all read and write requests in a reasonable amount of time.
- Partition Tolerance ensures that if a group of nodes is unable to communicate with other nodes due to a network failure, the distributed system continues to operate correctly;

But the approach of picking two out of the three properties is unfortunately misleading because network partitions are faulty, so they are not something we have a choice: they will happen.

At times when the network is working correctly, a system can provide both consistency (linearizability) and total availability. When a network fault occurs, you have to choose between either linearizability or total availability. Thus, a better way of phrasing CAP would be either *Consistent or Available when Partitioned* [17].

2.1.2 Distributed System Failures

Since a distributed system is composed of multiple nodes, the failures that can arise in those nodes are a concern. There are two main types of failures in a distributed system: Crash Faults and Byzantine Faults. [2.1]

When a node suffers from a crash fault, we can assume that it stopped functioning entirely (at least for the purposes of the protocol). In practice, a crash fault might be something as plain as a dead node or a node that loses its internet connection to other nodes in the network.

A Byzantine Fault refers to a situation where a node starts behaving abnormally, where it is still in communication with the rest of the nodes but sends faulty, purposeless, or even malicious data. Nodes can be honest, faulty, or malicious, and they have memory and a processor. A node that exhibits irrational behavior is also known as a Byzantine node named after the Byzantine Generals Problem.

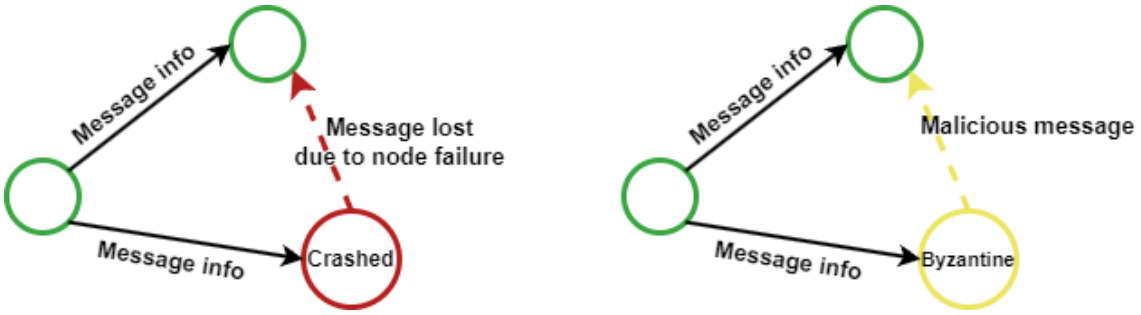


Figure 2.1: Type of Crashes: Fault and Byzantine.

A design goal in distributed systems is the construction of ways to automatically recover from failures without significantly impacting the performance of the system. In such ways that if failures occur, the distributed system should continue to operate while faults are present and when repairs are being made to address those faults. This feature is the fault tolerance of a distributed system.

There are two types of fault tolerance in distributed systems, one for each respective fault, Crash Fault Tolerance (CFT) and Byzantine Fault Tolerance (BFT). Comparing both types, CFT is much easier to achieve, meanwhile BFT is much harder to guarantee. Also to note, BFT consensus algorithms implicitly support CFT. To achieve CFT, replication is used. This is a common and widely used method to achieve it.

2.1.3 Consensus

When discussing consensus in a distributed system, we refer to the process of reaching an agreement among distrusting nodes on a final data state. To achieve consensus, consensus mechanisms are employed, which are a set of steps that are taken by all or most nodes in order to reach an agreement on a proposed value. The consensus between multiple nodes that are participating in a distributed system is referred to as distributed consensus.

These are various requirements that must be met to provide the desired results in a consensus mechanism:

- Agreement refers to the condition where all honest nodes decide on the same value.
- Termination means that all honest nodes terminate their execution and ultimately reach a decision.
- Validity dictates that the value agreed upon by all honest nodes must match the initial value proposed by at least one honest node.
- Fault-tolerant implies that the consensus algorithm should have the capability to operate even when faced with the presence of faulty or malicious nodes (Byzantine nodes).
- Integrity ensures that within a single consensus cycle, no node is allowed to make the decision more than once.

2.1.4 Types of Consensus Mechanism

There are two general categories of consensus mechanisms, which are as follows [7, 14]:

- Byzantine fault tolerance-based methods, without compute-intensive operations like partial hash inversion, rely on a straightforward scheme where nodes publish signed messages. While BFT performs well in scenarios with a limited number of nodes, it does not scale effectively.
- Leader-based consensus mechanisms entail nodes competing in a leader-election lottery, with the winning node being responsible for proposing the final value.

A few examples of Byzantine fault tolerance-based algorithms are as follows: Practical Byzantine Fault Tolerance, HotStuff, among several others. As for Leader-based consensus mechanisms, there is the Paxos algorithm, originally proposed by Leslie Lamport in 1989, and there is also Raft, introduced by Diego Ongaro in 2014 as a simpler version of Paxos, among many others [11].

2.2 Blockchain

A Blockchain is a ledger of blocks of information (e.g., a group of transactions or agreements) that are stored sequentially across a network of computers [15]. It is used to represent the data structure that stores a permanent history of transactions. Blockchain is a subset of Distributed Ledger Technology (DLT), a data structure that resides across multiple computing devices and is generally spread across regions. Its name derives from its primary activity, the recorded linking of individual blocks in a chain.

Since Blockchain is a decentralized system of peer-to-peer networks, it is highly available due to its decentralization. In these systems, the nodes in the network perform transactions in so-called blocks and then append these blocks to the Blockchain. To avoid conflicts, a block can only be added to the Blockchain if it has been validated by a consensus protocol, and once a block is added to the chain, it cannot be changed, as the Blockchain ledger is immutable. Immutability implies that the data cannot be changed. However, it is important to note that in blockchains, the data can, in theory, be changed, but with great difficulty, as it requires the cooperation of more than 50% of the network nodes in a Blockchain that uses Proof-of-Work as its consensus algorithm.

2.2.1 Block

A block in a blockchain functions as a container data structure that stores transactions. Individual blocks consist of several components, which can be roughly differentiated into the following fields [5]:

- The block header or the head of the block.
- The block body which is where the confirmed transactions are stored.
- The transaction counter that represents how many transactions are stored in the block.
- The block size, in bytes.

2.2.1.1 Block Header

The block header is composed of 3 sets of metadata [5]:

- The first one is a set that refers to the previous block, using the previous block's hash, which connects the current block with the previous one forming a chain between blocks.
- The second set includes the difficulty of the algorithm for this block, the timestamp of the block's creation, and the nonce. These values are related to the mining competition of the block.

- The last set includes the Merkle tree root of the block, which summarizes the transactions in the block.

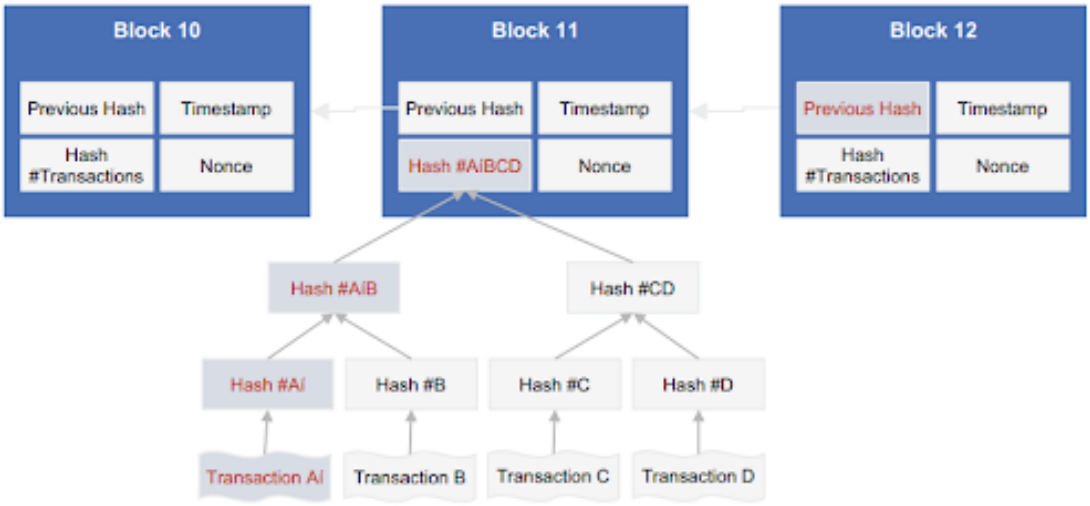


Figure 2.2: Blocks linked to each other.

