Transform contracts written in Peyton Jones' contract descriptive language to Solidity Assembly Language

by

K.S.M.Perera

2015/CS/098

This dissertation is submitted to the University of Colombo School of Computing

In partial fulfillment of the requirements for the Degree of Bachelor of Science

Honours in Computer Science

University of Colombo School of Computing

35, Reid Avenue, Colombo 07,

Sri Lanka

July 2020

Declaration

I, K.S.M.Perera (2015/CS/098) hereby certify that this dissertation entitled "Transform contracts written in Peyton Jones' contract descriptive language to Solidity Assembly Language" is entirely my own work and it has never been submitted nor is currently been submitted for any other degree.

....

Date Signature of the Candidate

I, Dr. Kasun Gunawardana, certify that I supervised this dissertation entitled "Transform contracts written in Peyton Jones' contract descriptive language to Solidity Assembly Language" conducted by K.S.M.Perera in partial fulfillment of the requirements for the degree of Bachelor of Science Honours in Computer Science.

....

Date Signature of the Supervisor

I, Dr. Chamath Keppitiyagama, certify that I supervised this dissertation entitled "Transform contracts written in Peyton Jones' contract descriptive language to Solidity Assembly Language" conducted by K.S.M.Perera in partial fulfillment of the requirements for the degree of Bachelor of Science Honours in Computer Science.

....

Date Signature of the Co-Supervisor

Abstract

Financial contracts play a vital role in the modern economy. As there are variety of contracts being traded in the financial markets, the natural language used to define those contracts imposed an ambiguity in the financial contracts. To eliminate that, Peyton Jones and co-authors proposed a standard representation towards the financial contracts by introducing a combinator library embedded in Haskell programming language. However, the fundamental problem of the need of a trusted central counterparty, suffered by every financial contract remained unchanged with this representation. The existence of the middle man in the financial contracts introduced certain risks and overhead to the contracts.

In order to overcome this situation, V.U. Wickramarachchi proposed an approach to facilitate the autonomous contract execution by exploiting the features and use cases of Ethereum blockchain and its scripting language Solidity. This was achieved through the special purpose compiler which facilitates transformation from Peyton Jones' Contract Descriptive language to Solidity. However, the cost related to the execution of contracts in Ethereum network curtail the benefits received through the transformation of those contracts.

In this dissertation, a novel approach to facilitate the reduction of cost by optimizing the smart contract is proposed using different optimization techniques. This approach involves the transformation of the Peyton Jones' Contract Descriptive language to Assembly language which enables the manipulation of data locations in the Ethereum Virtual Machine.

To substantiate that, the proposed method was able to scale down the execution cost factor in a significant manner, the solution is evaluated with the approach which transform Peyton Jones' CDL to solidity with a comparison between the execution cost of both the approaches. A formal verification is provided by verifying the semantic equivalence between the Peyton Jones' Contract Descriptive language and the proposed solution to make sure the correctness of the proposed approach is preserved while it is being optimized.

.

Preface

Transformation of financial contracts to smart contract in Assembly language is a novel approach proposed in this study. The objectives and aims of this study has been not explored by any other previous research of this particular domain. A novel design was introduced in order to facilitate this transformation. Two parts used in the design model; transforming from the Peyton Jones' Contract Descriptive language and the formal verification of the transformed contracts. The compiler used to transformed the Peyton Jones' Contract Descriptive Language was partially my own work. Part of it was taken from the previous work. Apart from that, the proposed design was solely my own work and a method similar has not been proposed in any other study relevant to this domain.

The implementation methodology used in this study was proposed by me. The evaluation model used in this study is a generally accepted formal verification. However, it was further improved by myself with the input of my supervisors. .

Acknowlegment

I would like to express my sincere appreciation to my principal supervisor, Dr. Kasun Gunawardena and co-supervisor Dr. Chamath Keppitiyagama for their constant guidance and encouragement, without which this work would not have been possible. For their unwavering support, I am truly grateful.

I would also like to extend my sincere gratitude to all the examiners and evaluators of my research for providing feedback on my research proposal and interim evaluation to improve my study.

My sincere thanks go out to our final year computer science project coordinator Dr. H.E.M.H.B.Ekanayake for his encouragement and support in keeping this research focused and on-track.

Foremost my special thanks to my parents for providing me a solid foundation in education and all the courage and love gave me on every moment. They are the guiding stars which strengthen me to become the person who I am today.

Finally, I express my sincere gratitude to all my friends who supported and encouraged me on all cause of challenges I faced during this research. All the help given by everyone to make this research a success owns my great appreciation. .

Contents

List of Figures

List Of Abbreviations

AST	Abstract Syntax Tree
DSEL	Domain Specific Embedded Language
DSL	Domain Specific Language
EVM	Ethereum Virtual Machine
HCCL	Haskell Contract Ccombinator Library
LTS	Labelled Transition Systems

Chapter 1 - Introduction

1.1 Background to the research

Technological evolution in the modern world has deeply affected the global economy. The access of growing technology has many benefits over the modern economy. It encourages new business and advances the communication. More importantly technology has contributed towards reducing cost in many industries in modern world. One such technology which support modern businesses immensely is smart contracts. It is a technology which allows credible transactions without the help of a trusted third party. In such an era, are the smart contracts in its optimized way?

1.1.1 Financial Markets and Contracts

A financial market is any marketplace where buyers and sellers participate in the trade of assets such as equities, bonds, currencies and derivatives. Transparent pricing, basic regulations on pricing and market forces determining the prices of securities that trade, defines a typical financial market. Investors have access to a large number of financial markets representing a vast array of financial products.

A derivative is a financial security with a value that is reliant upon or derived from an underlying asset or a group of assets. The underlying asset in subject is commonly known as an observable. The derivative itself is a contract commonly entered into by two or more participants in the financial markets.

Some of the financial derivatives which are commonly traded in financial markets are given below.

- Zero Coupon Bonds -It is a debt security that is issued at a deep discount to its face value but pays no interest.

- European Options - It is an option that can only be exercised at the end of its life, at its maturity.
- American Options - It is an option that can be exercised anytime during its life. American options allow option holders to exercise the option at any time on or before its maturity date.
- Futures -They are financial contracts obligating the buyer to purchase or the seller to sell a financial instrument, at a predetermined future date and price.

Most of the contracts which are in existence in financial markets are composite contracts. Composite contracts are derived through the combination of sub contracts.

1.1.2 Smart Contracts

With the emergence of the blockchain technology, smart contracts have reached the hype in many industries, especially in the financial industry.

A smart contract is a computer protocol which digitally facilitates, verifies or enforces the performance of a contract between the two or more parties involved in the contract. The agreements contained in these self-executing contracts exist across a distributed, decentralized blockchain network which gives it a certain security and immutability.

Smart contracts possess features such as autonomy, trust, backup, safety and accuracy. They are run in just the way they are programmed.When running on the blockchain a smart contract becomes like a self-operating computer program that automatically execute when a certain conditions are met.

1.1.3 Ethereum

Ethereum is a platform that runs smart contracts. It is an open source and decentralized block-chain based platform. Anyone can use this platform to create decentralized applications. Ether(ETH) is the currency for the Ethereum platform. Ethereum is a peer-to-peer network in which the participants store and

execute programs on an embedded virtual machine. Ethereum is a distributed public blockchain network simillar to bitcoin.

1.1.4 Solidity

Solidity is an object-oriented, contract-oriented, High-level programming language for writing smart contracts. It is used for implementing smart contracts on various blockchain platforms including Ethereum. Solidity is statically typed script language which does the process of verifying and enforcing the constraints at compile time as opposed to run time.

1.1.5 Etherum Virtual Machine

Ethereum Virtual Machine is a stack-based, big-endian Virtual machine which is used to run smart contracts on the Ethereum Blockchain. It uses its own machine language known as EVM bytecode[3]. EVM makes the process of creating blockchain applications much easier and efficient. This enables the development of potentially thousands of different applications all one platform, without having to build an entirely originaly blockchain for each new appliation.

1.1.6 Ethereum Gas

In order to execute an operation or a transaction in Ethereum, user must have to pay for the computation to the network. Gas is what is used to calculate the amount of fees that need to be paid and the gas is paid in ETH(Ether). Gas price is the amount of ether that the user willing to pay per gas. Even though the Gas cost is fixed per operation, the gas price is dynamic and it dictate the market condition. Before the transaction, user specify the gas price in GWei. GWei is the most commonly used unit of ether to specify the gas price. The total fee that the miners get paid for the transaction is

$$gasPrice * gasUsed \tag{1.1}$$

As EVM is a Turing-Complete, there is a possibility that a contract would enter into an infinite loop. The "gas" which is needed to pay before the execution of a contract would prevent this kind of situations. Each execution step that triggers would consume a certain amount of gas and when the provided gas is over, the executions will get stop.

1.2 Research Problem and Research Question

1.2.1 Research Problem

Financial Contracts play a vital role in modern financial and business context and they are best used as risk management tools in which transfer the risk associated with the underlying asset to a party who willing to take the risk. Peyton Jones'[24] have introduced a contract definition language that describes such contracts precisely and a compositional denotational semantics that says what such contracts are worth.

With the ongoing technological advancement in the world, execution of the financial contracts have become automated. Smart contracts enhance financial contracts by automating its function and reducing the risk by eliminating the need of the middle man to execute this contracts. An approaches has been taken to map Domain Specific Language(DSL)s to autonomous, self-executing smart contracts[27] through a high level language. Then they are deployed and executed on the Ethereum blockchain using EVM. It combines the Ethereum properties into financial contracts which reduces the security risk of exercising contracts.

Ethereum charges a fee for the transactions that take part in the Ethereum network. As this fee is calculated using the amount of computation effort require to execute a certain operation, this cost can be reduced by reducing the computational effort in a transaction. This cost reside not only as a price that user has to bear but also as a fact that determine the correctness of the smart contract. If the transaction runs out of gas, the transaction will reverted back to the original position. So that a transaction to complete successfully, it needs to be provided adequate amount of gas prior to the transaction. Lesser the cost, higher the chances that the transaction complete successfully.

1.2.2 Research Question 1

To what extent the efficiency can be increased when transforming contracts written in Peyton Jones' Contract Descriptive Language directly to Assembly Language. rather than transforming through Solidity?

1.2.3 Research Question 2

To what extent the verification can be done to the contracts directly transformed to Assembly language?

1.2.4 Research AIM and objectives

This research focuses on discovering the possibilities of improving the efficiency of contracts transformed from Peyton Jones' contract descriptive language into EVM, while preserving its properties. The main aim is to facilitate autonomous financial contracts with reduced risk and overhead without having to write identical contracts in EVM.

The objectives of the research are as follows.

- Explore the ability to reduce the overhead by removing the middle compiler
- Enhance financial contracts by reducing the risk
- Explore the ability to verify the contracts transformed to EVM

1.3 Justification for the Research

Considering the economical importance of financial contracts and the technological importance of smart contracts in the modern society where the technology has revolutionised in many aspects, the motivation behind this study is to enhance the uses of smart contracts by increasing the efficiency and reducing the computations of these smart contracts without reducing the behaviour . By analysing the EVM and how it works, this idea become both possible and practical.

When the computations in a transaction increases, it increases the gas cost and

ultimately it makes the user to pay more money for the transaction to proceed. Because the amount of gas needs for the transaction has to be paid using ether and ether has to bought using the physical money. The economic benefit received through the elimination of the central counter party in financial contracts by automating its functions through the use of smart contracts tend to be reduced by the cost of gas involved in the smart contract transactions.

In the situations where the provided gas for a transaction is insufficient for it to be completed, transaction will stop in the middle resulting in a failed transaction. Even the transaction reverted back to its original state like nothing actually happened, the gas spent on the uncompleted transaction will not returned back. So that, in order to complete a transaction successfully, sender must have to be provided sufficient gas amount before the proceeding of the transaction. Thus, reducing the gas limit of the transactions would increase the chances of successful completion of transactions.

Gas price, which is decided by the user is paid to the miner. So that miners would prioritize the transactions with the higher gas price. Higher the gas price user wiling to pay, faster the transaction that will be processed. Reducing the gas cost for a transaction would give the user more chances to specify a high gas price, so that they could process their transactions in a speedy manner. Hence, reducing the gas limit would contribute to speedup the transactions being processed.

It has been shown that there are various ways [12]that the gas cost for a transaction can be reduced. Therefore, rather than re-writing financial contracts as smart contracts in a way the gas cost is reduced, it is beneficial to efficiently transform those contracts from the domain specific language(Peyton Jones' Contract Descriptive Language) to the EVM Assembly. Reducing solidity code from the contracts to the maximum possibility would contribute to the reduction of the gas cost required for the transaction.

However, even though there are few examples of transforming a contract in the financial domain to the Ethereum Blockchain domain, there were not any direct transformation of contracts to EVM. This holds significant scientific value, as it combines the two major domains and derives the best features of both the domains.

Furthermore, the Correctness of the transactions holds a major importance in these two domains. Without a formal verification, transacting through different parties is a great risk. Several approaches[1][14] have been taken to the formal verification of the smart contracts. The motive of this research is to exploit the advantages of optimizing smart contracts while preserving the semantics of the financial contracts. The importance of the outcome of this research is pointed towards both the financial domain and the computer science research areas respective to blockchain technologies.

1.4 Methodology

The first phase of the research followed an applied research methodology[17], in which a solution for the optimization of smart contracts is provided by reducing the amount of gas needed for a transaction. A novel concept for transform contracts directly from source to assembly was explored in this phase.

The Haskell Combinator Library of Peyton Jones' is a set of primitive combinators defined based on Haskell. This library is capable of describing financial contracts and perform contract valuation. EVM Bytecode is machine language which is understood by the Ethereum Virtual Machine.

Preliminary study of the concept would be the initial step of the project followed by the demonstration of proof of concept. Solidity language, inline assembly of solidity, EVM Bytecode, and how a contract is working with respect to the stack, memory and storage was the main study that comprehensively carried out during this period of time.

The underline aim of the research is to optimize the smart contracts through a direct transformation to Byte code while preserving the properties of DSEL. Therefore, in order to support the properties of DSEL on the Ethereum Virtual Machine, a source to source compiler (as shown in Figure 1.4.1) is built.

The solidity code[27] which was transformed from HCCL was handcrafted by adding inline assembly, to examine the optimization that could be incorporated to the code. Development of the compiler was proceeded after inspecting the success

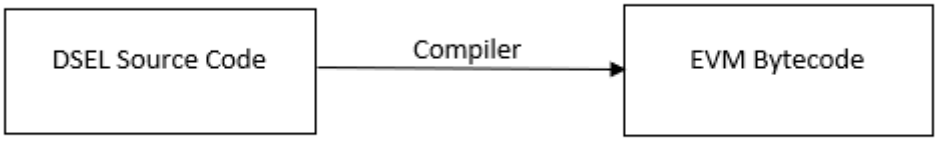


Figure 1.4.1: Source to Source Compiler

of optimization in the handcrafted code. The complete analysis and discussion of this step is included in section 4.

Furthermore, The abstract syntax tree of the transpiler[27] would be taken as the input to the proposed method. The process of compilation would be as follows(Figure 1.4.2).

The final step focuses on the evaluation plan of the transformed contract. This step would also include the formal verification of operational semantics in the contracts transformed to assembly language. The evaluation plan would mainly conducted in two phases.

1. Evaluation of the property preservation
2. Evaluation of the optimization

More details on the evaluation plan would be discussed in section 5.

As the latter part of the research, analysis of the smart contract with respect to gas consumption, how this optimization would support the correctness of the transaction, etc would be followed.

1.5 Outline of the Dissertation

The dissertation is as follows. Chapter two explores the existing approaches to create DSLs for financial contracts and re-implementations of the financial contracts for the Ethereum blockchain. In that chapter it further describes about the existing methods of formal verification of smart contracts. Chapter three describes the proposed research design and methodology. Potential ways of addressing research

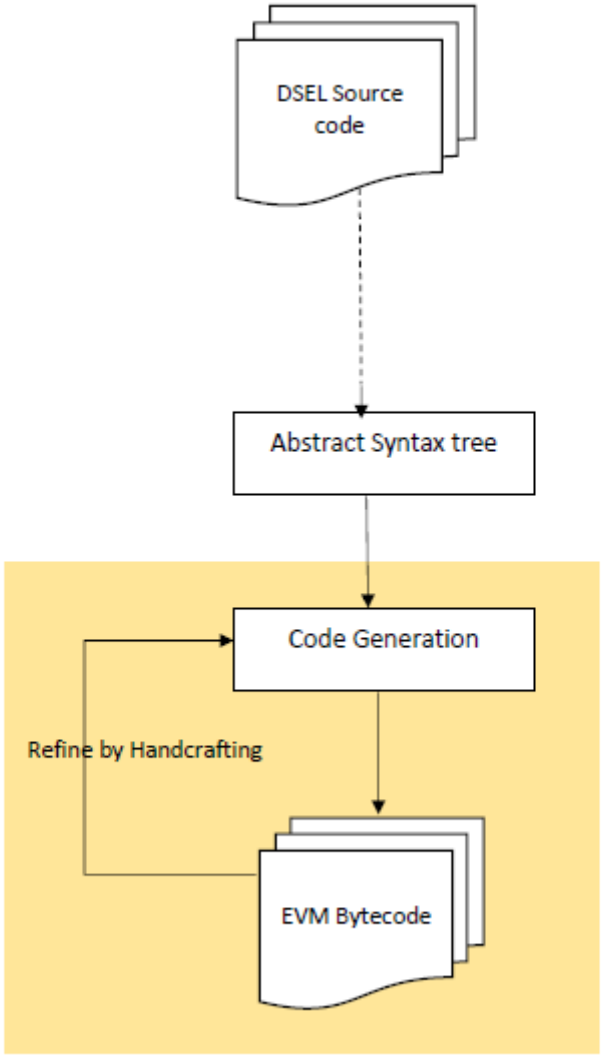


Figure 1.4.2: Research Approach Diagram

problem is discussed in this chapter. Chapter four demonstrates the implementation details of the proposed methodology. Chapter five presents the evaluation model and the evaluation results of the proposed approaches. The last chapter, chapter six provides the conclusion of the dissertation and outline of the future work.