2.2.1.2 Merkle Tree

Merkle trees, also known as binary hash trees, are tree-like data structures that work by capturing the hashes of individual data items in large datasets, thus increasing the efficiency of dataset verification. Within the Blockchain, Merkle trees function as a mechanism to prevent tampering, ensuring that no party can secretly alter any data.[2.2]

2.2.1.3 Types of Blocks

All the blocks in a Blockchain, are one of these three types [5]:

- Genesis Blocks, which are the first blocks ever created in a Blockchain, serve a crucial purpose in allowing network nodes to synchronize. Synchronization is only feasible when both nodes' databases have the same genesis block. This guarantees that the distributed transaction ledger on the Blockchain is the same for everyone, ensuring security.
- Valid blocks are all such blocks that have been mined and added to the Blockchain. These blocks constitute the confirmed and accepted transactions within the network.
- An Orphan Block is a block that has been solved within the Blockchain network but was not accepted by the network, often occurring when multiple miners simultaneously discover valid blocks, causing a temporary fork in the Blockchain. The fork is resolved when subsequent blocks are added and one of the chains becomes longer than the alternative(s).

2.2.2 Chain

The chain is a set of interconnected ordered blocks formed by the pointer to the hash of the previous block that each block has. Regarding the similarity to linked lists, the key difference is that in the Blockchain, the referenced block is the previous one, while in a linked list, the block refers to the next one.

When a new block is going to be added to a blockchain, a new hash is generated and recorded in the header of the last block as well as in the header of the new block, creating a chain of blocks also known as a cryptographic link of blocks. Old blocks cannot be modified, as any change in any block will invalidate all the subsequent blocks in the chain, since all their hashes will also change.

2.2.3 Gas

Gas is required to be paid for every operation, either a smart contract or transaction, performed on the Ethereum blockchain. This is a mechanism that ensures that infinite loops cannot cause the whole Blockchain to stall due to the Turing-complete nature of the EVM. A transaction fee is charged as some amount of Ether and is taken from the account balance of the transaction originator.

A fee is required for transactions to be included in mining. If the fee is too low, the transaction may never be picked up. The higher the fee, the greater the chances of miners including the transaction in a block. However, if a transaction with an appropriate fee is included in a block but has too many complex operations and the gas cost is insufficient, it can result in an out-of-gas exception. In such cases, the transaction will fail but still be included in the block, and the transaction originator will not receive a refund [7].

2.2.4 Types of Blockchain

All types of Blockchains can be categorized as permissionless, permissioned, or both. Permissionless Blockchains allow any user to pseudo-anonymously join the Blockchain network (that is, to become "nodes" of the network) and do not restrict the rights of the nodes on the Blockchain network. Permissioned Blockchains restrict access to the network to certain nodes and may also restrict the rights of those nodes on that network. The identities of the users in a permissioned Blockchain are known to other users of that same permissioned Blockchain. There are four main types of Blockchain structures.[29]:

- **Public blockchains** are permissionless, meaning anyone can join them, and they operate in a completely decentralized manner. Public Blockchains grant equal rights to all nodes within the Blockchain, allowing them to access the Blockchain, create new data blocks, and validate existing ones.

- **Private blockchains** which may also be referred to as managed Blockchains, are permissioned blockchains controlled by a single organization. In a private Blockchain,

the central authority determines who can be a node. The central authority also does not necessarily grant each node equal rights to perform functions. Private blockchains are only partially decentralized because public access to these blockchains is restricted.

- **Consortium blockchains** are permissioned and are governed by a group of organizations rather than a single entity, as in the case of private blockchains. Consortium Blockchains, therefore, enjoy more decentralization than private blockchains, resulting in higher levels of security.

- **Hybrid blockchains** are controlled by a single organization, but with a level of oversight performed by the public blockchain, which is required to perform certain transaction validations.

2.2.5 Consensus Protocols

This concept was discussed in a previous chapter. This section will now focus on consensus in the context of Blockchain technology. The following describes the two main categories of consensus mechanisms:

- Proof-based or leader-election lottery where a leader is elected at random (using an algorithm) and proposes a final value. This category is also referred to as the fully decentralized or permissionless type of consensus mechanism.

- BFT-based is a more traditional approach based on rounds of votes. This class of consensus is also known as the consortium or permissioned type of consensus mechanism.

2.2.5.1 Proof of Work

The proof of work (PoW) mechanism starts with the calculation of the hash value of the block header. The block header includes a nonce, which miners frequently modify to generate different hash values. Consequently, the consensus requires that the resulting value falls within a specific range. To maintain agreement across the network regarding the propagation of new blocks, PoW sets a complex problem/puzzle that must be solved by the cooperating nodes. Miners who successfully solve the problems earn the privilege to add a new block to the blockchain. The puzzle maintains an adjusted difficulty level and is tackled by estimating the nonce's value. This value is combined with the block's header information and fed into the SHA-256 hash function. The hash function processes all inputs to generate the resulting hash value. If the output of the hash function falls below a predetermined threshold, the proposed nonce is accepted, and the miner is granted permission to add the block to the Blockchain. Once a miner achieves the target value, it broadcasts the block throughout the network, prompting every node in the network to

verify the authenticity of the hash value and append the corresponding block to their own blockchain [18].

2.2.5.2 Proof of Stake

Proof of stake (PoS), which was initially proposed for Peercoin, has been used as an alternative to PoW to eliminate the excessive power consumption of nodes. Since the election of the block proposer based on the account balance seems unfair, many proposed solutions incorporate the stake size. Although PoS is energy efficient in comparison with PoW, it is not resilient to attacks. Accordingly, several blockchain solutions initially employ PoW and gradually transform to PoS [23, 18].

3

Smart contract languages

To address Smart Contracts, this chapter will focus on what a SC is exactly and explain why general-purpose languages are not typically recommended for developing such programs. To better understand SCs, I categorize Domain Specific Languages (DSLs) into two distinct categories: Domain-Specific Specification Languages and Domain-Specific Programming Languages. In each case, I assess several languages that serve as prime examples of the current state of the art.

3.1 What is a smart contract?

The concept of smart contracts was first proposed in the 1990s by Nick Szabo [28]. However, SCs only began to gain momentum when Blockchain technology emerged in 2009 because up to that point, there was no platform to execute SCs.

Unlike contracts in the real world, SCs are entirely digital and essentially containers of code that encode agreements. The term 'smart contracts' is not the clearest, and even the co-founder of Ethereum, Vitalik Buterin, has expressed regret about adopting it, with hindsight, he would have used something more technical and less exciting, such as "persistent scripts" [9].

A smart contract is self-executing code that carries out a set of instructions, which are then verified on the blockchain. SCs can establish trust among parties in a trustless contracting environment. The terms and conditions embedded in the contracts are automatically enforced when certain criteria are met. Compared with traditional contracts, SCs offer advantages such as reduced transaction risk, lower management and service costs, and improved business process efficiency, as they are typically deployed on and secured by Blockchain [31]. Even with the previous advantages, so as to have a safe contract, the need to prevent vulnerabilities from happening lies in correctly and accurately auditing the contracts since most of the time, we are dealing with assets within the contract. Vulnerabilities present in the smart contracts can lead to significant losses (loss of 31 Million $ from MonoX [21], due to the tokenIn and tokenOut being the same (it was missing the logic 'tokenIn != tokenOut')).

SCs provide significant benefits to various industries, including insurance, finance, and law, among others. Industries that traditionally rely on paper-based processes can leverage Blockchain for its immutability and further enhance efficiency through the automation offered by SCs.

3.2 General Languages

When it comes to choosing languages to write smart contracts, the options range from general-purpose programming languages to domain-specific programming languages. However, this choice is hardly a dilemma, as one outweighs the other in terms of advantages. Domain-specific languages are specifically designed for this type of application, while general-purpose languages are intended for a wide range of application domains. SCs require a high level of security to safeguard the valuable assets with which they deal. General-purpose languages do not offer the necessary tools to achieve this level of security, which discourages their use in building SCs.

In 2018, Vitalik Buterin, co-founder of the Ethereum Blockchain, shared his insights, highlighting that, unlike regular coding, smart contracts must focus on four essential properties [10]:

- Very small code size - making smart contracts less prone to bugs.
- Higher focus on safety - due to the fact that smart contracts receive, store, and transfer valuable user assets.
- Higher focus on auditability (misleading code is problematic) - code dictates what a participant can or cannot do, (principle of 'CODE IS LAW').
- Perfect determinism - The code in SCs must not exhibit unpredictable behavior.

For these reasons, the common choice should be to use domain-specific languages to implement SCs.

3.3 Domain-Specific Specification Languages

The term "Domain-Specific Specification Languages" as used in this section, refers to tools or languages that facilitate the implementation of SCs.

3.3.1 Scribble

Scribble is a language designed for specifying application-level protocols that govern interactions among communicating systems. These protocols define the agreed-upon rules and interactions between participating systems. The language has the potential for various uses, with its primary application being the validation and execution of a wide range of financial protocols and applications. In the context of this thesis, it is used as a

protocol between the different parties interacting with the contract and the actual contract itself.

It can be employed to specify protocols from both a global (neutral) perspective and the local perspective of a particular participant.

3.3.2 FSolidM

The FSolidM framework helps developers create smart contracts that are secure by design. Its features are as follows [22]:

- Formal Model - based on a simple, formal, finite-state machine (FSM) model designed to support Ethereum smart contracts.
- Graphical Editor - graphical editor that enables developers to design SCs as FSMs.
- Code Generator - a tool for translating a FSM into Solidity code.
- Plugins - extending the functionality of FSM-based SCs using plugins.

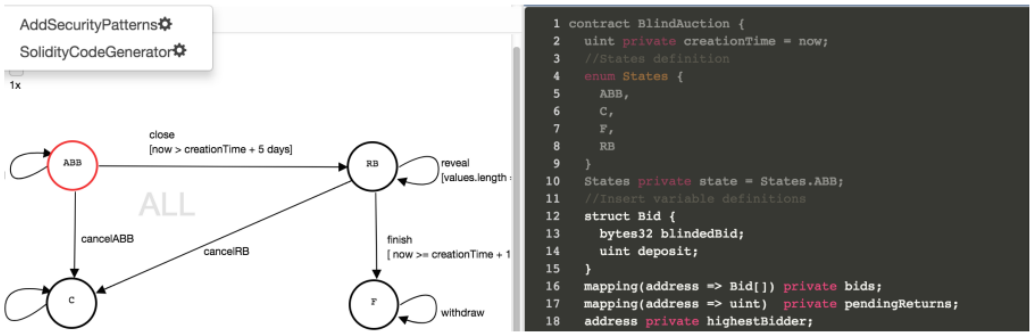


Figure 3.1: The FSolidM model and code editors. [22].

3.3.3 SmartScribble

SmartScribble is a protocol-based SC programming language that allows users to define protocols with expected interactions between users and the contract. Following the flow specified by the protocol, SmartScribble automatically generates a deterministic state machine that governs interaction behavior. The syntax is adapted from Scribble, a language used to describe application-level protocols among communicating systems, and it has been tailored to suit the blockchain environment.

SmartScribble uses a source protocol to generate Plutus code for the target language containing necessary boilerplate code for the smart contract and a designated module where the developer should add business logic [13]. Appendix A presents an example of the language.

3.3.4 Are these Domain-Specific Specification Languages not expressive?

In the researched DSSLs, all of them use a FSM approach but lack features that we are trying to secure for the contracts. For example, Scribble can extract the local type of each role that exists in the contract but it does not have access to assertions, failing to implement the logic of the contract. While FSolidM has logical assertions, it can not extract the local type of each role of a contract and also has no support for modelling inter-contract interactions. Lastly, SmartScribble has the following few disadvantages such as limited role validation, limited/restrictive individual state machines and ordered states (states are ordered by the implemented endpoints, it is a design decision but restrictive nonetheless). In the researched DSLSs, all of them use an FSM approach but lack features that we are trying to secure for the contracts. For example, Scribble can extract the local type of each role that exists in the contract, but it does not have access to assertions, failing to implement the logic of the contract. While FSolidM has logical assertions, it cannot extract the local type of each role in a contract and also has no support for modeling inter-contract interactions. Lastly, SmartScribble has several disadvantages, such as limited role validation, restrictive individual state machines, and ordered states (states are ordered by the implemented endpoints, it is a design decision but nonetheless restrictive).

3.4 Domain-Specific Programming Languages

In this next section, several representative languages will be presented, where context is given to each one, and they are shown in an order from a lower to a higher level language. To understand each of these languages, as well as their differences, I implemented several smart contracts in each respective language [2].

In the next section, several representative languages are introduced with the proper context for each one. They are analysed starting from lower-level languages and progressing to higher-level ones. To facilitate a better understanding of each language and their differences, example SCs are given for each respective language.

3.4.1 Scilla

Scilla is an innovative intermediate-level programming language for SCs on the Zilliqa blockchain. Designed with a strong focus on smart contract safety, Scilla offers standard memory and execution safety guarantees. Additionally, it employs a type checker to validate the legitimacy of inter-contract interactions.

Despite its minimalist nature, with the interpreter and type checker each comprising only a few dozen lines of OCaml code, Scilla remains expressive. While it lacks recursion, it has successfully implemented all categories of the most commonly deployed smart contracts [27].

3.4.2 Move

Move is an executable bytecode language used for implementing custom transactions and SCs. One of Move's key features is its ability to define custom resource types with semantics inspired by linear logic. In this system, a resource can never be copied or implicitly discarded; it can only be moved between program storage locations. These safety guarantees are statically enforced by Move's type system. Despite these unique protections, resources function as ordinary program values; they can be stored in data structures, passed as arguments to procedures, and more [8]. Move was originally developed by Facebook and tailored to serve as a secure and verified programming language.

3.4.3 Vyper

Vyper has been developed to provide a more user-friendly medium for writing SCs, with a focus on making contracts easier to understand.

Vyper aims to increase the difficulty for developers to intentionally write misleading or malicious code, and it also safeguards against developers unintentionally introducing vulnerabilities into their contract code. While Vyper is often compared to Solidity, it does not position itself as a replacement for Solidity. Its primary goals revolve around auditability, simplicity, and security. To achieve these objectives, Vyper intentionally omits various features and functionalities found in Solidity. This trade-off is made to enhance the human readability of code, especially for novice users. If a programmer requires any of the more complex features not supported by Vyper, they will need to use Solidity for writing the smart contract [16].

3.4.4 Plutus

Plutus is the native SC language for Cardano. It is a Turing-complete language written in Haskell, and Plutus SCs are essentially Haskell programs. When you use Plutus, you can have confidence in the correct execution of your SCs. It draws from modern language research to offer a secure, full-stack programming environment based on Haskell, which is the leading purely functional programming language [25].

3.4.5 Daml

DAML is a SC language specifically designed for distributed ledgers. It empowers users to safely, unambiguously, and at a high level, specify real-time business logic.

In DAML, contracts, agreements, and the involved parties are native constructs within the language. When using it, programmers define how contracts are created, which parties are authorized for this process, and which parties are granted rights within a contract.

The language allows for the direct expression of contracts, parties, rights, obligations, and authorization, freeing programmers to focus on refining their business logic rather

than the intricacies of encoding these concepts at a lower machine level. Furthermore, DAML's ability to express contracts directly enables the DAML system to automatically analyze the business logic, performing checks for issues related to rights, obligations, and authority [19].

3.4.6 Obsidian

Obsidian is a programming language that enables the development of smart contracts for the Ethereum Virtual Machine (EVM). Originally implemented for the Hyperledger Fabric Blockchain, the Obsidian language follows a state-oriented programming paradigm that allows for the explicit declaration and transition among states. Additionally, it incorporates linear types to prevent the abuse of resources managed by the smart contract. It's worth noting that the language is currently under development [24].

3.5 Comparison between DSPLs

Language	Turing Complete	Paradigm	Type of language	Syntax
Scilla	no	functional	interpreted	Ocaml
Move	yes	object-oriented	interpreted	Rust
Vyper	no	procedural	compiled	Python
Daml	yes	functional	interpreted	Ocaml
Obsidian	yes	object-oriented	compiled	C

Table 3.1: Comparison between Domain-Specific Programming Languages.

To facilitate a proper comparison of each language, an analysis of a SC example implemented in each language is used [2]. The goal is to highlight key differences between languages while also avoiding repetitive logic and code among them. For Plutus contracts, the use of SmartScribble was employed.

3.5.1 Illustrative example

The presented smart contract is a Simple Marketplace, which serves as an application outlining the workflow for a basic transaction between an owner and a buyer within a marketplace. The state transition diagram below illustrates the interactions among the states within this workflow.

Each party that interacts with the contract will have a role, which, in the contract above, is either an Owner (O) or a Buyer (B). The contract also has states, to restrict the actions that can occur, as each operation will be tied to a specific state.

In nearly every contract, there exists a variable to specify the current state of the contract. In this case, the available states are "Item Available", "Offer Placed", and "Accepted".