1.6 Delimitation of Scope

The proposed methodology of this study only considered about the Peyton Jones' Contract Descriptive language[24] as the source language for the language trans-

formation. The input for the proposed method is the AST of the transpiler[27] and AST optimization is not considered in the proposed method.

The intention of this research is not to develop a new domain specific language or a new library which describes such contracts. It is out of scope to address existing issues of Ethereum platform where the smart contract are deployed and executed.

The types of contracts considered in this study is Zero coupon bonds, American options, European options and composite contracts. Optimization of the contracts has been measured in terms of the amount of gas used in each contract. Due to the time constraint, verification of the contracts has been done to the "zero", "one" and "and" combinators.

1.7 Summary

Financial contracts are widely used in modern economy. These financial contracts consist of large vocabulary of financial jargon. In order to address the ambiguity caused by this, Peyton Jones' et al.[24] introduced a Domain Specific Embedded Language(DSEL) which describes the financial contracts. Another problem that these representations were suffered was the need for a trusted third party. This issue was solved by transforming the above mentioned language into smart contract[27] in which the contracts can execute in an autonomous and trustless environment. But as a result of this another problem occurred. That is due to the gas cost needs for the contracts to run on the ethereum network. Therefore, it is more desirable to optimize these smart contracts in order to reduce the gas limit of the each transaction.

This chapter mainly highlights the background of the research, the research problem and the research questions. Then the research was justified, the methodology was briefly described, the dissertation was outlined, and the limitations were given. On these foundations, the dissertation can proceed with a detailed description of the research.

Chapter 2 - Literature Review

2.1 Introduction

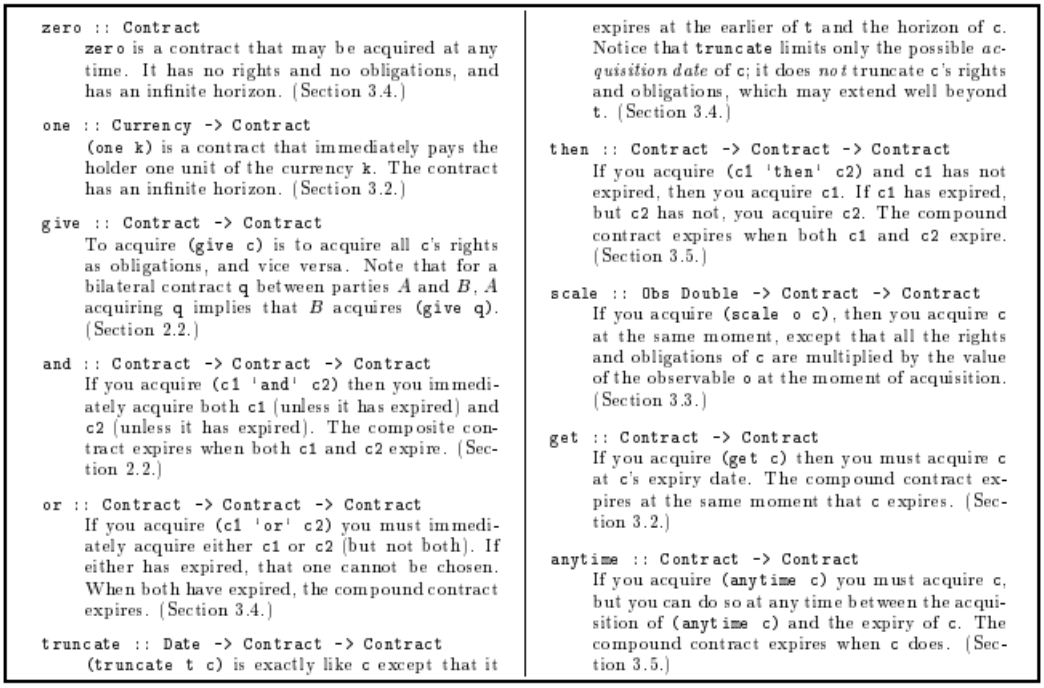
The research focuses on transforming financial contract in a DSL to Solidty Assembly language. So the approaches to define a DSL for financial contracts, smart contact Applications, Gas cost analysis and source to source compiling are the areas that we considered on related works. As this research focuses on the formal verification of the transformed contracts, related works of the formal verification and trace equivalence are also considered.

2.2 Approaches to define a DSL for financial Contract

The vocabulary used to name contracts was massive and it did not have capability to describe a novel contract with different operational semantics. In order to bridge the gap Peyton Jones' et al.[24] has introduced a domain specific embedded language (referred to as the Haskell Contract Combinator Library / HCCL) which was based on Haskell purely functional language. Financial contracts could be described and processed using the combinator library proposed by them.

Further, compositional denotational semantics were introduced in order to find the value of such contracts. They also sketched an implementation of valuation semantics, using as an example a simple interest rate model and its associated lattice. Due to the compositional nature of the approach, complex contracts could be easily described through the use of primitives for observables and primitives for contracts. The compositional nature of this notation enables to define complex contracts indefinitely based on the existing simpler ones. Figure 2.2.1 shows the combinators

introduced by Peyton Jones' et al. to describe financial contracts.



`zero :: Contract`
`zero` is a contract that may be acquired at any time. It has no rights and no obligations, and has an infinite horizon. (Section 3.4.)

`one :: Currency -> Contract`
`(one k)` is a contract that immediately pays the holder one unit of the currency `k`. The contract has an infinite horizon. (Section 3.2.)

`give :: Contract -> Contract`
To acquire `(give c)` is to acquire all c's rights as obligations, and vice versa. Note that for a bilateral contract `q` between parties A and B, A acquiring `q` implies that B acquires `(give q)`. (Section 2.2.)

`and :: Contract -> Contract -> Contract`
If you acquire `(c1 'and' c2)` then you immediately acquire both `c1` (unless it has expired) and `c2` (unless it has expired). The composite contract expires when both `c1` and `c2` expire. (Section 2.2.)

`or :: Contract -> Contract -> Contract`
If you acquire `(c1 'or' c2)` you must immediately acquire either `c1` or `c2` (but not both). If either has expired, that one cannot be chosen. When both have expired, the compound contract expires. (Section 3.4.)

`truncate :: Date -> Contract -> Contract`
`(truncate t c)` is exactly like `c` except that it expires at the earlier of `t` and the horizon of `c`. Notice that `truncate` limits only the possible *acquisition date* of `c`; it does *not* truncate `c`'s rights and obligations, which may extend well beyond `t`. (Section 3.4.)

`then :: Contract -> Contract -> Contract`
If you acquire `(c1 'then' c2)` and `c1` has not expired, then you acquire `c1`. If `c1` has expired, but `c2` has not, you acquire `c2`. The compound contract expires when both `c1` and `c2` expire. (Section 3.5.)

`scale :: Obs Double -> Contract -> Contract`
If you acquire `(scale o c)`, then you acquire `c` at the same moment, except that all the rights and obligations of `c` are multiplied by the value of the observable `o` at the moment of acquisition. (Section 3.3.)

`get :: Contract -> Contract`
If you acquire `(get c)` then you must acquire `c` at c's expiry date. The compound contract expires at the same moment that `c` expires. (Section 3.2.)

`anytime :: Contract -> Contract`
If you acquire `(anytime c)` you must acquire `c`, but you can do so at any time between the acquisition of `(anytime c)` and the expiry of `c`. The compound contract expires when `c` does. (Section 3.5.)

Figure 2.2.1: Combinators for defining contracts

Even though the representation of financial contracts were standardized through this approach, it did not solve many risks such as the counterparty risk, credit risk, etc. which still prevails when executing a financial contract. The need to trust a third party in order to exercise the rights of the contract was also a dependency the involved parties had to face.

Gaillourdet[11] worked on an extension to the HCCL by Peyton Jones' et al. which resulted in another domain specific language to describe financial contracts. She also focused on developing a notion of equality of derivative contracts. This helped her to model the ideal of the contract instead of its syntactic representation. The focus of this research was to develop a more generic language than the HCCL.

Both these studies have contributed immensely to the financial domain and the modern economy. The valuation semantics proposed, gives great insights as to how much a contract is worth.

2.3 Smart Contract Applications

The idea of autonomous smart contracts was first introduced by Vitalik Buterin[5] for the Ethereum blockchain platform. Unlike the Bitcoin blockchain, the Ethereum blockchain could process snippets of code written using a Turing-complete scripting language, Solidity. The contracts written for the Ethereum platform was decentralized. Naturally, this meant that the contracts are self-executing when triggered by a transaction or a function call.

Smart contracts define how entities may transact, and automatically execute these transactions when asked to. Once the smart contract is initiated, the participating entities cannot reverse or stop the transaction unless allowed by the contract itself or another smart contract; the agreement is enforced.

Gavin Wood in his Ethereum: A Secure Decentralized Generalized Transaction ledger[28] provided a system which guranteed the users to interact with other individuals or organizations with obsolute confidecne in the possible outcome and how those outcomes might come about.

Blocks, states, transactions and the execution of a transaction has been well explained by Gavin Wood in his paper. Gas limit(which is essential to any transaction in ethereum network) calculation was well explained by giving the specific gas cost for each instruction and the formula for gas cost calculation.

semi-automated translation of human-readable contract representations into computational equivalents was proposed by Christopher K. Frantz[10] as a modeling approach to enable the codification of laws into verifiable and enforceable computational structures which resides in a public bloackchain. They identified a smart contract components that corresponds to real world institutions and proposed a mapping using a domain specific language to support the contract modelling phase.

2.4 Gas Cost Analysis

As gas is essential for a transaction, gas should be provided before the execution of the transaction. The gas provided should be greater than the gas Limit of the transaction(amount of gas that is required for a transaction to complete). Because the execution proceed as long as gas is available. But the gas Limit generally cannot be predicted ahead of time. It is not possible to reveal where in particular within a transaction the high costs originated from.

Christoper Signer[25] has provided a tool which is known as Visualgas to visualize gas costs in depth to support more gas efficient development of smart contracts.Visualgas makes it easy for developers to test gas cost and explore best and worst case execution before deployment. Furthermore, it give a detailed overview of how the costs relate to different parts of the code.

Visualgas provides valuable insights so that it lets developers examine gas costs in detail and find ways to save gas. It offers visual, aggregated costs linked to the source code instead of step by step information.

As gas limit restricts the execution of Ethereum Smart Contracts, it is valuable resource that can be manipulated by an attacker to provoke unwanted behaviour in a victim's smart contract.Gas-focused vulnerabilities exploit undesired behavior when a contract runs out of gas. Neville Grech et al.[12] provided a way to classify and identify gas-focused vulnerabilities. They presented a static program analysis technique to automatically detect gas-focused vulnerabilities with high confidence and it is known as MadMax. It combines a control-flow-analysis-based decompiler and declarative program-structure queries. The combined analysis captures high-level domain-specific concepts and achieves high precision and scalability.

2.5 Source to source Compiling

V.U Wickramarachchi[27] implemented a source to source compiler that translate Peyton Jones' Contract Descriptive Language to Solidity. Peyton Jones' langauge is a domain specific language that was implemented using Haskell. The base language

is an object-oriented, high level language for implementing smart contracts. It was done to combine the properties of Peyton Jones' Contract Descriptive Language and the properties of the Ethereum blockchain. From that it eliminate the dependence on third-parties to execute financial contracts and facilitate autonomous execution of contracts in a trust less environment in order to reduce risks encountered when exercising financial contracts.

The overview of the compilation process is shown in the Figure 2.5.1. The initial methodology(up to Abstract Syntax Tree) of this research can be taken to our research as ours is a continuation of this research.

Matt Suiche[26] has developed a open source tool named as Porosity as a decompiler for EVM Bytecode that generate readable Solidity Syntax Contracts. So that it enabled static and dynamic anaysis of such compiled contracts.

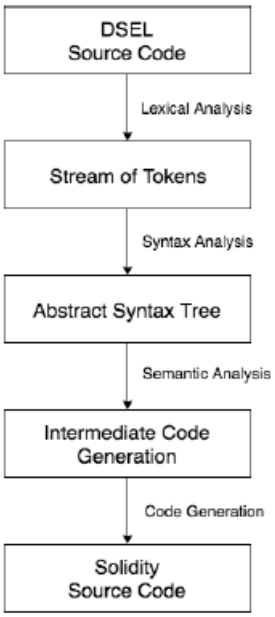


Figure 2.5.1: Compilation Process

2.6 Formal Verification

Due to the immutability nature of smart contracts, it is impossible to change after they arep deployed. So it is important to verify the contracts by defining the seman-

tics. Jiao Jiao et al. [16] develop the structural operational semantics for solidity, which allows to identify multiple design issues which underlines many problematic smart contracts.Furthermore, the semantics developed by them were executable in the K framework, so that it allows to verify/falsify contract automatically.

Karthikeyan Bhargavan et al. introduced a framework to analyze and verify both the runtime safely and functional correctness of the Ethereum contracts by translation to F*, which is a functional programming language to verify programs. Their approach was based on shallow embeddings and typechecking within an existing verifcation framework. So that it was convenient for exploring the formal verifcation of contracts coded in Solidity and EVM bytecode.

Magnus O. Myreen[18] presented a new approach for program verification based on translation.The automation and the proofs have been implemented in the HOL4 theorem prover, using a new machine code.

Xiaomin BAI[4] introduces a formal modelling and verification in formal methods to build smart contract model and verify the properties of smart contracts. Formal methods combined with smart contracts aim to reduce the cost and potential errors during contract development process. Spin model checker was used to verify the correctness and necessary properties for a smart contract template.

Zeinab Nehai et al.[19] propose a modelling method for an Ethereum application based on smart contracts. It was aim to apply a formal method(model checking) to verify that the application implementation complies with its specification, formalized by a set of temporal logic propositions. NuSMV tool has been chosen to support this model.

A formal verification tool for EVM Byetcode was introduced by Daejun Park at el.[23] To precisly reason about all the behaviours of the EVM Bytecode, they adopted KEVM. KEVM is complete formal semantics of the EVM, and instantiated the Kframework's reachability logic theorem prover. It is used to generate a correct-by-construction deductive verifier for the EVM. They further optimized the it by introducing EVM-specific abstractions and lemmas to improve its scalability. As they have chosen EVM bytecode as the verification target language, they can directly verify what is actually executed without the need to trust the correctnes

of the compiler.

Smart Contracts in Ethereum are executed by the Ethereum Virtual Machine. Yoichi Hirai[14] defined the Ethereum Virtual Machine for interactive theorem provers. It was the first formal EVM defnition for smart contract verifcation that implements all instructions. It was defined using Lem so that it can be compiled to a few interactive theorem provers such as coq,Isabella/HOL,HOL4. Using the definition, they proved some safety properties of Ethereum smart contracts in an interactive theorem prover Isabelle/HOL.

Everett Hildenbrandt et al.[13] presented a first fully executable formal semantics of EVM, the bytecode language in which smart contracts are executed. They have created these semantics in the K framework which was a framework for executable semantics. Their approach was feasible and not computationally restrictive.

Russell O'Connor[22] defined a new language known as Simplicity which was designed to be used for crypto-currencies and blockchains. Simplicity is bounded, without loop but is expressive enough to represent any unitary function. Simplicity is amenable to static analysis that can be used to generate upper bounds on the computational resources needed, prior to execution. while Turing incomplete, Simplicity can express any unitary function, which we believe is enough to build useful smart contracts" for blockchain applications.

2.7 Equivalance

An important aspect of language translation is to verify whether those systems are equivalent to each other after the translation has been taken place.The checking of the equivalence has to be done in a formal manner to make sure that it eliminate with certainty as many problems as possible. Behavioral Equivalence make sure that two expressions behaves in the same way and those expressions evaluate to the same value. One of the most successful approach for describing the formal behavior of concurrent systems is the operational semantics. In this approach, these concurrent systems are modeled as Labelled Transitions systems(LTS) and they are consisted of set of states, set of transitions labels and a transition relation. The states represent the programs while the transition labels between the states

represent the actions or interactions that are possible in a given state.

Rocco De Nicola presented different equivalences[20] that preserve significantly different properties of systems. Extensional equivalences[21] for Transition systems are also presented by her. R.D. Nicola has presented another paper about the testing equivalences[7] for processes. In that paper she defined three different equivalences on process when the the processes and the set of tests for those processes were given.