In most programming languages, an enumerator is used to store the possible states. However, there are exceptions, such as Obsidian, which allows for the declaration of the

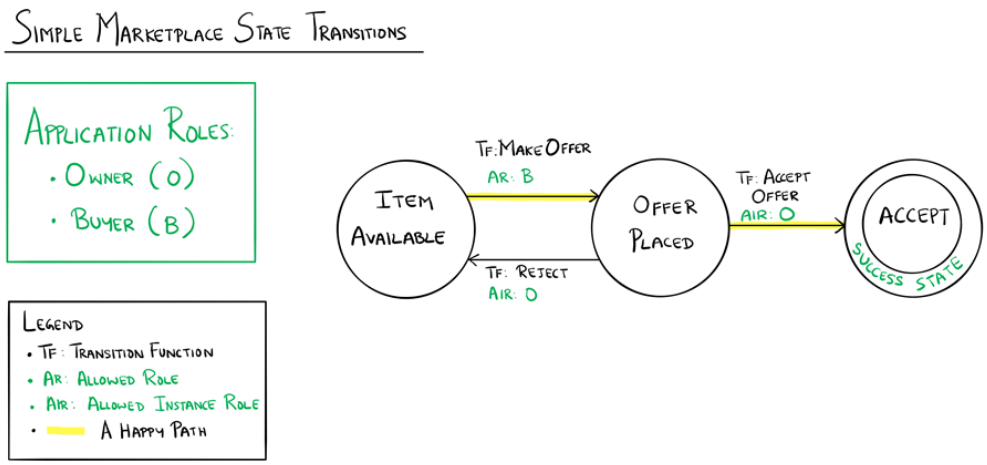


Figure 3.2: Simple Marketplace workflow [6].

origin and target states in each operation. Similarly, SmartScribble is different because it does not require specifying a state, as the language follows a sequential logic between operations. Lastly, Move is distinct as it does not offer an enumeration for states.

Every contract starts with the deployment of the contract itself using a constructor where the arguments of the contract are passed.

```
enum StateType:
    AVAILABLE
    PLACED
    ACCEPTED
..
state: public(StateType)
```

Listing 3.1: Vyper state enum and variable.

```
type StateType =
  | Available
  | Placed
  | Accepted
..
field state: StateType = Available
```

Listing 3.2: Scilla state enum and variable.

```
SimpleMarketPlace@Owned (string
    dsc, int price) {
        this.description = dsc;
        this.offerPrice = price;
        ->ItemAvailable;
    }
```

Listing 3.3: Obsidian constructor.

```
template SimpleMarketPlace
  with
    owner       : Party
    buyer       : Party
    priceTarget : Int
    offerPrice  : Int
    state       : StateType
```

Listing 3.4: Daml constructor.

Then, it revolves around how each language checks the restrictions of operations. Most languages employ the use of assertions (Move, Vyper, DAML), while others use pattern matching (Scilla) or regular old "if" conditions (SmartScribble). In the case of Obsidian, it's worth noting that switch cases can be applied to the state itself.

In Scilla, it attributes values after checking the necessary restrictions (if the sender can call that operation, if a value is too low, if the current state enables calling that operation).

```
transition reject_Offer()
 l_state <- state;
  match l_state with
  | Placed =>
   l_Owner <- i_Owner;
   is_owner = builtin eq l_Owner
    _sender;
   match is_owner with
    | False =>
     err = GeneralError;
     ThrowError err
    | True =>
     ..
     suc = GeneralSuc;
     CreateEventSuccess suc
    end
  | _ =>
   err = GeneralError;
   ThrowError err
  end
end
```

Listing 3.5: Scilla rejectOffer operation.

```
choice MakeOffer :
    SimpleMarketPlaceId
      with
        offerPriceParam  : Int
      controller buyer
      do
        assertMsg "Item is not
    available for an offer." (this.
    state == ItemAvailable)
        assertMsg "Offer too low." (
    offerPriceParam >= priceTarget)

        create SimpleMarketPlace with
            offerPrice =
    offerPriceParam
            state = OfferPlaced
            ..
```

Listing 3.6: Daml makeOffer operation.

The same applies to Vyper.

In Obsidian, the method itself can specify a contract's required current state and also the target states.

In Daml, each method has an actual role caller indicated as the controller of the operation. For example, the "makeOffer" method can only be called by the buyer party. If changes occur, a new instance of the contract (contract template) needs to be created, and the previous one archived.

In Move, since a key feature is the movement of valuable assets between storage locations, it is required the use a few keywords to perform several steps to move an asset. It starts with "borrow global," which we use to check if the target address has the stated data type. After checking, it is used "move to" to transfer resources between addresses.

```
 transaction makeOffer(
    SimpleMarketPlace@ItemAvailable
     >> ( OfferPlaced |
    ItemAvailable ) this, int
    offer)
```

Listing 3.7: Obsidian method.

```
case ItemAvailable {
 if (offer >= 0) {
  revert "Item is not Available";
 }
 else
 {
  this.offerPrice = offer;
  ->OfferPlaced;
 }
}
```

Listing 3.8: Obsidian target state return.

Looking at the array of languages that were presented, the Domain-Specific Programming Language that was chosen to improve its development process was Daml. This choice results because it is not as complex as Plutus or Move, in due part to its simple

syntax, and primarily because of its role-based access control. This is an essential part of how the modeling of participants is done in the following chapter.

4

Capture Behaviour

In this chapter I introduce the automata definition, to capture the behaviour of SCs. This addresses the concerns of the DSSLs in chapter 3. The automata was developed in conjunction with tools [3, 4] to support it, consisting in two parsers (one to generate the global type from the transitions declared by the user and the other to extract the projections from a valid global type automaton) in addition to a graphical tool to help visualize the generated automata.

4.1 Global type

The global type automaton [3] is used to convey the behaviour logic of the contract for every role present in it. The automaton has the following formal definition:

$$A = \langle S, P, Ac, s0, F, As \rangle$$

- `S` is a finite set of states. (made of two subsets, internal states and external states.)
- `P` is the participants involved in the automata. (contains participants and roles)
- `Ac` is the set of actions. (contains contracts, methods and inputs)
- `s0` $\in$ `S` is the initial state.
- `F` $\subset$ `S` is set of final states.
- `As` is the set of logical assertions.

A transition of the automaton consists of subsequent multiple sections:

```
iS preC participant action postC fS
```

So, the user expresses a transition by first specifying its initial state, (notably, the initial state of the transition may differ from that of the automaton). It then proceeds with identifying existing pre-conditions (assertions are expanded in their dedicated section).

After, it identifies the participants who can invoke the transition, details the action (also discussed with higher detail separately), and concludes with the identifying the post-conditions and discloses the resulting state of the transition.

The transition function, $\Delta : S, FA \rightarrow S', FA \subset As \times P \times Ac \times As$, maps a state and a full action (consists in `pre-condition` $\times$ `participant` $\times$ `action` $\times$ `post-condition`) to a new resulting state. The function abides by the following rules:

- **Initiating the automaton**, the contract requires starting with the built in deploy operation. Considering:

  `ac = ((>|<),`ϵ`, ., deploy(cId)), fa = (pre, p, ac, post)`

  `pre, post` $\in$ `As, p` $\in$ `P, ac` $\in$ `Ac`

  Δ`(s, fa) = s0`

- For **extra contract operations**, where `S = SI`$\times$`SE`:

  Δ`(s, fa) = sa, sa` $\in$ `SE` $\subset$ `S`

  The transition generates an intermediate state, where depending on the result of the external call:

  - Proceeds with an **OK method** to the resulting state.

    `ac = ((>), cId, ., OK()), fa = (pre, p, ac, post)`

    `pre, post` $\in$ `As, p` $\in$ `P, ac` $\in$ `Ac`

    Δ`(sa, fa) = s', s'` $\in$ `SI`

  - Or resets its state with the **NOK method**.

    `ac = ((>), cId, ., NOK()), fa = (pre, p, ac, post)`

    `pre, post` $\in$ `As, p` $\in$ `P, ac` $\in$ `Ac`

    Δ`(sa, fa) = s`

- For **intra contract operations**, where `S = SI`$\times$`SE`:

  Δ`(s, fa) = s', s'` $\in$ `SI` $\subset$ `S`

Before having the automaton, the user needs to express each transition contained in it.

```
68 # example of a contract
69 _ {} o:O > starts(c) { AND(true, true) } S0
70 S0 {} b:B > c.makeOffer(int:value) {} S1
71 S0 {} o > c.halt() {} S5+
72 S1 {} o > c.reject() {} S3
73 S1 {} o < c.accept() {} S2
74 S2 {} b > c.halt() {} S5
75 S3 {} b > c-leave() {} S5
76 S3 {} o > c.halt() {} S5
77 S3 {} b > c-stay() {} S4
78 S4 {} b|b1:B > c.makeOffer(int:value) {} S1
```

Listing 4.1: Input transitions of the automaton (Simple MaketPlace).

The transitions above generate an output with the structure shown in Appendix B.

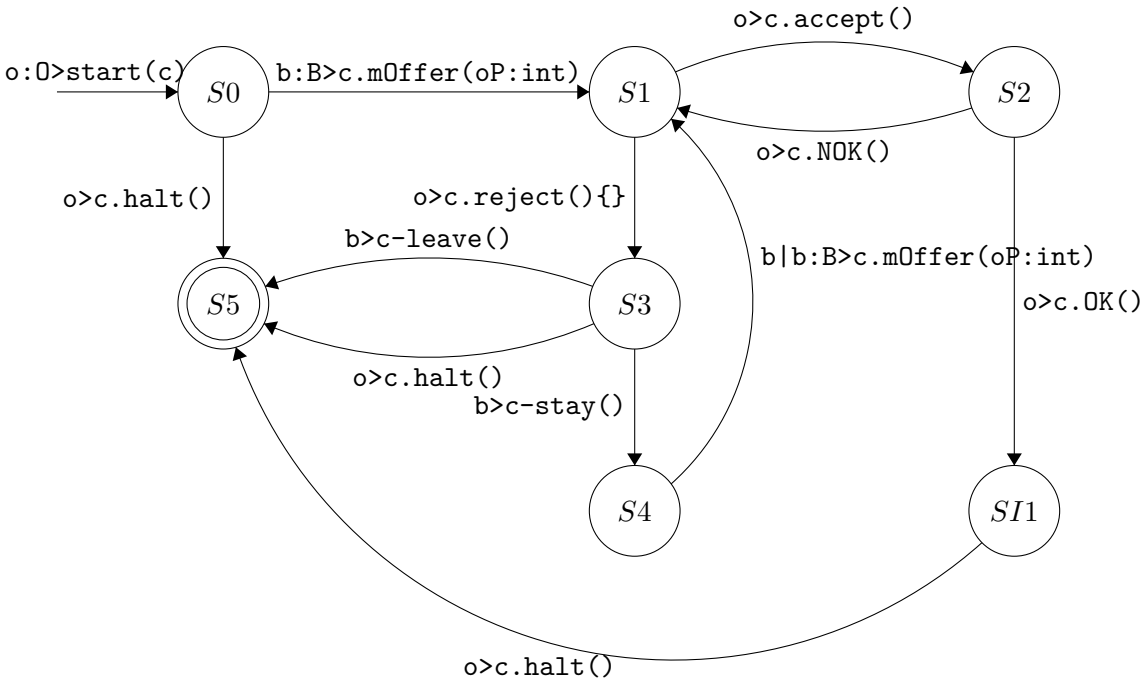


Figure 4.1: Global automaton for the Simple Marketplace [6].
Note: assertions are omitted to improve visibility.

4.2 Projections

From the global type we can also infer the behaviour of each role in the SC, resulting in a local type that expresses the behaviour of the SC that each specific role has in it (projection). In the projections, `the symbol ?` denotes an action that must be executed by a participant with a different role. `The symbol !` indicates an action that needs to be called by a participant with the same role as the projection. By obtaining the projection for each role in the contract, it facilitates the testing process as we can isolate and analyze

the behavior of a specific role. In particular, it is useful for run-time monitoring various participants involved in a SC.

To obtain the local type automaton of each role, the tool [4] parses the global type to gather all the required information for the projections. After obtaining the internal transitions that do not belong to the projected role, we use epsilon closure and minimize the automaton. Along with the other transitions, and by specifying the type of action for each transition using symbols (? and !), we obtain the local automaton for a specific role. This process generates the local type for each role in the global automaton.

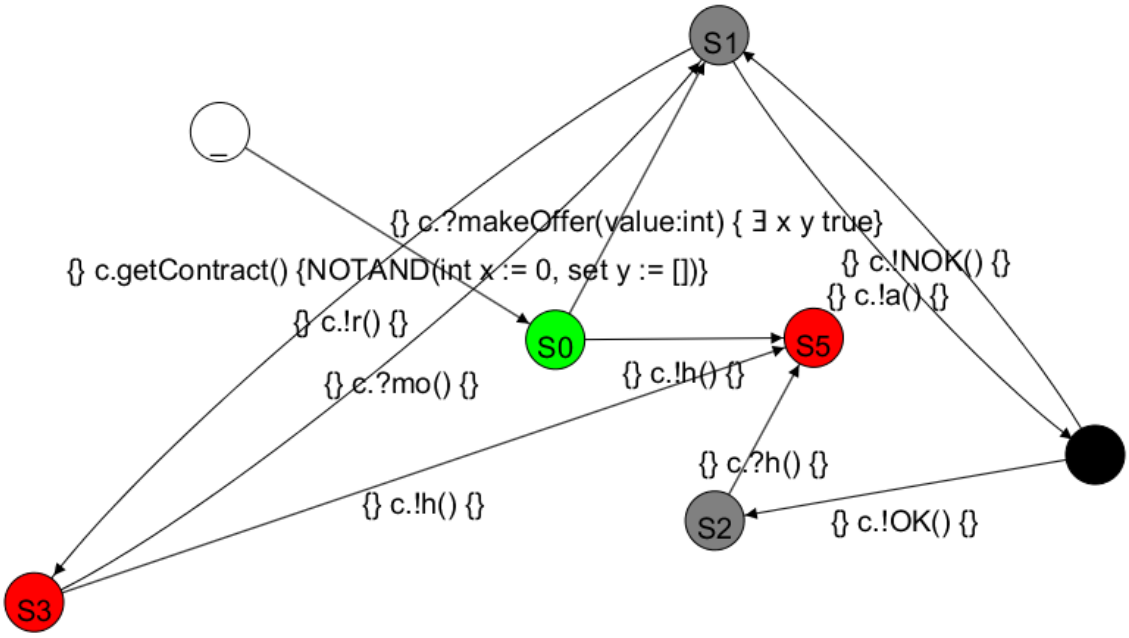


Figure 4.2: Example of the local type of role Owner of the SimpleMarketPlace.

4.3 Participants

A participant is an entity that interacts with the contract. They can initiate actions, access data, and participate in the contract's execution. Participants have their own unique identifier and roles associated with them. A participant can have multiple roles associated with it. They are defined as:

```
Role := r      Participant := b : Role || b || (b | b)
```

A participant can be one of three types:

- It can **a new participant** where the participant (identifier `b`) is a newly created entity that has an associated role (identifier `r`). New participants are registered into the contract.
- It can also be **an existing participant** (identifier `b`) meaning that the participant was previously registered into the contract.

- Or lastly, it can be **multiple participants** these being either newly ones or already existing ones.

4.4 Action

The action constitutes a specific operation or function that can be performed within a contract. It describes a behavior or task that participants can initiate, such as modifying the contract's state, interacting with other participants, or triggering events. They are defined by the following statements:

```
Action := > c . method | > c - method | < c . method
```

An actions comprises in a contract identifier, to know which contract the transition interacts with, and a method of the contract stated. Besides those two components the action can be one of three types:

- A **normal call**, where the action uses a greater than sign (>) before the contract id to indicate that the transition is intra-contract related, meaning that the action only involves that specific contract; After the contract id, a dot (.) is used to denote that the action is observable by all registered participants.

- An **internal call**, where the action starts with a greater than sign, (indicating intra-contract logic), followed by the contract id and then a dash sign (-) to signify that the transition is specific to the roles of the participants stated in the transition.

- An **external call**, where a less than sign (<) is used to signify that the transition has extra-contract behaviour which represents an interaction with a different contract.

All of the types of actions stated above end with a method, comprised of a method label (name of the operation of the contract) and the input parameters of the method. For example the following action, `> c1 . makeOffer(int: value, string: comment)`, indicates that the transition has only intra-contract behaviour where the method `makeOffer`, (this method can be observed by every participant registered in the contract), has two parameters, (value of type int and comment of type string).

There also exists two built-in methods:

- `start` - this method creates a new instance of a contract, (it requires the contract id of the contract to be instantiated as a parameter). It initializes the initial state of the contract by assigning variables, which are declared in the post Condition of the transition, and roles their proper values. The start method can have either intra or extra-contract behaviour, but it can not be an internal action. Example of the start method: `> start(c1)`, `c1` being the contract id.

- `halt` - this method freezes the contract, indicating that it has reached a state where it can no longer be modified or executed. This typically happens when the contract has fulfilled its purpose or when certain conditions are met to stop its execution.