The simplest of all equivalences is the trace equivalence[8][6]. Two processes are deemed trace equivalent, if and only if they can perform exactly the same sequence of actions, starting from their initial states. The Figure 2.7.1 further explains about the theorem. Tim Wood et al. has described a method for establishing the existence of the equivalence between the behavior of two programs using trace Equivalence. They have automatically checked a proof that their method is sound using the Dafny program verifier. But a drawback of Trace Equivalence is that it is not sensitive to deadlocks. Completed Trace Equivalence would solve that drawback. However, Completed Trace Equivalence is also consist of it is own problems.

LTS can be compared using the simulation or bi-simulation based equivalence relations. Bi-simulation can be considered as a strong form of equivalence and it is much stronger than the input/output equivalence. So that two LTSs may not be bisimilar despite the fact that they have the same behavior and they simulate each other. J.C. Fernandez et al.[9] deals with the efficiently minimizing and comparing LTSs with respect to bi-simulations and simulations based equivalence relations.They have implemented their experiments within the tool Aldebaran of various decision procedures for behavioral equivalence relations.

Formally, for any LTS $(S, A, \rightarrow, s_0)$ and state $s \in S$ we define $\mathrm{Tr}(s)$ to be the set of traces possible from s:

$$\mathrm{Tr}(s) = \{\sigma \in A^* | \exists_{t \in S} \ s \xrightarrow{\sigma} t\}$$

Given two LTSs $T = (S, A, \rightarrow, s_0)$ and $T' = (S', A', \rightarrow', s'_0)$, we say that T and T' are trace equivalent iff $\mathrm{Tr}(s_0) = \mathrm{Tr}(s'_0)$.

Figure 2.7.1: Trace Equivalance

2.8 Summary

This chapter described about the current status of the research domain, especially targeting towards the Peyton Jones' Contract Descriptive language, Source to source compiling and smart contracts. Then this chapter described about the different formal verification methodologies. Finally, it discussed about the trace Equivalence theorem.

Chapter 3 - Design

3.1 Introduction

This chapter elaborates the overview of the proposed solution to the research problem. It consist of four major sections, namely; Research Design, Optimizing the contracts, compiler design and formal verification.

3.2 Research Design

The research design comprise of three main sections. They are optimizing the contracts, Source-to-source compilation and the formal verification of the transformed contracts.

3.3 Optimizing the contracts

In order to do an optimization to the contracts, identification of how the gas cost calculated in a contract was the first phase in this section. Gas fees for the EVM are charged under three distinct circumstances. The first is the fee built-in to the computations of the operation. Different operations cost differently while the cost per operation is fixed for each operation. The detailed list of the fee schedule is shown in Figure 3.3.1. The second is the gas deducted to form the payment for a subordinate message call or contract creation. This forms the payment for the CREATE, CALL or CALLCODE. The contracts used in this study does not need these mentioned instructions to be included. So considering this for the optimization is purposeless. The third is the gas paid due to an increase in the usage of memory. The three data locations in EVM, the storage, memory and the stack require different amount of gas cost to store data in each of those locations. Among

them storage requires the highest gas cost while the stack requires the lowest gas cost. When considering those facts, it is understood that appropriate use of operations and data locations would enable the optimization of the contracts.The definition of the general gas cost function is shown in Figure 3.3.2

APPENDIX G. FEE SCHEDULE

The fee schedule G is a tuple of 31 scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

	Name	Value	Description*
execution costs	G_{zero}	0	Nothing paid for operations of the set W_{zero}.
	G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
	$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
	G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
	G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
	G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
	$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
	$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
	G_{sload}	200	Paid for a SLOAD operation.
	$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
	G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
	G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
	R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
	$R_{suicide}$	24000	Refund given (added into refund counter) for suiciding an account.
	$G_{suicide}$	5000	Amount of gas to pay for a SUICIDE operation.
	G_{create}	32000	Paid for a CREATE operation.
	$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
	G_{call}	700	Paid for a CALL operation.
	$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
	$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
	$G_{newaccount}$	25000	Paid for a CALL or SUICIDE operation which creates an account.
	G_{exp}	10	Partial payment for an EXP operation.
	$G_{expbyte}$	10	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
	G_{memory}	3	Paid for every additional word when expanding memory.
transaction costs	$G_{txcreate}$	32000	Paid by all contract-creating transactions after the *Homestead transition*.
	$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
	$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
	$G_{transaction}$	21000	Paid for every transaction.
execution costs	G_{log}	375	Partial payment for a LOG operation.
	$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
	$G_{logtopic}$	375	Paid for each topic of a LOG operation.
	G_{sha3}	30	Paid for each SHA3 operation.
	$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
	G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
	$G_{blockhash}$	20	Payment for BLOCKHASH operation.

Figure 3.3.1: Fee Schedule[28]

Two main costs are associate with the transactions that is executing on the Ethereum network. They are the Transaction cost and the execution cost .Execution cost is the cost of the computational operations which are executed as a result of the transaction. It is mainly consisted with the storage of the global variables, processing power used for calculations and the manipulation of local variables. As the execution cost is included in the transaction cost, it has been understood that reducing the execution cost which lead to the reduction of both the execution and the transaction cost. As mentioned above, efficient use of storage and calculations will lead to the reduction of the execution cost. Transaction cost is the cost of sending the contract code to the ethereum network and it depends on the size of the contract. It consisted with four main parts. The base transaction cost which

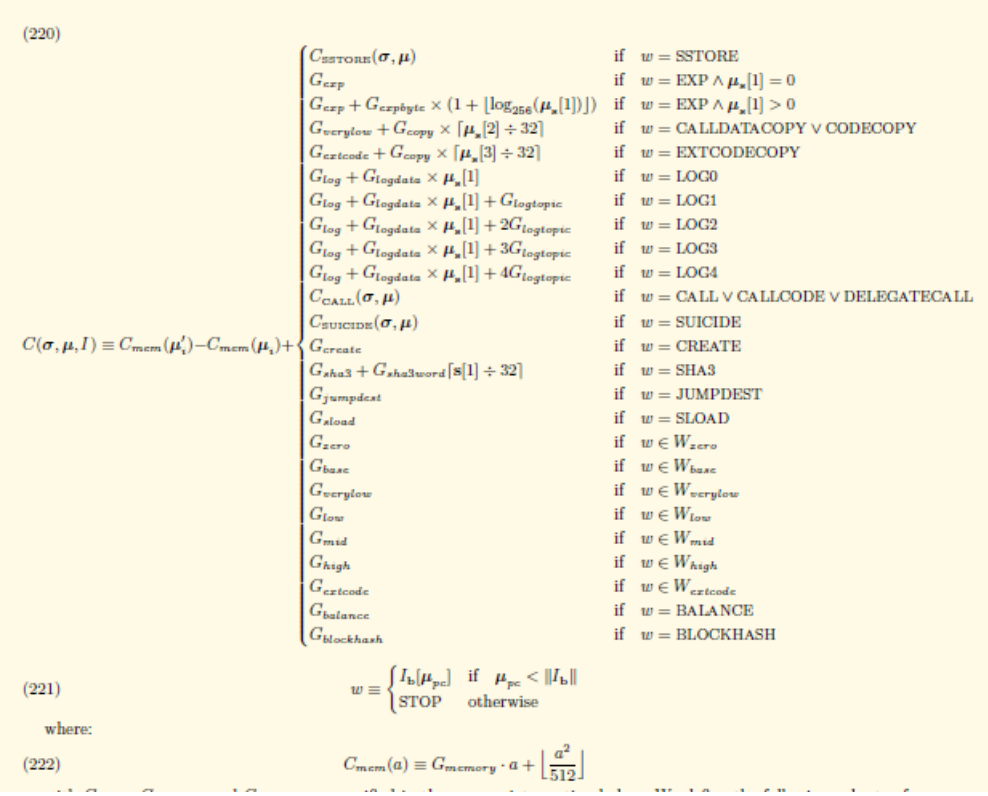


Figure 3.3.2: Gas Cost[28]

is 21 000(gas) and the cost for the contract deployment which is 32 000(gas) are fixed cost for any transaction. It is understood that the Transaction cost for any contract would be higher than 53 000(gas). Other two parts of the transaction cost are the transaction input cost and the contract initiated transactions cost. Reducing the transaction input and the size of the contract appropriately will lead to the reduction of the transaction cost. The figure 3.3.1 has shown the categorization of the operations according to the transaction and execution cost.

Current implementations of the smart contracts are in Higher level languages like solidity. Use of operations and data locations in an optimized manner is not easily done with higher level languages. In order to handle operations and data locations effectively, inline assembly was embedded into the Solidity Code. Handcrafting the solidity code by replacing it with inline assembly as much as possible and replacing the code beyond the line by line replacing, enhanced the code with optimization in a better way.

3.4 Compiler Design

The source language for this study is Peyton Jones' Contract Descriptive Language. The unambiguous and composible nature of this language allows to create complex contracts by combining the simple combinators.

The target language, which is to be executed in the EVM is the solidity language with the embedded inline assembly.

The compiler for this study has not been designed from the beginning. The beginning to the AST generator was taken from the Synergy compiler[27] to reduce the redundant work(The parse tree outputs of Synergy compiler includes in Appendix A). The compiler, from the AST has been designed to support the optimization. The tree walker of the compiler, walks the AST for the transformation of the contract in terms of combinators and the necessary logic to the target language. This performs the final geneartion of the optimized code. The compiler architecture is showed in Figure 3.4.1(Code listing includes in Appendix B)

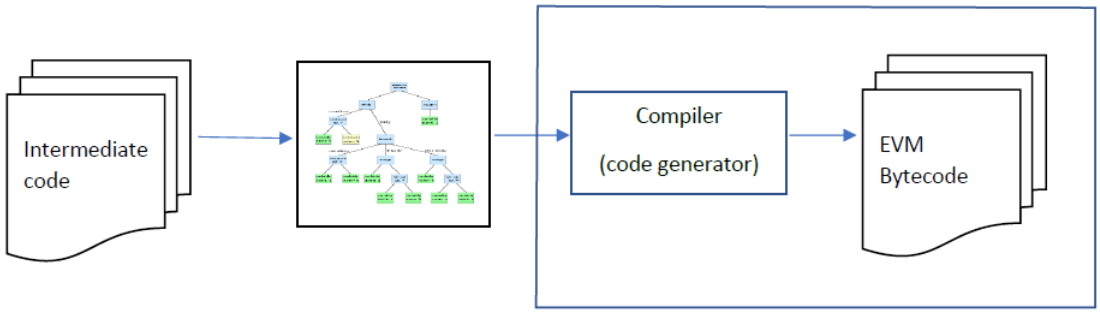


Figure 3.4.1: Compiler Architecture

According to Peyton Jones' CDL, a contract is consisted with a set of standard basic primitives(combinators). The following Figure3.4.2 shows the basic structure of the contract according to the AST. A basic contract would comprised with one or more combinators and that is depend on the input provided to the compiler(contract in the Peyton Jones' CDL).

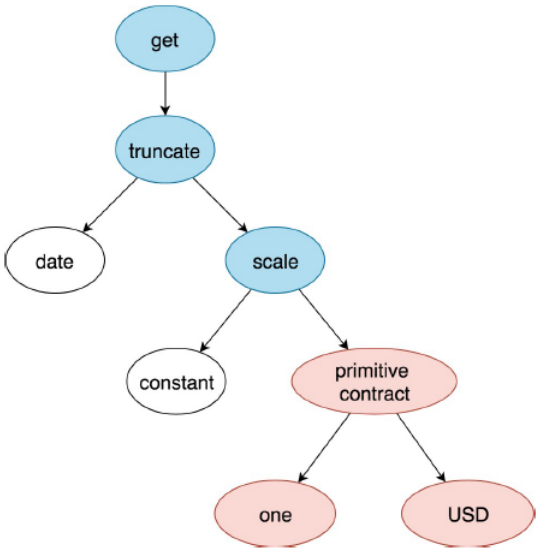


Figure 3.4.2: Structure of a basic contract

While traversing the AST, when the tree walker gets the 'get' combinator, a basic contract between two parties are created. According to the other combinators necessary components added to the created contract.

Apart from the basic combinators provided by Peyton Jones' CDL, there are additional information and actions required by the smart contracts in order to execute in the blockchain. For that purpose, an external contract named as Marketplace was introduced. All the functions that required for the execution is implemented in this Marketplace. The compiler generating contracts would call the functions from the Marketplace.

However, developing the compiler further more with the aspect of optimization, the Marketplace was eliminated by adding the additional information provided by the Marketplace to the transformed contract itself.

3.5 Formal Verification

The contracts introduced by the Peyton Jones' Language are more general in nature and they are not specifically designed to be executed on the EVM. When those contracts executed on the Ethereum network, those contracts need to be instantiated

with the parties related to the contract. When those general contracts executed in Ethereum with real addresses of the users, ambiguities might be occurred.When the contracts deployed and execute in the ethereum network, additional information and actions required which has not been needed to the contracts in the Peyton Jones' CDL. In order to show that the proposed contracts behave exactly the same way as the contracts in Peyton Jones' CDL behave and the semantics of Peyton Jones' CDL are preserved in the proposed contract, verification of the contracts in a formal manner was required. Therefore the proposed methodology in this study developed the traces for the contracts in Peyton Jones' CDL and the proposed contract and verified their equality using trace equivalence.

Chapter 4 - Implementation

4.1 Introduction

This chapter provides the implementation details of the proposed solutions. Section 4.2 describes the software tools utilized for the implementation process, section 4.3 illustrate how the contracts were optimized, section 4.4 describes the formal verification of the contract.

4.2 Technologies and Software Tools

The Remix IDE was used to deploy and execute the Ethereum smart contracts. It was used to check the gas cost for each transaction. It was used to test the behavior of the deployed contracts using test data. The code-generator of the compiler developed using JavaScript. The traces for the formal verification was built using the spin model checker. The language used to model the logic was PROMELA verification modeling language.

4.3 Optimizing the contracts

4.3.1 The Marketplace

This is an external contract which the compiler generated contracts used to call functions. All the functions that needs to be executed in order to do a transaction was implemented in this contract.The other additional details that the Ethereum network requires other than the Peyton Jones' CDL has mentioned are included in this contract. Marketplace enable the modularize and the scalable nature of the proposed solution. Further, information such as contract addresses and user balances for different commodities are kept in here. Before deploying any contract,

the instance of an Marketplace need to be deployed to the blockchain.

As all the function implementations are included in the Marketplace, the amount of gas need for the deployment of this contract is higher than the other contracts. So that the major optimization techniques are applied to the Marketplace. The Marketplace contract was handcrafted in order to achieve the optimization.

As the most expensive operation in a transaction in terms of gas cost is storing data in the storage data location of EVM, the use of storage has been minimized in the code. However, as there are information that has to be kept through out the contract, it is impossible to avoid using the storage locations completely. So that, few information has been stored in the storage in the Marketplace contract. Rather than declaring three new variables for user addresses(address of the contract holder, counter party, and the creator), one variable known as **ContractMetadata** was introduced. In the same way for the balances of the contract holder and the counter party two variables have declared for the two commodities, USD and GBP as **USDbalances** and **GBPbalances**. The information other than the addresses used the uint data type. It is always better to use uint data type over the string data type whenever possible because it is expensive to use string data type in terms of the gas cost. uint is equal to uint256. Even the data that we need to store is less than uint256 and can be used some other data type less than uint256(e.g.-uint8), it is still recommended to use uint because uint cost lesser than the other data types(e.g.- uint is cost lesser than the uint8). The reason for that is, the base uint for EVM is 256bits, so to downscale from 256 to a lesser size it need some operations and that makes the cost increased.

```
address public ContractMetadata;
uint isSigned = 0;
uint public USDbalances;
uint public GBPbalances;
address creator;
```

Few key functions enables the contract execution on the Ethereum network. Inline assembly was used to implement all the functions in the Mareketplace. As it is used as an intermediate language for the solidity compiler, it has reduced the

gas cost in a large number. It has been beneficial to provide an efficient code. This gives more control towards the code by providing the ability to handle the instructions and data locations. However, the method signature was not written in inline assembly. Solidity was used for that. If the method signature was in inline assembly, calling functions internally required some of the instructions like 'jump' which obfuscate the control flow. It was mentioned in the solidity documentation[2] that not to make use of explicit 'JUMP', 'JUMPI' statements. Not using of those instructions makes it easier to analyze the control flow, so that it help in formal verification and optimization.

The **propose** function is used to propose or sell the contract to another party by the counter party. When the propose method is executed it will keep up the necessary information for the further transaction. The addresses of the counter party, holder and the creator of the contract are get stored in the storage memory location, so that it can be used through out the contract. The addresses stored in the storage sequentially using the variable declared outside the function. As the Ethereum address holds a 20 bytes value, starting from the declared variable, the location to store address were manually mentioned while leaving the 20 bytes value for each address.

```
function propose(address counterparty, address holder) public{
    assembly{
        sstore(ContractMetadata_slot , counterparty)
        sstore(add(ContractMetadata_slot, 0x20), holder)
        sstore(add(add(ContractMetadata_slot, 0x20), 0x20), sload(creator_slot))
    }
}
```

The **sign** method allows the other party to agree with the contract. When the counter party propose the contract, holder can agree to it by the execution of this method. During the process, the state of the contract is changed by changing the variable of **isSigned**. Type of that variable is uint and the default value of that variable is '0', which means the contract is not signed and it is assigned '1' when the contract is signed.

Preparatory too execute the following functions, the contract needs to be signed. That means the **isSigned** variable needs to be stored with '1'. The **get** method

```
function sign(address holder) public{
     assembly{
        if eq(sload(add(ContractMetadata_slot,0x20)),holder) {
            for{} eq(sload(isSigned_slot),0) {} {
                sstore(isSigned_slot,1)
            }
        }
    }
}
```

is used to establish the contract between two parties while storing the expiry date of the contract. The **give** method does the same with reversed counter party and holder. The **receive** method allows counter party to transfer a certain amount of money in the particular commodity to the contract holder. As the use of String data type is expensive in terms of gas cost, uint data type was used to identify the commodity type. The values '0' and '1' used for that while '0' represent the 'USD' commodity, '1' represent the 'GBP' commodity.

```
function receive(uint commidity,int quantity) public{
  //0 for USD 1 for GBP
    assembly{
        let val
        mstore(val, commidity)
        let balanceOfUser
        switch mload(val)
        case 0{
            balanceOfUser := USDbalances_slot
        }
        case 1{
            balanceOfUser := GBPbalances_slot

        }
        for{} eq(sload(isSigned_slot),0) {} {
            let balanceHolder :=  add(sload(balanceOfUser),quantity)
            let balanceTo := sub(sload(add(balanceOfUser,0x20)),quantity)
            sstore(balanceOfUser,balanceHolder)
            sstore(add(balanceOfUser,0x20),balanceTo)
        }
    }
}
```

Another approach was also introduced for the receive functionality. In this approach, user balances were stored in wei which is the smallest denomination of ether. As the Peyton Jones' has defined the CDL to the currencies of USD and GBP, the commodities needs to be in USD and GBP. But the money transfer in

the Ethereum network happens using the currency ether. In order to reduce more gas cost, the USD or GBP to wei conversion was done in the compiler. By using the **balance(a)** instruction, wei balance at address 'a' can be easily accessed. By using this approach, input parameters of the function and the computations inside the function has been reduced, which would lead to the reduction of the gas cost.

```
function receive(int quantity) public{
    assembly{
        let sender := sload(_metaContract_slot)
        let receiver := sload(add(_metaContract_slot,0x20))
        let balanceHolder := add(balance(sender), quantity)
        let balanceTo := sub(balance(receiver), quantity)
        }
    }
```

4.3.2 Contract Transformation

The contract transformation has been done in two approaches. They are,

1. Contracts with the Marketplace
2. Contracts without the Marketplace

In the first approach solidity code was generated from the transformation. The calls to the necessary functionalities in the Marketplace was done in this code. The parameters for the functions were passed in the compiler.

```
contract c  {
    constructor (market marketplace, string memory horizon,int scale)public{
    }
    function proceed() public{
    marketplace marketplace;
    marketplace.get("2019-08-17");
    marketplace.receive(1,100);
    marketplace.AskForKill();
    }
}
```

In order to execute each contract that transformed through the compiler, Marketplace contract needed to be deployed before the compiled contract is deployed to the network. It was understood that to execute some of the simple transaction, deploying a contract with many functionalities was unnecessary. If only the necessary functions deployed with each contract, the gas cost could be saved from a high number.

So that, in the second approach Marketplace was removed. Necessary functionalities require to do the transactions are implemented within the contract that transformed through the compiler. An assembly code was generated from the compiler to make sure to achieve a reduction in the gas cost. The propose and the sign functions also included in the contract. After the counter party has proposed the contract and as soon as the other party sign the contract, the transaction is started to be executed. Less functions and functional calls would enable to reduce the gas cost of these contracts.

When analysing the AST it was understood that code templates could be created for the each part of traversal of the AST. The template of the propose function

```
function propose(address counterparty, address holder) public{
      assembly{
          sstore(ContractMetadata_slot , counterparty)
          sstore(add(ContractMetadata_slot, 0x20), holder)
          sstore(add(add(ContractMetadata_slot, 0x20), 0x20), sload(creator_slot))
      }
  }

  function sign(address holder) public{
       assembly{
          if eq(sload(add(ContractMetadata_slot,0x20)),holder) {
              for{} eq(sload(isSigned_slot),0) {} {
                  sstore(isSigned_slot,1)
              }
          }
      let expireDate := "1569868200 "


      if lt(sload(today_slot),expireDate) {
      let balanceHolder :=  add(sload(USDbalances_slot),1)
      let balanceTo := sub(sload(add(USDbalances_slot,0x20)),1)
      sstore(USDbalances_slot,balanceHolder)
      sstore(add(USDbalances_slot,0x20),balanceTo)
      }
      }
  }
```

and the part of the sign function needs to be included in every contract. The other templates are generated according to the AST and they are parameterized appropriately. While traversing the AST, compiler select the suitable template from the list of predefined templates. Finally the transformed contracts were deployed to the Ethereum Blockchain.

4.4 Formal Verification of the contract

In order to verify the generated contracts, it was required to develop the traces for the each processes in the propsoed approach and the Peyton Jones' approach. The traces ware developed using the spin(Spin Promela Interpreter)[15] model checker. It is an open source tool which used to verify correctness of the concurrent software models. It is one of the most popular and powerful tool for detecting software defects in concurrent system designs. The intended use of this tool is to verify the process behaviour which are considered suspect. The relevant behaviour was modeled in Promela(a Process Meta Language). It is a verification modelling language. The structure of Spin simulation and verification is shown in Figure 4.4.1.

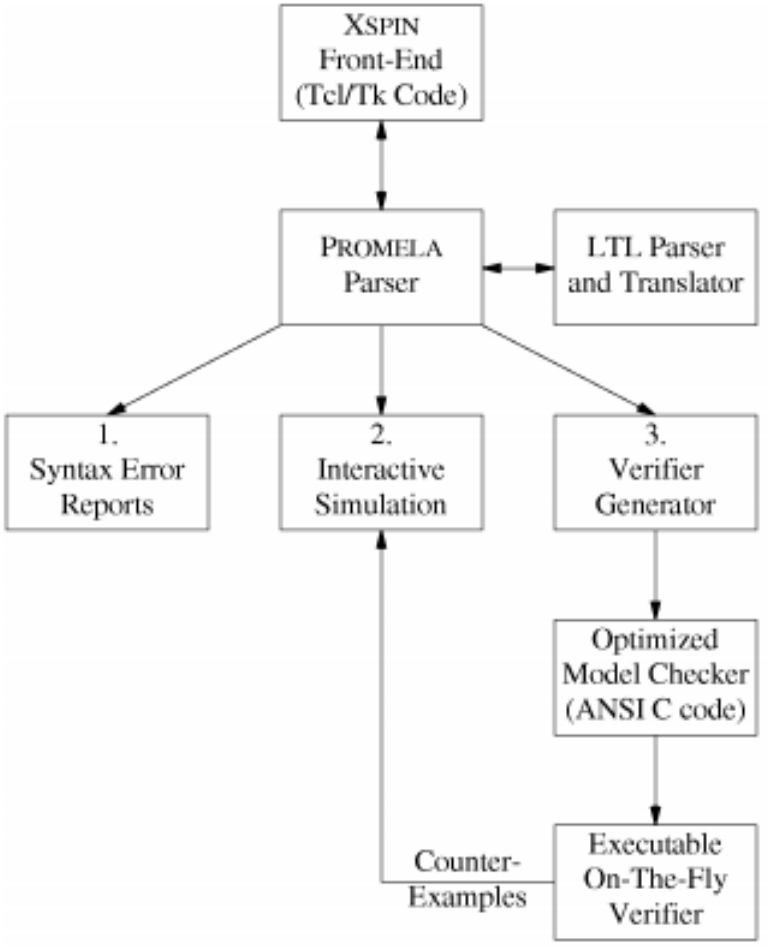


Figure 4.4.1: Structure of Spin simulation and verification[15]

```
1 active proctype ONE{
2     int t1=5,t=2,HolderAccount=50,CounterpartyAccount=50;
3     if
4     ::t<=t1->HolderAccount = HolderAccount-1;CounterpartyAccount
     +1;
5     ::else->skip
6     fi;
7 }
```

A promela model consist of type declarations, channel declarations, variable declarations, process declarations and an init process. The above shows a process defined for the ONE combinator of Peyton Jones' CDL. A process is defined by a proctype definition. Process can be created using the active keyword by adding

it in front of the proctype declaration. Several variables have been declared as local variables.The variables 't' and 't1' are declared for the current date and the horizon. For easy and more clear comparison the dates variables are declared in type int. However, there is no specific data type for date in Promela. The date of the transformed assembly code also compared usingg the timestamp. So that comparing two integers to model the date will not do any harm to the correctness of the model.

First it check whether the current date is before the horizon. The fund transfer will execute if only the current date is a date before the horizon. The value of the sender is decremented by one unit while the holder's account is incremented by one unit.

The following code shows the transfer of 100 USD in accordance to the Peyton Jones' CDL. In the Peyton Jones' CDL, this happens by scaling the ONE combinator by the scale amount and transfer the amount between the parties. The variable 'k' is defined for the amount to be sent between two accounts.

```
1 active proctype ONE{
2     int t1=5,t=2,HolderAccount=50,CounterpartyAccount=50,one=1,
    scale=100,k;
3     if
4     ::t<=t1->k=one*scale;
5             ->HolderAccount = HolderAccount-k;CounterpartyAccount+k
    ;
6     ::else->skip;
7     fi;
8 }
```

The Promela modeled code for the same scenario but in the proposed solution is shown in below. The scaling happens at the compiler so that the compiler generated code would transfer the scaled amount without a scaling operation at the code. There are some of the extra parts(e.g.-Propose and sign) added to this code which has not added to the Promela model of the Peyton Jones' CDL.

```
1 int ContractMetaData[3];
```

```
int isSigned=0,HolderAccount=50,CounterpartyAccount=50,t=5,t1=8,
    creator=3, k;
proctype Propose(int Counterparty, int holder){
    ContractMetaData[0]=Counterparty;
    ContractMetaData[1]=holder;
    ContractMetaData[2]=creator;
}
proctype sign(int holder){
    if
    ::(isSigned==0)->isSigned=1;
    ::else->skip;
    fi
    if
    ::(t<+t1)->k=100; HolderAccount=HolderAccount-k;
    CounterpartyAccount=CounterpartyAccount+k;
    ::else->skip;
    fi
}
init{
run Proposal(1,2);
run sign(2);
}
```

Uisng the Promela codes, the traces were generated to each scenario in the Spin Model checker. The equivalence of the generated traces for same scenario both in accordance to the Peyton Jones' CDL and the proposed approach was done using the trace Equivalence. The drawback of trace equivalence, which is that it does not sensitive to deadlocks will not be an effect to the verification because as the traces are drawn using spin model checker, it will check for the absence of deadlocks, unspecified receptions and unexecutable code. The two traces cannot be prove equal using bi-simulation because there is an additional part in the proposed method which is required for a transaction to be executed on ethereum network. But using the trace equivalance we can prove that all the necessary parts in the Peyton Jones' CDL us inlcuded in the proposed method.

When the source language and the proposed language compared with each other for the equivalence, it is to be noted that these two languages are different in type. Haskel, the language in which the Peyton Jones' has implemented combinator li-

brary is a functional programming language while the assembly in which the source is transformed to can be classified as procedural language. There are difference in those language itself. Functional programming languages emphasized on evaluating functions and it does not work on executing commands. Contrast to that procedural languages works on executing systematic sequence of statements,and commands to complete a computational task or program. By the nature of functional languages, the order of execution does not provide any problems but procedural language might resulted in different way for the different order of execution. This was considered and reckoned while doing the verification.

4.5 Summary

In this chapter the tools and technologies used in implementing the proposed solution is elaborated. The different options used for the optimization is mentioned and elaborated the important functionalities of each of those proposed options. The generation of the codes from the AST is also explained in this chapter. The steps followed in the formal verification is also explained in detail in the last subsection of this chapter.

Chapter 5 - Results and Evaluation

5.1 Introduction

This chapter elaborates the obtain results, how the results are evaluated and the success level of the proposed solution.

5.2 Transformed Contracts

Once the input of Peyton Jones' CDL is processed by the compiler, the transformed contract is obtained. The contract is generated by the tree walker after walking the AST.

5.2.1 With the Marketplace

This is the first option when generating the contracts. The external contract which is known as 'Marketplace' is deployed to the network and the contract which is generated from the compiler will call the required function from the Marketplace. As the functions in the Marketplace are implemented in Solidity assembly, the compiler generated code is a solidity code. The below code shows the compiler generated code of the basic 'one' contract, where 1 GBP is immediately received at contract execution.

```
1 pragma solidity ^0.5.10;
2 pragma experimental ABIEncoderV2;
3 import {marketplace} from './Marketplace.sol';
4 contract one  {
5    constructor (Marketplace marketplace)public{
6    }
7
```

```
8     function proceed() public{
9           marketplace marketplace;
10          marketplace.receive(1,1);
11          marketplace.AskForKill();
12    }
13 }
```

Contract invocation is done by calling the **proceed()** method in the generated code. The contract logic execution happens through calling the particular functions inside the proceed function. The two parameters in the **receive** function refers to the commodity type which is GBP(1 is for GBP) and the amount which is one.

Then the compiler generated code for a basic contract component is shown below.(Receive $100 on a particular date("2020-03-17") in future.)

```
1  pragma solidity ^0.5.10;
2  pragma experimental ABIEncoderV2;
3  import {marketplace} from './Marketplace.sol';
4  contract one  {
5     constructor (Marketplace marketplace)public{
6     }
7     function proceed() public{
8           marketplace marketplace;
9           marketplace.get("2020-03-17");
10          marketplace.receive(0,100);
11          marketplace.AskForKill();
12    }
13 }
```

5.2.2 Without the Marketplace

This is the second option when generating the contracts. To deploy this contract no other external contract is not required. Once the input of the Peyton Jones' CDL is processed by the compiler, the transformed contract in assembly was obtained. The contract was obtained after walking the tree walker and appropriate code templates are generated from the predefined list of code templates. Then the

generated templates are parameterized appropriately. The below code shows the compiler generated code of the basic 'one' contract, where 1 GBP is immediately received at contract execution.

```
 pragma solidity ^0.5.10;
 pragma solidity ^0.5.10;
pragma experimental ABIEncoderV2;
contract A {
   address public ContractMetadata;
       uint isSigned = 0;
       uint public USDbalances;
       uint public GBPbalances;
       address creator;
       uint today = now;
       function propose(address counterparty, address holder)
   public{
            assembly{
                  sstore(ContractMetadata_slot , counterparty)
                  sstore(add(ContractMetadata_slot, 0x20), holder)
                  sstore(add(add(ContractMetadata_slot, 0x20), 0
   x20), sload(creator_slot))
           }
       }
       function sign(address holder) public{
            assembly{
                  if eq(sload(add(ContractMetadata_slot,0x20)),
   holder) {
                     for{} eq(sload(isSigned_slot),0) {} {
                           sstore(isSigned_slot,1)
                     }
                  }
                  let expireDate
                  let balanceHolder
                  let balanceTo
                  expireDate := 1547058600
                  if lt(sload(today_slot),expireDate) {
                       balanceHolder :=  add(sload(
   GBPbalances_slot),1)
```

```
                        balanceTo := sub(sload(add(GBPbalances_slot
    ,0x20)),1)
                        sstore(USDbalances_slot,balanceHolder)
                        sstore(add(USDbalances_slot,0x20),balanceTo
    )
                        }
                        selfdestruct(ContractMetadata_slot)
            }
        }
}
```

When the holder proposed, the meta data of the contract will be stored in the ContractMetadata slot in the storage data location of the EVM. As soon as the counterparty sign the contract, the contract execution happens by transferring the one GBP. Finally the contract will call the self destruct function in order to suicide itself.

Next the compiler generated assembly code for a basic contract component is shown below. (Receive $100 on a particular date("2020-03-17") in future.)

```
 pragma solidity ^0.5.10;
 pragma solidity ^0.5.10;
pragma experimental ABIEncoderV2;
contract A {
   address public ContractMetadata;
       uint isSigned = 0;
       uint public USDbalances;
       uint public GBPbalances;
       address creator;
       uint today = now;
       function propose(address counterparty, address holder)
   public{
            assembly{
                  sstore(ContractMetadata_slot , counterparty)
                  sstore(add(ContractMetadata_slot, 0x20), holder)
                  sstore(add(add(ContractMetadata_slot, 0x20), 0
   x20), sload(creator_slot))
           }
```

```
17        }
18        function sign(address holder) public{
19             assembly{
20                  if eq(sload(add(ContractMetadata_slot,0x20)),
    holder) {
21                     for{} eq(sload(isSigned_slot),0) {} {
22                          sstore(isSigned_slot,1)
23                     }
24                  }
25                  let expireDate
26                  let balanceHolder
27                  let balanceTo
28                  expireDate := 1547058600
29                  if lt(sload(today_slot),expireDate) {
30                       balanceHolder :=  add(sload(
    USDbalances_slot),100)
31                       balanceTo := sub(sload(add(USDbalances_slot
    ,0x20)),100)
32                       sstore(USDbalances_slot,balanceHolder)
33                       sstore(add(USDbalances_slot,0x20),balanceTo
    )
34                       }
35                       selfdestruct(ContractMetadata_slot)
36             }
37        }
38 }
```

5.2.3 Order of execution

The order of contract execution of the proposed solution under the aproach 1 is in the Figure 5.2.1. The process starts with the transformation of the Peyton Jones' contract and ends when the transformed contract logic is executed on the Ethereum blockchain.