4.5 Assertions

In this section of the language, we take cues from Hoare logic, a formal system used for reasoning about the correctness of computer programs, to base assertions on describing the pre-conditions and post-conditions of program statements. Hoare triples are used to express assertions about program correctness. A Hoare triple is of the form P C Q, where:

- P is the pre-condition (a statement that holds true before executing the program).
- C is the operation/method.
- Q is the post-condition (a statement that should hold true after executing the program if P is true before execution)

The goal of Hoare logic is to provide a formal way to prove or verify that programs meet their intended specifications. Both pre-conditions and post-conditions require a valid boolean outcome of the first-order logic expression provided. The pre-conditions do not allow for the declaration of variables or changes to contract variables. While the post-conditions do not have such constraints. Also of mention, is the deploy method (first action of the contract) which does not take any pre-conditions and is the only transition where variables can be declared (declaring contract variables).

The valid logical expressions are designed as:

LogicalExpression LE	:=	*BE*	
	|	*NOT BE*	Negation
	|	*AND BE BE*	Logical AND
	|	*OR BE BE*	Logical OR
BooleanExpression BE	:=	*AE == AE*	Equal
	|	*AE != AE*	Different
	|	*AE >= AE*	Greater than or equal
	|	*AE > AE*	Greater than
	|	*AE <= AE*	Lesser than or equal
	|	*AE < AE*	Lesser than
	|	*id := AE*	Assign Value to Variable
	|	*t id := AE*	Variable Declaration
	|	*(i | b | s) in set*	Contains
	|	*Exists (i | b | s) set (LE)*	Existential Quantifier
	|	*Forall (i | b | s) set (LE)*	Universal Quantifier

Figure 4.3: Assertion language.

ArithmeticExpression AE	:=	*NE*	
	|	*NE + NE*	Sum
	|	*NE - NE*	Subtraction
	|	*NE * NE*	Times
	|	*NE / NE*	Subtraction
NegationExpression NE	:=	*UE*	
	|	*not UE*	Negation for boolean values
UnaryExpression UE	:=	*true*	
	|	*false*	
	|	*[]*	
	|	*"s"*	
	|	*i*	
	|	*id*	
	|	*t id*	
Type t	:=	*int i*	Integer
	|	*bool b*	Boolean
	|	*string s*	String
	|	*set*	Set

Figure 4.4: Assertion language.

As for the parameters of methods, they can be indicated using the `type t` (example, (int: value, string: comment)) and to indicate a participant as a parameter, it is enough to state the identifier for the role and participant (for example, (Buyer: newP) where Buyer would be the role and newP the participant identifier).

4.6 Verifications

To generate a valid output, the tool needs to first do several checks to make sure that the transitions stated in the input produce an output. From syntactic analysis to more focused ones, like:

- Having unique participant, variable ids and parameter ids to not cause unnecessary errors/confusion when referencing each one.
- Confirm that the first action of the contract was its deployment, as the operations of a contract depend on the contract being deployed.
- Assures that the deployment method does not have preconditions, this is so that there are no restrictions to deploying a contract.
- Contract variables, or global variables, can only be declared in the post-condition of the deploy method, as it mimics the behaviour of a normal contract. Currently,

only global variables are available, so the declaration of a variable outside of the deployment will throw an error.

- A participant always needs to first be registered in the contract, (existing participants need to first be a new participant), to interact with it. We achieve this verification using the Depth-first search algorithm, which is an algorithm for traversing graph data structures. By starting from the initial state and analysing all paths from it to each state, we determine if it exists at least one path where the participant was previously registered in the contract. If no such path exists it throws an error informing the user of such.

- Since we are using an FSM approach, each state needs to have at least one valid path from the starting point to itself and also a valid path to an end state. Like the previous one, we also use the Depth-first search algorithm in this situation, since it provides use with paths from one node to another. So using the algorithm, we ensure that exists at least one path from the starting state to every other state in addition to a path from every node to at least one end state.

4.7 Visualization

To facilitate the user, the user experience, a graphical tool was also developed in order to view the automata being built. Works for both global types and projections since the format of both is the same.

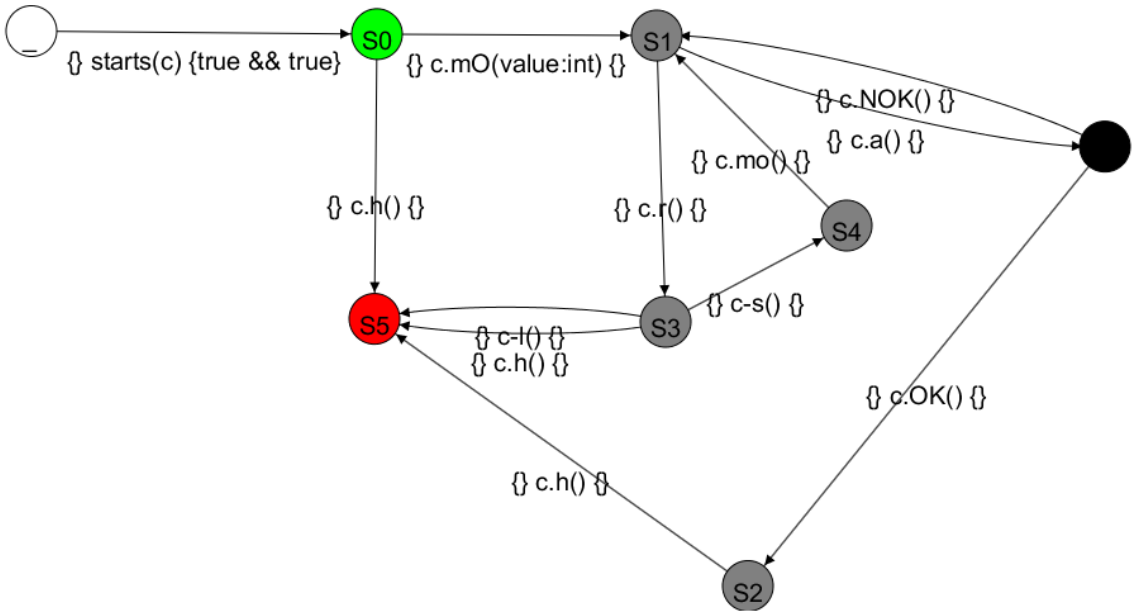


Figure 4.5: Graphical tool to preview the generated automaton.

5

Validation

For this section, we cover the regenerate tool, the adaptations made for it to work on SCs, the validations of existing contracts done against our automata, finding valid values for parameters from SC and lastly generating script code to be tested in Daml.

5.1 Regenerate

Regenerate [26] is a powerful and innovative tool designed to facilitate the generation of extended regular expressions. Regular expressions, commonly referred to as regex, are a fundamental part of text processing and pattern matching in computer science and data analysis. They are used to search, validate, and manipulate text by defining patterns of characters.

Regenerate takes regular expressions to the next level by offering an extended set of features and capabilities. This tool empowers users to create complex and highly specific regex patterns with ease. Here's a breakdown of what makes Regenerate a valuable resource:

- It provides an intuitive and user-friendly interface that simplifies the process of crafting regular expressions.
- It offers a unique feature that allows users to visualize their regex patterns in a clear and comprehensible manner. This visual representation helps users understand and refine their patterns more effectively. Generates regular expressions for various use cases, such as matching specific data formats or extracting information from text.
- Allows for the generation of both positive and negative examples for the language of the provided regex.
- Users can extend the functionality of Regenerate by incorporating custom regex libraries or plugins, allowing for even greater flexibility and adaptability.

In short, Regenerate is a versatile and user-friendly tool that enables users to create, optimize, and validate complex regular expressions. In the context of this work, we adapt

this tool to provide us with the possible sequences that a contract, either an existing Daml contract or one from our language, can have.

5.2 Adaptations

The original Regenerate tool only supported regular expressions composed of single characters, a situation which would greatly limit the tool for our use since contract method identifiers are not generally composed of a single character. An approach would be to map a character, for example, the first character of the string, to the string itself (e.g., `operation` $\rightarrow o, $`operation`). This would ultimately be restrictive since we only have a limited number of characters meaning at some point there would be no characters left to use. So instead, the approach followed was to use strings themselves.

To enforce a string the regular expression needs to follow the sequence of the string every time one appears. The problem lies in the use of special characters, such as repetition for example: where we want the cycle of `operation+` to apply to the whole string and not just to the previous character of the repetition symbol. To get around this constraint we make use of the language of the tool itself.

To impose this condition, we use a "fake string" by making it mandatory to apply the sequence of the string every single time. This is done by replacing the string used in the regular expression with an encapsulated version of it, where the string is surrounded by both left and right parenthesis, (e.g., `operation+` $\rightarrow$ `(operation)+`). Another change done to the tool was to restrict the cycle of repeater symbols, (+ and *), where instead of having an infinite number of times where the cycle can repeat itself it is capped at a maximum of two times. So the repeater symbols changes are the following:

$$Original : a^* \; is \; a\{0,\} \rightarrow Revised : a^* is \; a\{0,2\}$$

$$Original : a+ \; is \; a\{1,\} \rightarrow Revised : a+ \; is \; a\{1,2\}$$

Besides these previous changes, which are the main ones, the user also has the possibility to limit the number of positive examples for the language of the provided regular expression. By default, the program will give every valid path that could be executed on the contract.