The order of contract execution of the proposed solution under the approach 2 where there is no Marketplace contract is in the Figure 5.2.2

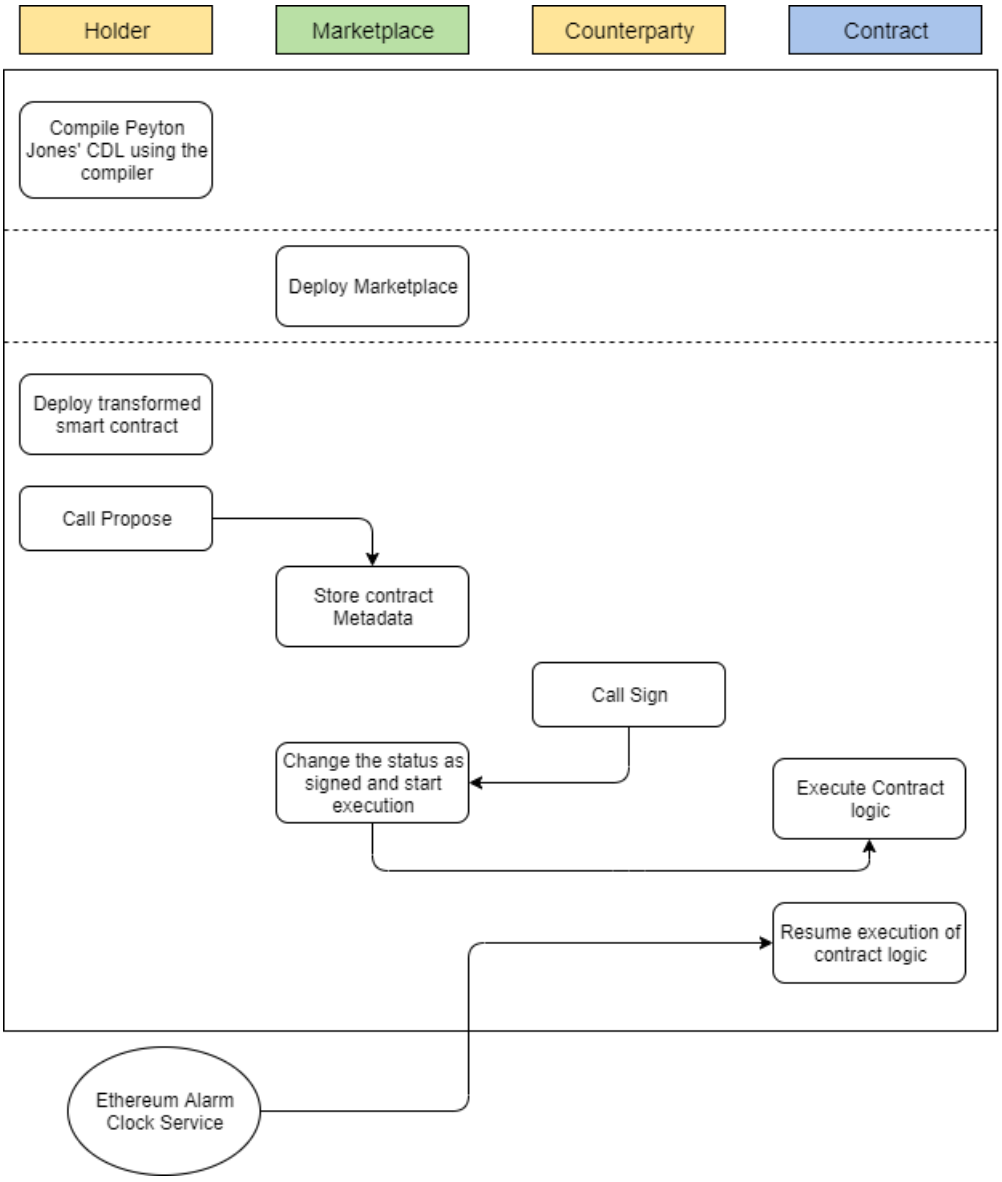


Figure 5.2.1: Order of execution of a contract from transformation to contract logic execution in the approach 1

5.2.4 Formal Verification

The traces have been generated for the different scenario in the Peyton Jones' CDL and the proposed solution. The following Figure 5.2.3 shows the trace for the ONE contract in the Peyton Jones' CDL which has been generated from the Spin model checker.

The following figures 5.2.4 5.2.5 5.2.6 shows the corresponding trace for the ONE contract in the proposed solution. As it is a combination of two functions, three traces have been generated from the proposed assembly code and they for

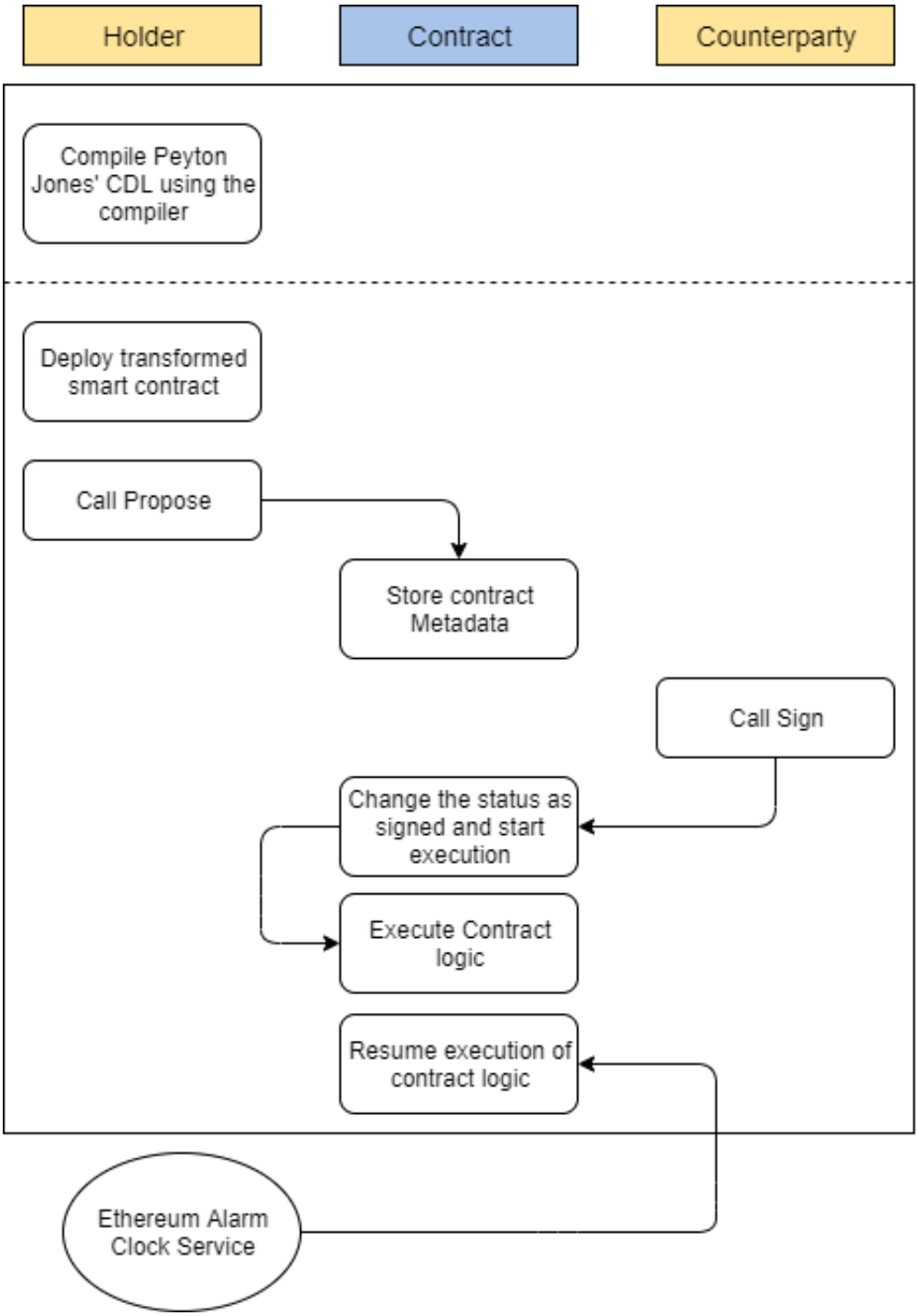


Figure 5.2.2: Order of execution of a contract from transformation to contract logic execution in the approach 2

the propose function, sign function and the init function which combines the both the functions.

The formal verification by the trace equivalence to check the equivalence of both contract will be explained in the section 5.3.2.

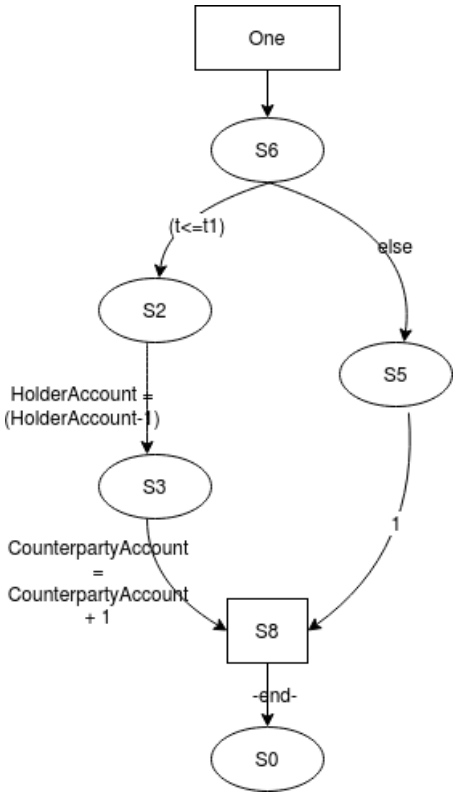


Figure 5.2.3: Trace for ONE contract in Peyton Jones' CDL

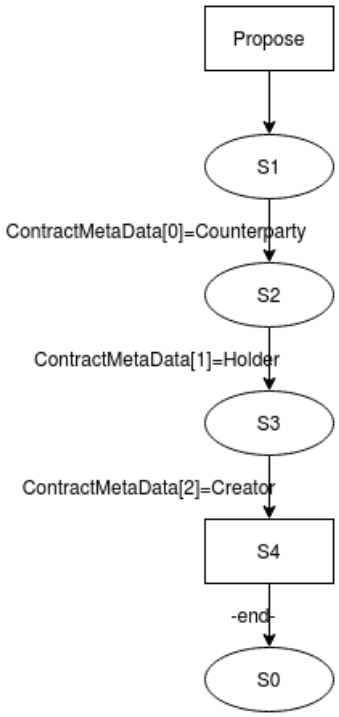


Figure 5.2.4: Trace of the Propose function in the proposed solution

5.3 Evaluation

In order to evaluate the results obtained, the evaluation was done in two phases. Evaluation of the two research questions has been done to make sure the success

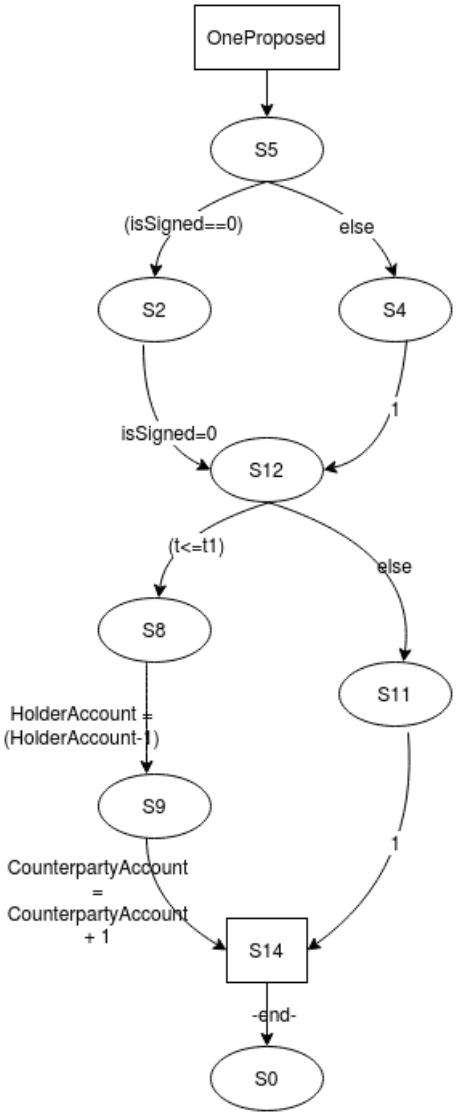


Figure 5.2.5: Trace of the sign function in the proposed solution

of the proposed solution. Evaluation has been done in terms of optimization and the correctness.

5.3.1 Optimization

The optimization of the proposed solution has been evaluated in terms of the cost of the execution. The execution cost on the Ethereum bockchain is measured in unit of gas on the Ethereum blockchain. The comparison of the two approaches proposed in this research has been compared to each other and the proposed solution has been compared with a previous approach. The transformed contracts were executed and tested using Remix IDE.

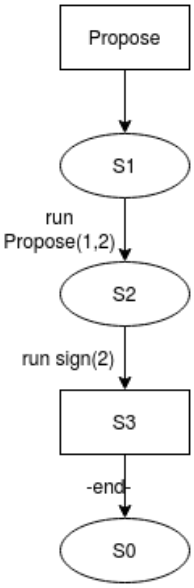


Figure 5.2.6: Trace of the init function of the promela code

The gas estimation for a contract is done beforehand and then the adequate number of gas units are specified for a contract. If the execution runs out of gas before completion, the contracts get void and the gas is lost. The executor cannot regain the gas spent. Therefore, it is critical to specify adequate number of gas units for execution completion.

The cost of a contract in terms of physical money can be calculated as follows. As of January 2020, the price of the Ether is approximately $167. Additionally, the cost per unit of gas is approximately 1.1GWei (2 $\times 10^{-}9$ Ether). Let's assume, a particular contract to be executed cost 1797270 gas.

$$2 \times 10^{-}9 \frac{Ether}{gas} \times \frac{\$167}{Ether} \times 1797270 gas = \$0.6002 \tag{5.1}$$

Remix, an in-browser Solidity compiler and blockchain simulator was used in this study to compile, deploy and measure the contracts. All contracts were compiled with the Solidity compiler –optimize flag enabled. Remix reports both the transaction and the execution cost of a contract. The Figure 5.3.1 shows the comparison of transaction costs of different actions of two approaches of proposed solution(Named as approach 1 and approach 2) and the previous approach(Synergy). Approach 1 of the proposed solution is the approach with the external contract Marketplace and the approach 2 is the approach without the external contract

Marketplace. When comparing the cost, the transaction cost is considered, as the execution cost is also included in the transaction cost.

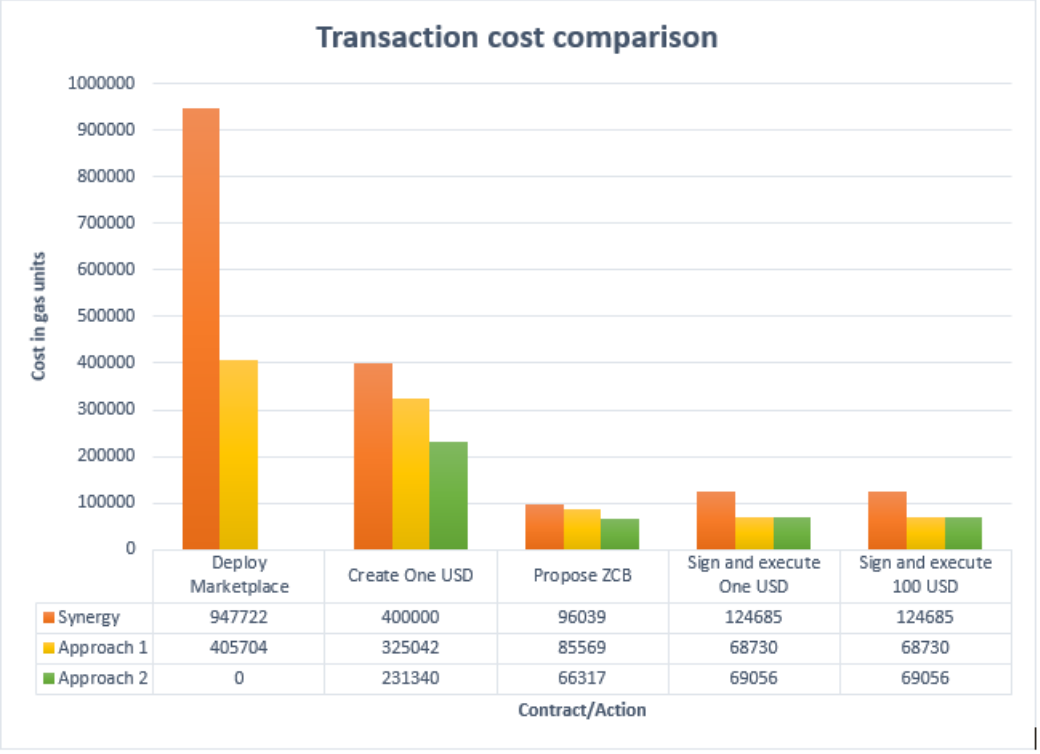


Figure 5.3.1: Comparison of transaction costs in terms of gas units on the Ethereum blockchain

By analysing the cost of each part in the contracts, it is understood that the composibility nature of the contracts is preserved in the cost calculation as well.