5.3 Adapted regenerate

So in respect to this work, we adapted the original regenerate [1] for it to take two contracts and analyze them. One is an existing contract, with a few specific writing prerequisites. The mandatory pre-requisites are the following:

- The contract needs to have a datatype of type "state" which identifies which states are possible within the contract.

- As a follow-up, it also needs to have a "state" contract variable. As we follow a FSM approach having a specific variable to outline the state of the contract, facilitates the identification of the current state of the contract and it also simplifies the recognition of required operation states.

- For each operation:

  - It needs to check what is the state that the contract must have in order to be able to call it.
  - And also requires, stating the resulting state at the end of the method.

- For last, the file can only have one template/contract declared.

So we take an existing contract that is able to satisfy these conditions and an automaton output that expresses the behaviour of the existing contract, courtesy of the tool in the previous chapter.

By generating both of these contracts regular expressions we are able to ascertain if they are both equivalent. Using each regular expression, gives the possible positive paths that the automaton can have. Since these sequences are finite due to both repeater symbols being capped at a maximum of two cycles per sequence, (it can have nested cycles, but ultimately each loop has a max), by comparing both results we can extract if the contracts are similar to each other.

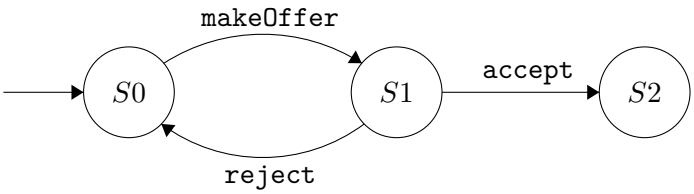
```
A = existing_contract output and
```

We generate the following regular expression:

```
B = automaton_contract output
```

If A = B then they are both equivalent. Leaving aside assertions both of the contracts can generate the same paths some of the transitions of the contract will be the same. To make sure the effects of the transition are also the same the assertions need to be evaluated and compared. If one is a subset of the other, $A \subset B1$ or $B \subset A2$, we can determine that one of them is more expressive than the other. Being that the goal is to make sure that the behaviour of existing contracts is the same as the ones we obtain from the tool, we can revise the automaton for the second situation or make a new Daml contract in order for the behaviour of the existing contract (in this case it would be a new one) to align with the automaton. In the off chance that the contracts aren't a subset of each other, it requires further analysis of the possible paths of each in order to understand where they diverge from each other.

For the following contract:



We generate this regular expression:

((ϵ)(makeOffer))((reject)(makeOffer)){0,2}(accept)

Where the generated possible positive paths are the following:

- ϵ makeOffer accept
- ϵ makeOffer reject makeOffer accept
- ϵ makeOffer reject makeOffer reject makeOffer accept

Why does the adapted regenerate tool only give us possible positive sequences and not positive sequences? In order to confirm that a sequence is positive we also need to make sure that the assertions that the contract has allow for valid solutions.

5.3.1 Assert

So at this step, we already have the regular expression of the contracts and also the possible paths that the contract can follow. At this point we need to make sure that at every step of a possible viable path, exists a solution for the variables used in that step, be either a contract variable being used in the pre-conditions or a parameter of the method of the current step. In order to figure out these values, it is employed the use of an SMT, or Satisfiability Modulo Theory, which is a powerful framework in computer science and mathematics used for solving complex decisions and optimization problems. A SMT combines elements of both propositional logic (SAT, or Boolean satisfiability) and first-order logic while accommodating specific theories related to various domains. The SMT used is the Z3v[12].

The only logic that is analysed in the assertions is arithmetic logic seeing that, it is equal for both Daml and Z3 and this is the first iteration of the tool.

Providing every expression to the solver, being pre-conditions identified by asserts in Daml, method parameters, contract variables, updated values to variables (post-conditions), the solver either tells us that the set of expressions is satisfiable or not thus throws an error. If they are satisfiable we can retrieve the valid values and use them to generate correct by construction Daml scripts.

If both contracts have satisfiable models, we can then compare them in regard to their assertions. In this aspect, there is no room for interpretation, either the assertions are equal so that the behaviour corresponds to their counterpart, or not leaving room to rewrite the automaton contract.

5.3.2 Generating valid scripts

At this point we have everything we need, where the contract has already been checked for its general behaviour, during the regenerate phase of the project, comparing the possible positive sequences. And in addition, it also checked its more specific behaviour, by comparing its assertions with its automaton counterpart. So it ends by generating valid script files, for the Daml contract, that can be used in the Daml testing environment.

```
module Main1 where
import Daml.Script
import Hello
setup = script do
  alice <- allocateParty"Alice"
  local <- submit alice do
    createCmd  HelloBlockChain
    with
                owner = alice
                state = S0
  local <- submit alice do
    exerciseCmd local MakeOffer
  submit alice do
    exerciseCmd local Accept
```

Listing 5.1: Scilla rejectOffer operation.

```
module Main2 where
import Daml.Script
import Hello
setup = script do
  alice <- allocateParty"Alice"
  local <- submit alice do
    createCmd  HelloBlockChain  with
        owner = alice
        state = S0
    local <- submit alice do
        exerciseCmd local MakeOffer
    local <- submit alice do
        exerciseCmd local Reject
    local <- submit alice do
        exerciseCmd local MakeOffer
    local <- submit alice do
        exerciseCmd local Reject
    submit alice do
        exerciseCmd local Accept
```

Listing 5.2: Daml makeOffer operation.

6

Conclusion and future work

In conclusion, this thesis has centered on the modeling of smart contract behavior, employing a Finite State Machine (FSM) approach to establish an automata notion capable of capturing state transitions within contracts. Our proposed approach serves the comprehensive objective of improving the development of smart contracts. Discerning essential aspects during contract development, equips developers with the means to provide agreed-upon outputs based on a given set of actions.

As an initial implementation, our approach adheres to a role-base procedure, requiring, either previous or current identification of the participant's role when invoking an action. While this initial implementation is a promising starting point, it holds potential for refinement and simplification in the declaration of input transitions.

Our tool offers the capability to specify that a user with a specific role can invoke methods associated with that role. The projections showcase the distinct behavior associated with each role.

By incorporating assertions, our tool excels at replicating or constructing contract behaviors observed in real contracts. It can constrain transitions with pre-conditions or alter contract variable states with post-conditions. Furthermore, by adapting the regenerate tool, we can achieve static verification of contracts alongside their respective automata. When combined with valid assertions, this approach facilitates a comparative analysis of the behaviors exhibited by different contracts. In instances where assertions are satisfiable, our tool is capable of generating Daml scripts suitable for the Daml testing environment.

In summary, the work produced not only contributes to the modeling and validation of smart contracts but also provides a practical toolset for developers to enhance the reliability and efficiency of contract implementation and verification processes.

For future research, there are several promising avenues for exploration in regard the enhancement of the developed tools:

- Development of the automaton language - the ongoing refinement and simplification of the automaton language constitute an iterative process that remains open for

continual enhancement. As the language evolves and matures, it necessitates concurrent updates to the associated parser tool to ensure seamless compatibility and functionality. This dynamic interplay between language development and parser tool updates underscores the perpetual nature of this research pursuit.

- Improvement and expansion to the assertion language - the enhancement and expansion of the assertion language is paramount in order to be able to cover more possible contract behaviours. As an example the addition of functions. It ultimately reinforces the sophistication and utility of our toolset.
- Direct communication with different smart contracts - The introduction of the external call feature, enabling the simulation of interactions between different contracts, indicates a noteworthy progression in our approach. Looking ahead, the potential for enhancing this capability lies in transitioning from abstract interactions to comprehensive contract interaction implementations.
- Evaluation assertions of the parser itself.
- Z3 covers more expressions on regenerate - The adapted regenerate tool has enabled contract analysis, but its current limitation lies in the range of accepted expressions. To enhance the tool's capabilities, an expansion of the accepted expression types is necessary.
- Generation of correct by construction code from the given automaton.

Bibliography

[1] J. Afonso. *Adapted Regenerate*. URL: https://github.com/jotaAfonso/generate_sc_behaviour (cit. on p. 29).

[2] J. Afonso. *Implemented Smart Contracts*. URL: https://github.com/jotaAfonso/Smart_Contracts (cit. on pp. 14, 16).