5.3.2 Correctness

In order to check the correctness, a basic verification that can be done is the semantic comparison of the two contracts, contracts in Peyton Jones' CDL and contracts in the proposed solution. So for that, the comparison is done between a financial contract written in natural language, a financial contract represented in Peyton Jones' CDL and the contract which was trasformed to a smart contract.

Zero Contract

- Natural Language - No transactions
- Peyton Jones' CDL - 'zero'

- smart contract code snippet

```
1  pragma solidity ^0.5.10;
2  pragma solidity ^0.5.10;
3 pragma experimental ABIEncoderV2;
4 contract A {
5    address public ContractMetadata;
6        uint isSigned = 0;
7        uint public USDbalances;
8        uint public GBPbalances;
9        address creator;
10       uint today = now;
11       function propose(address counterparty, address holder)
    public{
12            assembly{
13                  sstore(ContractMetadata_slot , counterparty)
14                  sstore(add(ContractMetadata_slot, 0x20), holder)
15                  sstore(add(add(ContractMetadata_slot, 0x20), 0
    x20), sload(creator_slot))
16           }
17       }
18       function sign(address holder) public{
19            assembly{
20                  if eq(sload(add(ContractMetadata_slot,0x20)),
    holder) {
21                     for{} eq(sload(isSigned_slot),0) {} {
22                           sstore(isSigned_slot,1)
23                     }
24                  }
25                  let expireDate
26                  let balanceHolder
27                  let balanceTo
28                  selfdestruct(ContractMetadata_slot)
29            }
30       }
31 }
```

The 'zero' contract is one of the basic contracts in the financial contract world. It simple says do nothing. In order to define this contract which does nothing, Pey-

ton Jones' et al. has introduced the 'zero' combinator. The transformed assembly code of this basic combintor does not execute any transaction, it only performs the propose and the sign functions of the contract which is an essential thing in the contract. It simply kills the contract as soon as the it executed the sign function with no transaction happening. There is no action as a result of this contract. Therefore, the 'Zero' combinator and the 'Zero contract in Assembly' functions in the same manner which make those semantically equivalent.

It is important to note that, as this contract does not involve any transactions and it will not result with any changes to the parties related to the contract, the contract is included with the propose and sign functions. The reason for this is even though this contract does not provide any result, in a case of a sequential or inter dependent contracts, the execution of this zero contract might lead to another contract. The chain of execution will be break at the middle if this contract is not executed with these important functionalities. Therefore, inclusion of the propose and sign functions regardless of the cost for the execution of them is very important when writing these contracts.

One Contract

- Natural Language - Receive $1 immediatly
- Peyton Jones' CDL - 'one USD'
- smart contract code snippet

```
1  pragma solidity ^0.5.10;
2   pragma experimental ABIEncoderV2;
3
4      contract A{
5          address public ContractMetadata;
6          uint isSigned = 0;
7          uint public USDbalances;
8          uint public GBPbalances;
9          address creator;
10         uint today = now;
```

```
11
12      function propose(address counterparty, address holder)
    public{
13          assembly{
14              sstore(ContractMetadata_slot , counterparty)
15              sstore(add(ContractMetadata_slot, 0x20), holder)
16              sstore(add(add(ContractMetadata_slot, 0x20), 0x20)
    , sload(creator_slot))
17          }
18      }
19
20      function sign(address holder) public{
21           assembly{
22              if eq(sload(add(ContractMetadata_slot,0x20)),
    holder) {
23                  for{} eq(sload(isSigned_slot),0) {} {
24                      sstore(isSigned_slot,1)
25                  }
26              }
27          let expireDate := "1569868200 "
28
29
30          if lt(sload(today_slot),expireDate) {
31              let balanceHolder :=  add(sload(USDbalances_slot)
    ,1)
32              let balanceTo := sub(sload(add(USDbalances_slot,0
    x20)),1)
33              sstore(USDbalances_slot,balanceHolder)
34              sstore(add(USDbalances_slot,0x20),balanceTo)
35              }
36              selfdestruct(ContractMetadata_slot)
37
38          }
39      }
40    }
```

The one contract is the other basic contract among financial contracts. The semantic of this contract is for the counterparty to receive $1 immediately from the holder of the contract. For this, Peyton Jones et al. have introduced the

'one' combinator and the contract is written as 'one USD'. When the transformed assembly code of this contract is executed, the 'sign' function of the contract is executed by the counter party after the holder is proposed, When the 'sign' function is called by the counter party, the contract transfers $1 from the holder's account to counter party's account. Therefore, the behavior of the transformed contract is the same as how the semantics suggests in the Peyton Jones' CDL. As such, the transformed 'one contract' is semantically equivalent to the Peyton Jones' one contract.

Contract Component (i.e.: Zero Coupon Bond)

- Natural Language - Receive $100 on the 20th of March 2020
- Peyton Jones' CDL - 'get (truncate "20 03 2020" (scale 100 (one USD)))'
- smart contract code snippet

```
1 let balanceHolder :=  add(sload(USDbalances_slot),1)
2 let balanceTo := sub(sload(add(USDbalances_slot,0x20)),1)
3 sstore(USDbalances_slot,balanceHolder)
4 sstore(add(USDbalances_slot,0x20),balanceTo)
```

As explained in chapter 3, the basic contract can consist one or more combinators from one, scale, truncate and get/give. A zero coupon bond is one such contract which includes all these combinators. The semantics of this particular contract states that $100 should be received by the counterparty at the specified future date. The transformed assembly code of this particular contract transform 100 USD to the counterparty on the said date when 'sign' function is invoked afte the calling of the 'propose' functionaity. Therefore, the semantics stand correct for the basic contract component introduced as well.

Complex Cotract (i.e.: A contract with operator 'and')

- Natural Language - Receive $100 on the 20th of March 2020 and give $10 on the 5th of April 2020
- Peyton Jones' CDL - 'get (truncate "20 03 2020" (scale 100 (one USD))) 'and' give (truncate "5 04 2020" (scale 10 (one USD)))'

- smart contract code snippet

```
1  pragma solidity ^0.5.10;
2  pragma experimental ABIEncoderV2;
3  contract A {
4     address public ContractMetadata;
5         uint isSigned = 0;
6         uint public USDbalances;
7         uint public GBPbalances;
8         address creator;
9         uint today = now;
10        function propose(address counterparty, address holder)
     public{
11              assembly{
12                    sstore(ContractMetadata_slot , counterparty)
13                    sstore(add(ContractMetadata_slot, 0x20), holder)
14                    sstore(add(add(ContractMetadata_slot, 0x20), 0
     x20), sload(creator_slot))
15             }
16        }
17        function sign(address holder) public{
18              assembly{
19                    if eq(sload(add(ContractMetadata_slot,0x20)),
     holder) {
20                       for{} eq(sload(isSigned_slot),0) {} {
21                             sstore(isSigned_slot,1)
22                       }
23                    }
24                    let expireDate
25                    let balanceHolder
26                    let balanceTo
27                    expireDate := 1580668200
28                    if lt(sload(today_slot),expireDate) {
29                         balanceHolder :=  add(sload(
     USDbalances_slot),100)
30                         balanceTo := sub(sload(add(USDbalances_slot
     ,0x20)),100)
31                         sstore(USDbalances_slot,balanceHolder)
```

```
32                          sstore(add(USDbalances_slot,0x20),balanceTo
    )
33                          }
34                    expireDate := 1588530600
35                    if lt(sload(today_slot),expireDate) {
36                          balanceHolder :=  add(sload(
    USDbalances_slot),10)
37                          balanceTo := sub(sload(add(USDbalances_slot
    ,0x20)),10)
38                          sstore(USDbalances_slot,balanceHolder)
39                          sstore(add(USDbalances_slot,0x20),balanceTo
    )
40                          }
41                          selfdestruct(ContractMetadata_slot)
42              }
43        }
44 }
```

A complex contract is combined with one or more operators(and/or). The semantics of a contract which has two basic contracts combined by "and" implied that both basic contract components should be executed. The transformed assembly code behaves in a way, where both contracts are executed. When the contracts are transformed, the each basic contract is transformed one after the other to a single contract file. This enables the sequential execution of the transformed contracts in their appropriate days. This replicates the functionality of the 'and' combinator introduced by Peyton Jones' CDL. Therefore, two basic contract components tied together with an 'and' behaves as expected in par with the representation of Peyton Jones' CDL. As the semantics of the proposed contract and the Peyton Jones' CDL are equal, it has been shown that the composable nature of the Peyton Jones' CDL has preserved through the compiler built in this study.

The correctness of the proposed solution was verified formally by the formal verification for the traces develop for the proposed solution and the Peyton Jones' CDL. Trace Equivalence has been used for the formal verification.

The formal verification of some of the contracts in Peyton Jones' CDL and proposed solution, and their relevant traces are as follows. (In the figures the left figure shows the trace for the Peyton Jones' CDL and the right figure shows the

trace for the proposed solution for the particular contract in Peyton Jones' CDL.) Please note in the verification, CPA is referred to CounterPartyAccount and HA referred to HolderAccount.

Zero Contract

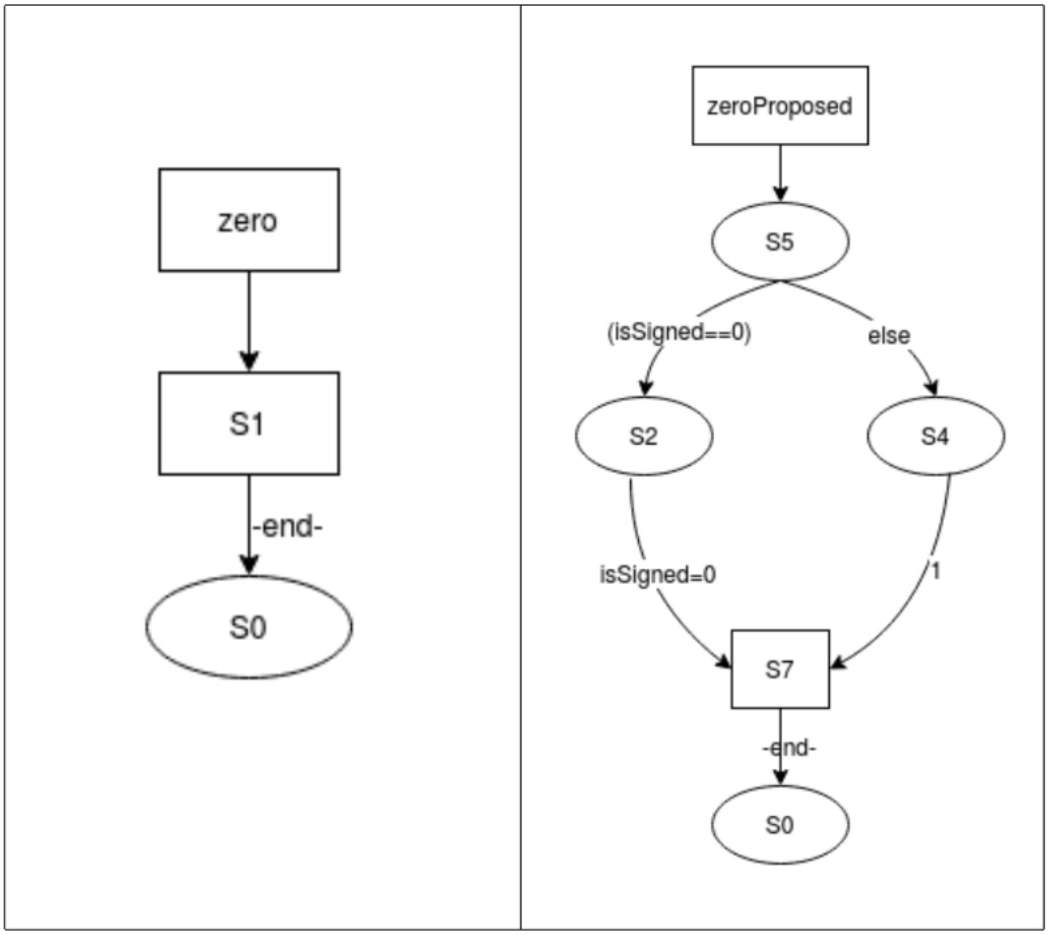


Figure 5.3.2: Traces of Zero Contract

Verification for Zero Contract

$$\begin{aligned} Traces(s1) &= \{end\} \\ Traces(s7) &= \{end\} \\ Trace(s1) &= Trace(s7) \end{aligned} \tag{5.2}$$

$$Therefore, we can say,$$

$$Trace(zero) = Trace(zeroProposed)$$

One Contract

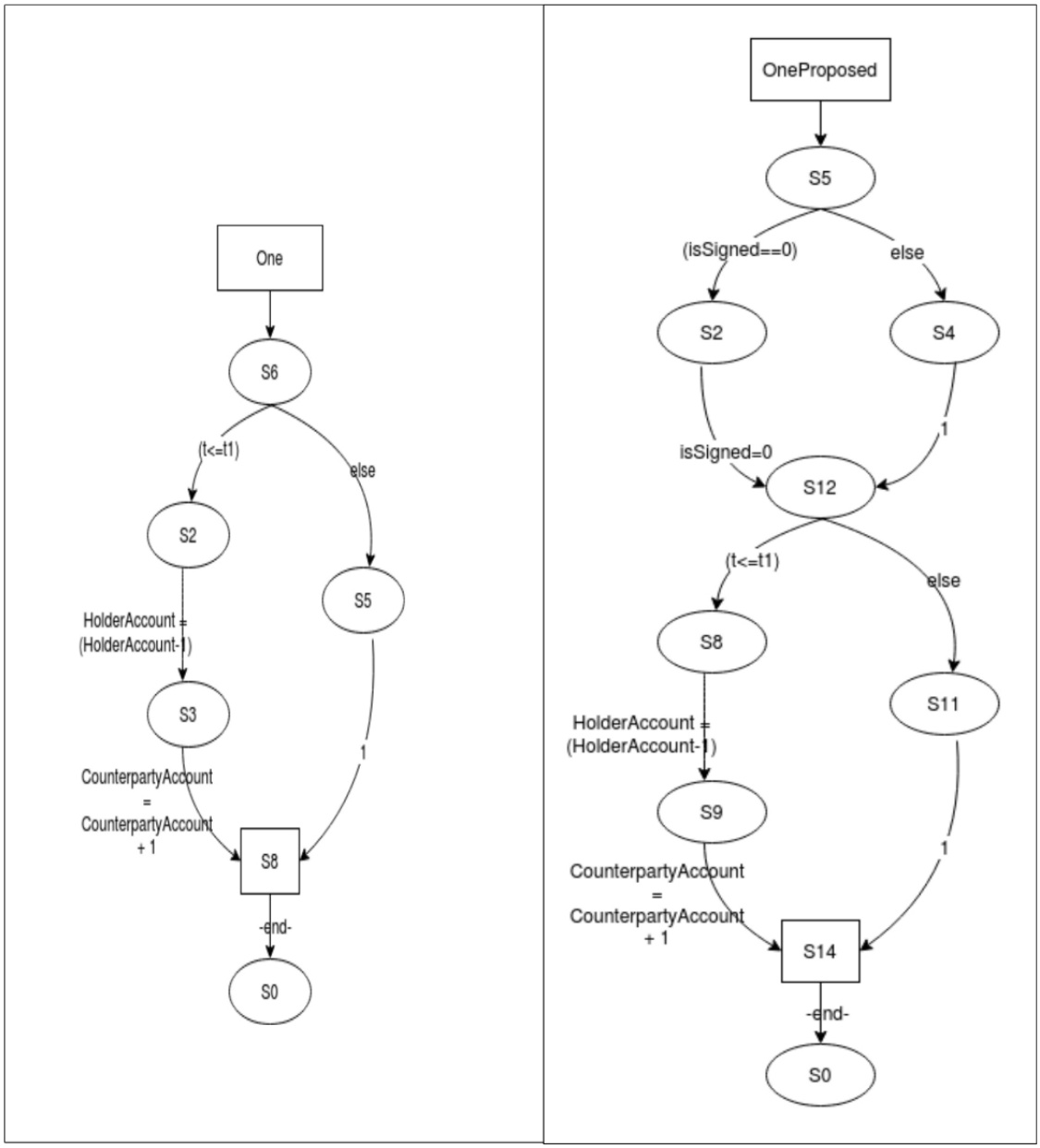


Figure 5.3.3: Traces of One Contract

Verification for One Contract

$$Traces(s6) = \{(t <= t1), CPA, HA, else, 1, end\}$$

$$Traces(s12) = \{(t <= t1), CPA, HA, else, 1, end\}$$

$$Traces(s6) = Traces(s12)$$

$$Trace(s2) = \{CPA, HA, end\}$$

$$Trace(s8) = \{CPA, HA, end\}$$

$$Traces(s2) = Traces(s8)$$

$$Trace(s3) = \{HA, end\}$$

$$Trace(s9) = \{HA, end\}$$

$$Traces(s3) = Traces(s9)$$

$$Trace(s5) = \{1, end\}$$

$$Trace(s11) = \{1, end\}$$

$$Traces(s5) = Traces(s11)$$

$$Trace(s8) = \{end\}$$

$$Trace(s14) = \{end\}$$

$$Traces(s8) = Traces(s14)$$

$Therefore, we can say,$

$$Trace(One) = Trace(OneProposed)$$

Contract Component(i.e. Zero Coupon Bond)

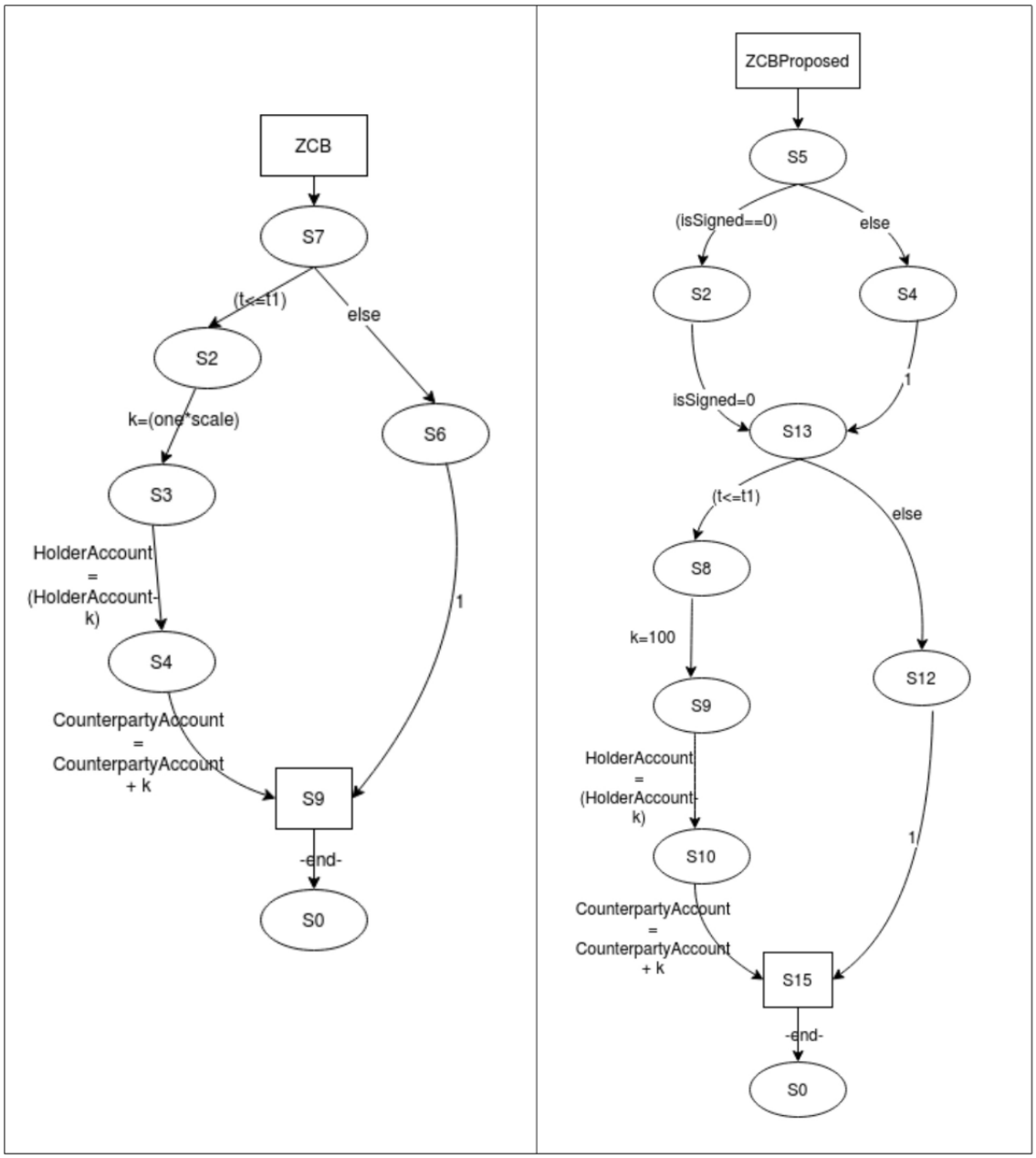


Figure 5.3.4: Traces of ZCB Contract

Verification for ZCB Contract

$$Trace(s7) = \{(t <= t1), k, CPA, HA, else, 1, end\}$$
$$Trace(s13) = \{(t <= t1), k, CPA, HA, else, 1, end\}$$
$$Traces(s7) = Traces(s13)$$

$$Trace(s2) = \{k, CPA, HA, end\}$$
$$Trace(s8) = \{k, CPA, HA, end\}$$
$$Traces(s2) = Traces(s8)$$

$$Trace(s3) = \{CPA, HA, end\}$$
$$Trace(s9) = \{CPA, HA, end\}$$
$$Traces(s3) = Traces(s9)$$

$$Trace(s4) = \{HA, end\}$$
$$Trace(s10) = \{HA, end\}$$
$$Traces(s4) = Traces(s10)$$

$$Trace(s6) = \{1, end\}$$
$$Trace(s12) = \{1, end\}$$
$$Traces(s6) = Traces(s12)$$

$$Trace(s9) = \{end\}$$
$$Trace(s15) = \{end\}$$
$$Traces(s9) = Traces(s15)$$

$Therefore, we can say,$

$$Trace(ZCB) = Trace(ZCBProposed)$$

Complex Contract(i.e. A contract with operator 'and')

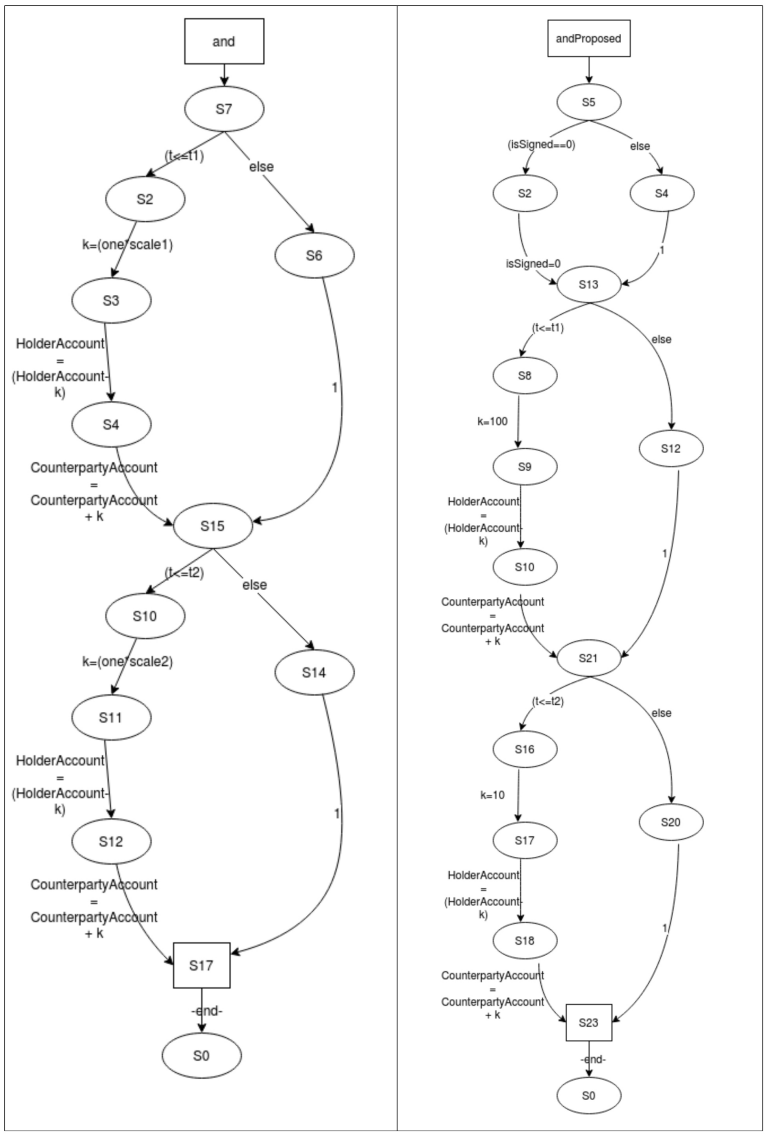


Figure 5.3.5: Traces of 'and' Complex Contract

Verification for 'and' Complex Contract

$$Trace(s7) = \{(t <= t1), k, CPA, HA, else, 1, (t <= t2), k, CPA, HA, else, 1, end\}$$

$$Trace(s13) = \{(t <= t1), k, CPA, HA, else, 1, (t <= t2), k, CPA, HA, else, 1, end\}$$

$$Traces(s7) = Traces(s13)$$

$$Trace(s2) = \{k, CPA, HA, (t <= t2), k, CPA, HA, else, 1, end\}$$

$$Trace(s8) = \{k, CPA, HA, (t <= t2), k, CPA, HA, else, 1, end\}$$

$$Traces(s2) = Traces(s8)$$

$$Trace(s3) = \{CPA, HA, (t <= t2), k, CPA, HA, else, 1, end\}$$

$$Trace(s9) = \{CPA, HA, (t <= t2), k, CPA, HA, else, 1, end\}$$

$$Traces(s3) = Traces(s9)$$

$$Trace(s4) = \{HA, (t <= t2), k, CPA, HA, else, 1, end\}$$

$$Trace(s10) = \{HA, (t <= t2), k, CPA, HA, else, 1, end\}$$

$$Traces(s4) = Traces(s10)$$

$$Trace(s6) = \{1, (t <= t2), k, CPA, HA, else, 1, end\}$$

$$Trace(s12) = \{1, (t <= t2), k, CPA, HA, else, 1, end\}$$

$$Traces(s6) = Traces(s12)$$

$$Trace(s15) = \{(t <= t2), k, CPA, HA, else, 1, end\}$$

$$Trace(s21) = \{(t <= t2), k, CPA, HA, else, 1, end\}$$

$$Traces(s15) = Traces(s21)$$

$$Trace(s10) = \{k, CPA, HA, end\}$$

$$Trace(s16) = \{k, CPA, HA, end\}$$

$$Traces(s10) = Traces(s16)$$

$$Trace(s11) = \{CPA, HA, end\}$$

$$Trace(s17) = \{CPA, HA, end\}$$

$$Traces(s11) = Traces(s17)$$

Trace(s12) = {HA,end }

Trace(s18) = {HA,end }

Traces(s12) = Traces(s18)

Trace(s14) = {1, end }

Trace(s20) = {1, end }

Traces(s14) = Traces(s20)

Trace(s17) = {end }

Trace(s23) = {end }

Traces(s17) = Traces(s23)

Therefore, we can say,

Trace(and) = Trace(andProposed)

5.4 Summary

This chapter detailed about the results of the proposed evaluation model for contract transformation from the financial contract domain to smart contract domain.The proposed evaluation method for the proposed method was explained in detail in this chapter. Formal verification of the proposed method was included in this chapter precisely. Comparison with the past method and the two methods proposed in this study was included in this section and the benefits of the proposed method was highlighted in this section.

Chapter 6 - Conclusion

6.1 Introduction

This chapter is comprised with a review of the research aims and objectives, research problem, limitations of the current work and implications for the further research.

6.2 Conclusion about the research questions

The main aim of this research was to optimize the autonomous smart contracts in terms of the gas cost required for a particular contract. After a thorough analysis of the cost structure and the cost evaluation of Ethereum contracts, it was understood that manipulating the data locations in the contracts would lead to the gas reduction of these Ethereum smart contacts. In order to achieve this, by gaining a control of the internal structures of the contracts, Assembly language was used.

The first research question was about the extent of the efficiency that can be increased when transforming contracts written in Peyton Jones' CDL to Assembly language. A compiler was built to transform contracts in Peyton Jones' CDL to Assembly and the compiler was enhanced to preserve the composable nature of the Peyton Jones' CDL. Finally the converted contracts were deployed to the Ethereum blockchain to check their functionality. After comparing the cost required for the proposed method with the previous method, it was possible to show that transforming the contracts to Assembly language has increased the efficiency of the contracts in terms of gas cost in a significant way. Through this, we managed to achieve the main objectivity and the first research question of this study.

The second research question was about the formal verification of the transformed

contracts. Formal verification make sure the properties of Peyton Jones' CDL has preserved in the proposed solution. By comparing the semantics of the Peyton Jones' CDL and the proposed solution, it was identified that the semantics were preserved in the transformed contract. By the formal verification, it was possible to prove the equivalence of the Peyton Jones' CDL and the proposed method formally. Further, the composable nature of the Peyton Jones' CDL was also preserved through the compiler built in this study as the transformed contract too portrayed composability. The composable nature of the contract enabled the formal verification of any given complex contract using the formal verification done to the basic contracts. It was possible to let the transformed contract execute at a future date through the Ethereum Alarm clock service which guaranteed the autonomous execution at a future date. In this way, we were enabled to achieve second research question of this study.

Thus, it can be concluded that the proposed solution in this study provides an optimized code through a proposed compiler and enables to formally verify the correctness of these provided codes.

6.3 Conclusion about research problem

The concern of the risk of involving the central counter party in the financial contracts have been eliminated with the approaches of building autonomous contracts, which did not need a central counter party to manage the transactions. There were many attempts in the past to re-implement the financial contracts in the Ethereum blockchain using solidity. However, they suffered from the cost that has to be incurred for the deployment and the execution of these contracts. The gas cost sustained on the contracts in Ethereum network might hinder the cost benefit that gained by eliminating the central counter party. This adverse effect of this gas cost might be in a higher degree for a contract which runs frequently and transact a small amount of money. Even more this might effect the correctness of the smart contract by running out of gas before the transaction completes. This was a clear research gap identified after conducting the literature review.

As the gas cost is an essential part in the ethereum network, it was impossible to

remove gas cost completely from the transactions. This study has contributed to the smart contract domain significantly by reducing the gas cost imposed on the smart contracts by transforming the smart contracts to assembly language rather than to solidity as in the previous approaches. Preparatory to building the compiler to serve the transformation, the assembly code was handcrafted in order to explore the instances where the optimization could embedded in. The code in the assembly has improved the accessibility of the internal data locations of the contract and the ability to manipulate those locations. The proposed solution of this study has contributed to the correctness of the smart contracts as well by reducing the chances of in-completed contracts due to insufficient gas balance.

A compiler was built in this study which is a significant computer scientific contribution. The compiler can be extended to transform financial contracts of different types. At the same time the compiler can be extended to transform contacts with different commodities. As mentioned above assembly was taken as the language for the transformed output of code.

The another problem that arise with this is the correction of the transformed contract with respect to Peyton Jones' CDL. As the contracts are transformed from Peyton Jones' CDL to Assembly, to make sure the proposed solution is correct even after using the optimization strategies, the proposed solution was compared with the Peyton Jones' CDL. To make a more formal comparison, equivalance of the Peyton Jones' CDL and proposed solution was considered in a formal manner. With the contribution of this study, it could be concluded that the smart contracts can be written in optimized manner while preserving its properties.

In summary, this study has proposed a compiler to convert financial contracts to smart contracts, explored the ways to optimize such contracts in order to reduce the gas cost associate with it and has proven that financial contracts suggested by Peyton Jones' CDL can be expressed in Ethereum smart contracts in assembly language in the similar manner.

6.4 Limitations

In the transformation from source language to assembly language, it was unable to completely remove solidity from the transformed code due to the unavailability of an assembler that converts solidity assembly language to EVM Bytecode, As the transformed contracts deploy and executed on the Remix IDE, it did not support standalone assembly to be deployed on that. The assembly language has to be included as inline to solidity, to be deployed in the Remix IDE. Even though the inclusion of solidity language was at its minimum level in this approach, if the entire code is written in assembly, cost reduction could have been achieved further.

The source language of this research is restricted to the Peyton Jones' CDL and did not explore any other DESL for financial contracts due to time constraint. Further, the language extension (of the Peyton Jones' CDL) done in this study was only for the combinators of the contracts. The combinators for observables weren't extended nor utilized for the transformation in the compiler. A readily usable, optimized compiler was not build through this.

As the contracts are deployed and executed on the Etherum platform, the extent to which a transformed contract is reliable, transparent and efficient depends on the facilitations of the Ethereum platform as well. Since it is still under development, certain limitations are imposed on our results as well. Some of thosee limitations are discussed below.

- Reliability of contract execution is mostly weighed upon the Ethereum platform. One major dependency of contract execution in this study was facilitating autonomous execution through the Ethereum Alarm Clock Service. This is a third party service which needs to be integrated explicitly as a separate smart contract. The lifetime of autonomous execution of contracts proposed in this study depends on how long this service would be maintained by the developers/ community.

- Efficiency is another major problem when it comes to financial contracts. Financial markets operate on the scale of milliseconds. However, operations on the blockchain cannot accommodate such speed efficiency.As of now, in the Ehtereum network it takes 10 to 20 seconds approximately for a block to be

mined and to be added to the blockchain. As there is no parallel processing in Ethereum mining, the maximum throughput of the Ethereum blockchain is approximately 15 transactions per second. By contrast, the NYSE(New York Stock Exchange) Group takes only 5 milliseconds to execute a trade. Therefore, the blockchain does not seem to facilitate the speed required by financial markets.

- Transparency is achieved on the blockchain by the distributed ledger system. Each peer on the network would be aware of all transactions and contract executions on the blockchain. However, since Ethereum addresses are not tied up with public identities, transparency in terms of who performs a transaction is limited. This creates a problem of liability on the network in terms of debt enforcement when executing contract logic.

Even though these limitations have minor impact on our proposed solution, further research and development is required to mitigate these in order to use such a system in a real financial market.