[3] J. Afonso. *Parser for the Automata*. URL: https://github.com/jotaAfonso/Parser (cit. on p. 20).

[4] J. Afonso. *Projections*. URL: https://github.com/jotaAfonso/OcamlProjections (cit. on pp. 20, 23).

[5] A. M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. 1st. O'Reilly Media, Inc., 2014. ISBN: 1449374042 (cit. on pp. 6, 7).

[6] Azure-Samples. 2018. URL: https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples/simple-marketplace (cit. on pp. 17, 22).

[7] I. Bashir. *Mastering Blockchain*. Packt Publishing, 2017. ISBN: 978-1-78883-904-4 (cit. on pp. 5, 8).

[8] S. Blackshear et al. "Move: A language with programmable resources". In: *Libra Assoc* (2019), p. 1 (cit. on p. 15).

[9] V. Buterin. URL: https://twitter.com/VitalikButerin/status/1051160932699770882?ref_src=twsrc%5C%5Etfw%5C%7Ctwcamp%5C%5Etweetembed%5C%7Ctwterm%5C%5E1051160932699770882%5C%7Ctwgr%5C%5E43d0189293e62c84176c8b0535b3526e127cbe13%5C%7Ctwcon%5C%5Es1_c10&ref_url=https%5C%3A%5C%2F%5C%2Fbitcoinist.com%5C%2Fvitalik-buterin-ethereum-regret-smart-contracts%5C%2F (cit. on p. 11).

[10] V. Buterin. URL: https://twitter.com/VitalikButerin/status/1024265820849860609 (cit. on p. 12).

[11] D. Callaghan. *Consensus Algorithms in Blockchain and Beyond*. 2018. URL: https://blogs.perficient.com/2018/05/04/consensuswithblockchain1/ (cit. on p. 5).

[12] L. De Moura and N. Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340 (cit. on p. 31).

[13] A. N. R. F. Falcão. "A protocol language for smart contracts". PhD thesis. 2021 (cit. on p. 13).

[14] hedera.com. *Understanding leader-based consensus algorithms*. URL: https://hedera.com/learning/consensus-algorithms/understanding-leader-based-consensus-algorithms (cit. on p. 5).

[15] D. Hellwig, G. Karlic, and A. Huchzermeier. *Build Your Own Blockchain: A Practical Guide to Distributed Ledger Technology*. 2020-01. ISBN: 978-3-030-40141-2. DOI: 10.1007/978-3-030-40142-9 (cit. on p. 6).

[16] M. Kaleem, A. Mavridou, and A. Laszka. "Vyper: A security comparison with solidity based on common vulnerabilities". In: *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE. 2020, pp. 107–111 (cit. on p. 15).

[17] M. Kleppmann. *Designing Data-Intensive Applications*. Beijing: O'Reilly, 2017. ISBN: 978-1-4493-7332-0. URL: https://www.safaribooksonline.com/library/view/designing-data-intensive-applications/9781491903063/ (cit. on p. 4).

[18] B. Lashkari and P. Musilek. "A Comprehensive Review of Blockchain Consensus Mechanisms". In: *IEEE Access* 9 (2021), pp. 43620–43652. DOI: 10.1109/ACCESS.2021.3065880 (cit. on p. 10).

[19] B. Lippmeier and E. Kmett. *A new language for a new paradigm: smart contracts.* 2018. URL: https://medium.com/daml-driven/a-new-language-for-a-new-paradigm-smart-contracts-648cc30294ad (cit. on p. 16).

[20] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[21] J. Lyanchev. "Over $30 Million in WETH and MATIC Stolen From DeFi Project MonoX". In: *cryptopotato* (2021). URL: https://cryptopotato.com/over-30-million-in-weth-and-matic-stolen-from-defi-project-monox/ (cit. on p. 11).

[22] A. Mavridou and A. Laszka. "Tool demonstration: FSolidM for designing secure Ethereum smart contracts". In: *Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7*. Springer. 2018, pp. 270–277 (cit. on p. 13).

[23] C. T. Nguyen et al. "Proof-of-Stake Consensus Mechanisms for Future Blockchain Networks: Fundamentals, Applications and Opportunities". In: *IEEE Access* 7 (2019), pp. 85727–85745. DOI: 10.1109/ACCESS.2019.2925010 (cit. on p. 10).

[24] obsidian. URL: https://obsidian.readthedocs.io/en/latest/ (cit. on p. 16).

[25] Plutus. URL: https://docs.cardano.org/plutus/learn-about-plutus (cit. on p. 15).

[26] G. Radanne and P. Thiemann. "Regenerate: a language generator for extended regular expressions". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 2018, pp. 202–214 (cit. on p. 28).

[27] scilla. URL: https://scilla.readthedocs.io/en/latest/ (cit. on p. 14).

[28] N. Szabo. "The idea of smart contracts". In: *Nick Szabo's papers and concise tutorials* 6.1 (1997), p. 199 (cit. on p. 11).

[29] K. E. Wegrzyn and E. Wang. "Types of blockchain: Public, private, or something in between". In: *Foley & Lardner* (2021) (cit. on p. 8).

[30] "What Was The DAO?" In: *gemini.com* (2022). URL: https://www.gemini.com/cryptopedia/the-dao-hack-makerdao (cit. on p. 2).

[31] Y. Xu, H.-Y. Chong, and M. Chi. "A review of smart contracts applications in various industries: a procurement perspective". In: *Advances in Civil Engineering* 2021 (2021), pp. 1–25 (cit. on p. 11).

A

Appendix A

A.1 SmartScribble protocol

```
protocol Auction (signed role Seller, role Buyer) {
  field currBid:Value,lastBidder:PubKeyHash,asset:Integer;

  initialise (initBid:Value, assetToSell:Integer) from Seller[|
    returnOutputOk $ setCurrBid initBid
                   $ setLastBidder noKey
                   $ setAsset assetToSell output|];

  rec Loop {
    choice at Seller {
      continueAuction [|returnOk|]: {
        bid (newBid:Value) from Buyer[|
          pkh <- ownPaymentPubKeyHash
          if (newBid `V.gt` mempty) && (newBid `V.gt` currBid)
          then returnOutputOk $ if (lastBidder == noKey)
            then setCurrBid newBid
              $ setLastBidder pkh
              $ setStateVal newBid output
            else setCurrBid newBid
              $ setLastBidder pkh
              $ setStateVal newBid
              $ setConstraint (Constraints.mustPayToPubKey lastBidder
    currBid) output
          else returnError "Invalid Bid!" |];
        Loop;
      }
      closeAuction [|returnOk|]: {
        collectFundsAndGiveRewards () from Seller[|
          returnOutputOk $ setConstraint (Constraints.mustPayToPubKey (head
     sellerId) currBid)
                         $ setStateVal mempty output |];
      }
    }
  }

  checkWinner () from Seller;
}
```

A.2 SmartScribble generated output (State machine)

```
146 # State Machine transactions and client definition #
147 transition :: State AuctionState -> AuctionInput -> Maybe (TxConstraints
        Void Void, State AuctionState)
148 transition State{stateData=oldData,stateValue} input = case (oldData, input
        ) of
149     (None _ _, InitialiseInput ... ) -> Just(mempty, State{stateData =
        InitialiseState ... })
150     (CloseAuctionState _ _ _ _ _, CheckWinnerInput ... ) -> Just(mempty,
        State{stateData = CheckWinnerState ... })
151     (ContinueAuctionState _ _ _ _ _, BidInput ... ) -> Just(mempty, State{
        stateData = InitialiseState ... })
152     (InitialiseState _ _ _ _ _, ContinueAuctionInput ... ) -> Just(mempty,
        State{stateData = ContinueAuctionState ... })
153     (InitialiseState _ _ _ _ _, CloseAuctionInput ... ) -> Just(mempty,
        State{stateData = CloseAuctionState ... })
154
155     _ -> Nothing
```

B

Appendix B

B.1 Generated automaton output.

```
{
  "id" : "contract identifier",
  "initialState" : "S0",
  "states" : [ "S3", "S4", "S5", "I0", "S0", "S1", "S2" ],
  "finalStates" : [ "S5" ],
  "transitions" : [ {
    "from" : "I0",
    "to" : "S2",
    "actionLabel" : "OK",
    "newParts" : [ ],
    "existantParts" : {
      "role" : "",
      "participants" : [ "o" ]
    },
    "input" : "",
    "preCondition" : "",
    "postCondition" : "",
    "internal" : false,
    "externalCall" : false
  }, ...],
  "roles" : [ "B", "O" ],
  "rPAssociation" : [ {
    "role" : "O",
    "participants" : [ "o" ]
  }, ... ]
}
```