6.5 Implication for further research

6.5.1 Increasing the speed

The proposed solution does not give any concerns to the efficiency of the contracts in terms of the time that it require to execute the transactions. To make these solutions more usable in real world, when transformed from financial contract to autonomous smart contracts, the contracts need to be inline with the speed of the tradings in the financial markets. Despite financial markets trade in milliseconds, it takes 5 seconds to 15 minutes to process a transaction in Ethereum network if the standard gas price is being paid. The concept of gas price make the process time of a transaction in ethereum volatile. Therefore, the time taken to execute these smart contracts is a crucial thing which would require to be studied and developed in the future.

6.5.2 Insufficient account balance

In a situation where the holder's account balance is less than the amount to be send or in a situation where the holder's commodity balance is zero, the way the situation handled is not discussed in this study. After the commodity balance gets zero, there would be no commodities transferred to the counter party. Handling this type of situation in Ethereum blockchain is not a trivial task and it would require extensive further study to solve this issue.

6.5.3 AST Optimization

It was understood that optimization of gas cost could be achieved by optimizing the AST. By the AST optimization, it would be enable to reduce the number of contracts created for a particular input. It would lead to the further reduction of cost and increase the performance of contract execution on the blockchain. Even though the AST optimization is concerned and explored in this study, it was not implemented in this study due to time constraint. Further study and development on that would be able to explore many aspects of optimization.

6.5.4 Extending the compiler

This study and the previous studies concerned only the primitive combinators of Peyton Jones' CDL. As Peyton Jones' CDL consisted with primitive combinators and obeservables, the compiler could be extended to support observables as well. This is a key extension that will be required as most derivatives execute based on observable values.

Bibliography

[1] The coq proof assitent. Available at: https://coq.inria.fr/.

[2] Solidity documentation. Available at: https://solidity.readthedocs.io/en/v0.5.10/solidity-in-depth.html.

[3] Ethereum virtual machine opcodes[online]. Updated 2018-09-09. Available at: https://ethervm.io/.

[4] Xiaomin Bai, Zijing Cheng, Zhangbo Duan, and Kai Hu. Formal modeling and verification of smart contracts. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, ICSCA 2018, pages 322–326, New York, NY, USA, 2018. ACM.

[5] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. Accessed: 2016-08-22.

[6] Vincent Cheval. Apte: An algorithm for proving trace equivalence. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 587–592, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[7] Rocco De Nicola and Matthew Hennessy. Testing equivalence for processes. pages 548–560, 07 1983.

[8] Joost Engelfriet. Determinancy (observation equivalence = trace equivalence). *Theoretical Computer Science*, 36:21 – 25, 1985.

[9] Mounier L. Fernandez JC. A tool set for deciding behavioral equivalences. pages 587–592, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[10] Christopher Frantz and Mariusz Nowostawski. From institutions to code: Towards automated generation of smart contracts. *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 210–215, 2016.

[11] Jean-Marie Gaillourdet. A software language approach to derivative contracts in finance. *CEUR Workshop Proceedings*, 750, 01 2011.

[12] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA):116:1–116:27, October 2018.

[13] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. Kevm: A complete semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium*, pages 204–217. IEEE, 2018.

[14] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. pages 520–535, 04 2017.

[15] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[16] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanán, Yang Liu, and Jun Sun. Executable operational semantics of solidity. *CoRR*, abs/1804.01295, 2018.

[17] C.R. Kothari. *Research Methodology: Methods and Techniques*. Willey Eastern Limited, 1985.

[18] Magnus Oskar Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, UK, 2009.

[19] Zeinab Nehai, Pierre-Yves Piriou, and Daumas Frédéric. Model-checking of smart contracts. 07 2018.

[20] Rocco De Nicola. Behavioral equivalences. Dipartimento di Sistemi e Informatica, Universit'a degli Studi di Firenze Viale Morgagni 65, 50134 Firenze, Italia.

[21] Rocco De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.

[22] Russell O'Connor. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, PLAS '17, pages 107–120, New York, NY, USA, 2017. ACM.

[23] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A formal verification tool for ethereum vm bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 912–915, New York, NY, USA, 2018. ACM.

[24] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: An adventure in financial engineering (functional pearl). *SIGPLAN Not.*, 35(9):280–292, September 2000.

[25] Christopher Signer. Gas cost analysis for ethereum smart contracts eip-150 revision (759dccd - 2017-08-07), 2018.

[26] Matt Suiche. Porosity: A decompiler for blockchain-based smart contracts bytecode. *DEF con*, 25:11, 2017.

[27] V. U. Wickramarachchi. Efficiently transform contracts written in peyton jones contract descriptive language to solidity. January 2019. Thesis.

[28] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07), 2017. Accessed: 2018-01-03.

Appendices

Appendix A - Diagrams

get(truncate "10 2 2019" (scale 100 (one USD)))

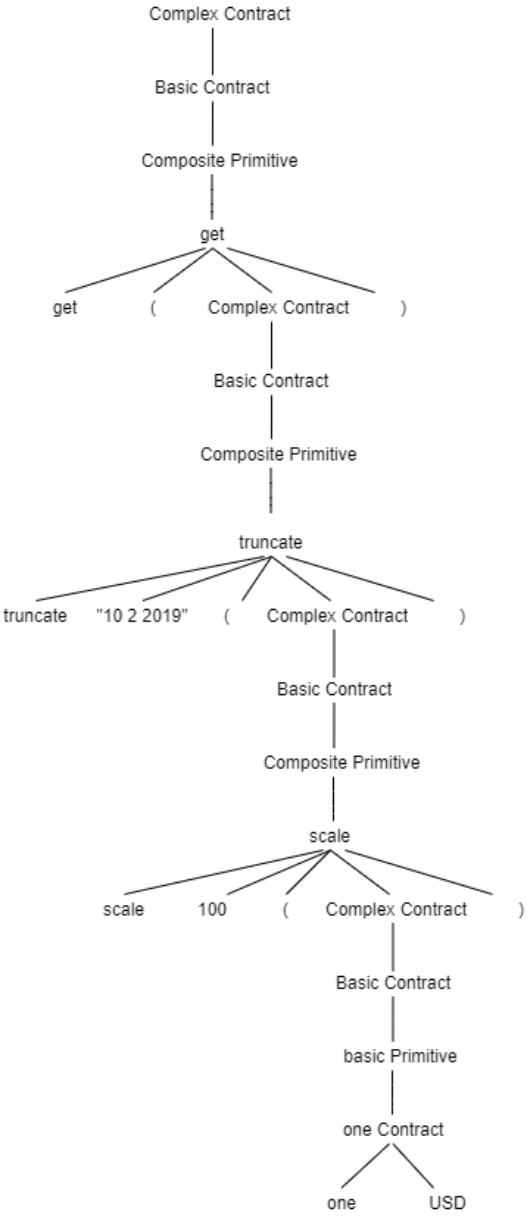


Figure A.0.1: Parse tree of a zero coupon Bond

get(truncate "10 2 2019" (scale 100 (one GBP))) 'and' get(truncate "10 5 2019" (scale 30 (one GBP)))

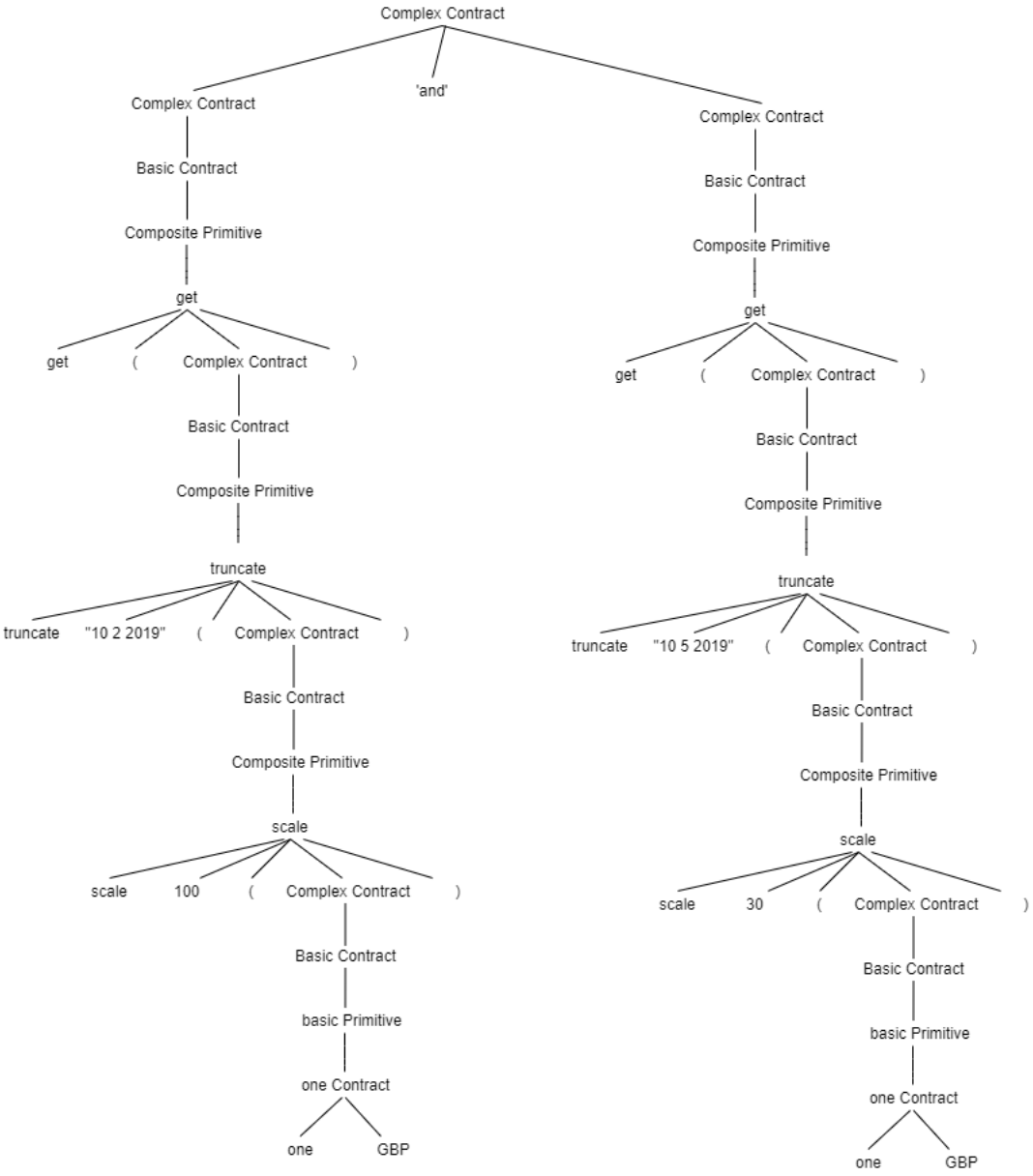


Figure A.0.2: Parse tree for a complex contract

Appendix B - Code Listings

```
1  const fs = require('fs');
2  let contract = ``;
3
4  function contractCreation(tree) {
5      let TimeStamp = "";
6      let list = [];
7      let contractName = "A";
8      let headers = `` +
9          `pragma solidity ^0.5.10;\n` +
10         `pragma experimental ABIEncoderV2;\n`;
11
12     fs.writeFileSync("./contractFiles/test.sol", headers, function
       (err) {
13         if(err) {
14             return console.log(err);
15         }
16         console.log("The file was saved!");
17     });
18
19     contract = contract + `contract ${contractName} {\n` +
20         `    address public ContractMetadata;\n` +
21         `        uint isSigned = 0;\n` +
22         `        uint public USDbalances;\n` +
23         `        uint public GBPbalances;\n` +
24         `        address creator;\n`+
25         `        uint today = now;\n`+
26         `        function propose(address counterparty, address
       holder) public{\n` +
27         `            assembly{\n`+
```

```
28          `                   sstore(ContractMetadata_slot , counterparty)\n`+
29          `                   sstore(add(ContractMetadata_slot, 0x20), holder)\n`+
30          `                   sstore(add(add(ContractMetadata_slot, 0x20), 0x20), sload(creator_slot))\n`+
31          `               }\n`+
32          `        }\n`+
33          `        function sign(address holder) public{\n`+
34          `             assembly{\n`+
35          `                   if eq(sload(add(ContractMetadata_slot,0x20)),holder) {\n`+
36          `                      for{} eq(sload(isSigned_slot),0) {} {\n`+
37          `                           sstore(isSigned_slot,1)\n`+
38          `                      }\n`+
39          `                   }\n`+
40          `                   let expireDate\n`+
41          `                   let balanceHolder\n`+
42          `                   let balanceTo\n`;
43
44      fs.appendFile("./contractFiles/test.sol", contract, function (err) {
45      if (err) {
46          return console.log(err);
47      }
48      console.log("The file was saved!");
49  });
50
51      list = recurse(tree, list);
52      console.log(list);
53
54    let closeBrackets = ``;
55    closeBrackets = closeBrackets +
56    `                   selfdestruct(ContractMetadata_slot)\n`+
57                      `             }\n`+
58                      `        }\n`+
59                      `}\n`;
60
```

```
61      fs.appendFile("./contractFiles/test.sol", closeBrackets,
        function (err) {
62       if (err) {
63           return console.log(err);
64       }
65
66       console.log("The file was saved!");
67       });
68  }
69  function recurse (tree, list) {
70       let tempList = [];
71       let currentKeyword;
72
73
74       if (tree.getChildCount() == 0) {
75           let nodeText = tree.getText();
76           if (nodeText === "(" || nodeText === ")") {
77               return list;
78           }
79           list.push(tree.getText());
80           console.log("list"+list);
81           return list;
82       }
83
84       for (let i = 0; i < tree.getChildCount(); i++) {
85           tempList = recurse(tree.getChild(i), tempList);
86           console.log(tempList);
87       }
88
89       for (let i = 0; i < tempList.length; i++) {
90           let keywordList = ['get', 'scale', 'one', 'zero', 'give',
        'truncate', 'then', 'anytime', ''and'', ''or''];
91           for (let j = 0; j < keywordList.length; j++) {
92               if (keywordList[j] === tempList[i]) {
93                   currentKeyword = tempList[i];
94                   console.log("CurrentKeyword"+ currentKeyword);
95               }
96           }
97       }
```

```
98      switch (currentKeyword) {
99          case 'scale': {
100             console.log("scale");
101             list.push(tempList[1]);
102             list.push(tempList[2]);
103             console.log("listScale"+ list);
104             return list;
105         }
106         case 'get': {
107             console.log("get");
108             let commodity = tempList[1].contractValue[1];
109             let quantity = tempList[1].contractValue[0];
110             let balanceSlot;
111             if (commodity == "USD"){
112                 balanceSlot = "USDbalances_slot";
113             }
114             else{
115                 balanceSlot = "GBPbalances_slot";
116             }
117             let contract = `` +
118             `                   expireDate := ${TimeStamp}\n` +
119             `                   if lt(sload(today_slot),expireDate)
    {  \n`+
120             `                       balanceHolder :=  add(sload(${
    balanceSlot}),${quantity})\n` +
121             `                       balanceTo := sub(sload(add(${
    balanceSlot},0x20)),${quantity})\n` +
122             `                       sstore(USDbalances_slot,
    balanceHolder)\n` +
123             `                       sstore(add(USDbalances_slot,0
    x20),balanceTo)\n`+
124             `                       }\n`;
125
126             fs.appendFile("./contractFiles/test.sol", contract,
    function (err) {
127                 if (err) {
128                     return console.log(err);
129                 }
130                 console.log("The file was saved!");
```

```
            });
            return list;
        }
        case 'give': {
            let commodity = tempList[1].contractValue[1];
            let quantity = tempList[1].contractValue[0];
            let balanceSlot;
            if (commodity == "USD"){
                balanceSlot = "USDbalances_slot";
            }
            else{
                balanceSlot = "GBPbalances_slot";
            }
            let contract = `` +
                `                   expireDate := ${TimeStamp}\n` +
                `                   if lt(sload(today_slot),
expireDate) {  \n`+
                `                       balanceHolder :=  add(
sload(${balanceSlot}),${quantity})\n` +
                `                       balanceTo := sub(sload(add
(${balanceSlot},0x20)),${quantity})\n` +
                `                       sstore(USDbalances_slot,
balanceHolder)\n` +
                `                       sstore(add(
USDbalances_slot,0x20),balanceTo)\n`+
                `                       }\n`;

            fs.appendFile("./contractFiles/test.sol", contract,
function (err) {
                if (err) {
                    return console.log(err);
                }
                console.log("The file was saved!");
            });
            return list;
        }
        case 'truncate': {
            let truncateObj = {};
            truncateObj.horizon = tempList[1];
```

```
164             truncateObj.contractValue = [tempList[2], tempList
    [3]];
165             let date = truncateObj.horizon;
166              TimeStamp = toTimestamp(date)
167             list.push(truncateObj);
168             return list;
169         }
170         case 'one': {
171             console.log("one");
172             return [tempList[1]];
173 
174         }
175         default: {
176             if(list.length > 0){
177                 for(let i=0; i<tempList.length;i++)
178                     list.push(tempList[i]);
179                 return list;
180             }
181             return tempList;
182         }
183     }
184 }
185 
186 function toTimestamp(strDate){
187  var datum = Date.parse(strDate);
188  console.log("print datum")
189  console.log(datum)
190  return datum/1000;
191 }
192 
193 exports.contractCreation = contractCreation;
```