A Unified Approach to Distributed Application Development for DLT

Ryan Falzon

Supervised by Prof. Gordon Pace

Co-supervised by Dr Joshua Ellul

Centre for Distributed
Ledger Technologies

University of Malta

September, 2021

A dissertation submitted in partial fulfilment of the requirements for the degree of M.Sc. in Blockchain and Distributed Ledger Technologies.

L-Università ta' Malta

WWW.UM.EDU.MT

First edition, Wednesday 15th September, 2021

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Gordon Pace and co-supervisor Dr. Joshua Ellul for their continual support and guidance throughout this research study. Additionally, I would like to express my gratitude to all lecturers of the Centre of Distributed Ledger Technologies for the knowledge they imparted on throughout their lectures.

Moreover, I would like to thank all my friends and family who in one way or another supported me during my journey. Above all, I would like to express my gratitude to my beloved Kylie for her moral assistance and words of encouragement.

Abstract

The widespread interest surrounding blockchain systems, has brought forth the introduction of decentralized applications. Such applications are built using Smart Contracts running on a blockchain network. Due to the siloed nature of blockchains and smart contracts, parts of such applications may have to be deployed on different blockchains, or outside the blockchain altogether. For instance, due to privacy constraints arising from GDPR, keeping private data on a public blockchain may not be an option, and would have to be kept on a centralized server which communicates with the blockchain in question.This shift in development methodology introduces new challenges for developers to achieve seamless communication and interaction between off-chain and on-chain code of decentralized applications. The current solution is to program the parts separately including additional code to handle communication between the different systems. Hence, this is considered as a source of additional complexity and also a potential source of error.

In this dissertation, we propose UniDAPP, a unified programming model to decentralized application development. We explored techniques that have been used to achieve blockchain interoperability, IoT enabled Smart Contracts, as well as the field of macroprogramming for wireless sensor networks. Our approach takes a macroprogramming approach, thus allowing for such systems to be programmed as a monolithic system, but with annotations to add information regarding where each part of the system should be deployed and executed. Ultimately, our aim is to create a development environment where developers can easily explore the placement of data and control flow on different target locations.

In order to demonstrate and evaluate the use of our approach we designed a software system use-case which would require shifting certain components between centralized and decentralized environments. The final results were made possible through the experiment carried out during the evaluation phase. This experiment included development of a number of tasks on the use-case using both the traditional method and the framework proposed herein.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

1

Introduction

1.1 | Overview

Blockchain technology, which is a type of Distributed Ledger Technology (DLT), offers a trustless environment for individuals without the need of any entity overseeing communication and actions. This is a result of the append-only data structure that such a network utilizes, thus making it an immutable ledger of transactions that is accessible by everyone. Data that is generated by the chain is not owned by just one person, but rather by everyone that participates within this network, which is a step forward when comparing it to traditional centralized systems that are owned by one entity (Yaga et al., 2019). DLTs can be categorized in one of two types, public permissionless ledgers, and private permissioned ledgers. The former allows virtually anyone to access and participate within the network while the latter requires users to be authenticated and given access to carry out any form of action within the network. For the scope of this dissertation we will be focussing the text on public chains.

The first widespread use case of this technology was Bitcoin, a protocol which was introduced by Satoshi Nakamoto. This allowed for the first peer-to-peer digital cash system that did not require a trusted party. The Bitcoin network allowed address holders to submit transactions to the network, indicating a form of payment to a recipient address, which would later be validated and appended within the next block in the chain (Nakamoto, 2008).

Instigated by the inclusion of executable on-chain contracts called Smart Contracts, blockchain technologies have attracted global interest. First coined by Szabo (1994), these smart contracts allow for self-executable pieces of code once certain conditions have been met, without the need for any third-party user intervention. Smart contracts are at the core of decentralized applications, more commonly referred to as dApps.

These types of systems typically consist of a user interface that can send and receive data to and from the back-end Smart Contract found on the blockchain. Just like participants, Smart Contracts are given an address when deployed to the network. This address is used to call Smart Contract functions by proposing a new transaction where the recipient is the Smart Contract address. All the security features offered by blockchain technologies are inherited by such systems, thus providing ease of mind that transactions carried out from such systems utilize the same level of protection that blockchain transactions have (Cong & He, 2019).

Systems composed of individual pieces of code being executed on different platforms are more commonly referred to as distributed applications. Adopting this practice meant that relatively complex systems were able to be split into smaller components executing on different platforms, thus leveraging the benefits of each separate platform. This brought forth a burden for developers as different execution platforms required different coding principles and languages, making it a difficult domain for novice developers.

In an effort to mitigate this problem, macroprogramming principles started being adopted for developing such systems. Such a technique allows developers to define a distributed system as a single program which will later be automatically decompiled down to the separate components that make up the whole system. However building a macroprogramming framework is non-trivial for various reasons. Primarily, such a unified model requires the abstraction of all features available by multiple platforms. Furthermore, the unified model needs to take into consideration how the different components will communicate with each other. Finally, achieving interoperability and heterogeneity within the unified model are also difficulties one might face.

1.2 | Motivation

Due to different approaches as to how smart contract code is executed, each blockchain platform has its own domain-specific instruction set and types. Solidity, for example, one of the smart contract languages for Ethereum, is a Turing complete, statically typed language that uses JavaScript-like syntax and a Gas mechanism for execution. A gas mechanism is one which forces users to pay for the execution of any Smart Contract interaction. Such a mechanism helps in reducing malicious code from consuming all Ethereum Virtual Machine (EVM) resources and leaving other transactions waiting for an available resource. On the other hand, Bitcoin Script, used for creating simple Smart Contracts for the Bitcoin Network, is not Turing complete and can be considered similar to Forth due to being stack-based (Tyurin et al., 2019). These constraints may limit the developers in terms of what code they can execute on-chain and may therefore need to move the execution of specific code blocks to a centralized off-chain environment.

Due to mass growth in interest that Smart Contracts have attracted, system operators are opting to shift to decentralized applications. Similar to distributed applications, decentralized applications have multiple execution locations, with the possibility to have Smart Contract code executing within a blockchain network. Such systems are adopted for several benefits including:

- Costs – As opposed to centralized systems, decentralized applications require users to pay a small fee otherwise known as Gas, in order to carry out executions. Therefore, to minimize the overhead that is created, developers can choose to have specific code blocks running off-chain.
- Security & Privacy – Blockchain networks utilize an architecture that prioritizes security above all else, to create what is known as trustless trust such that cryptographic procedures are at the forefront. However, due to regulations such as the Data Protection Act (2018), specific data should not be stored on-chain due to it being accessible by anyone participating within the network. By creating a decentralized system, developers can therefore choose to have personal data stored off-chain.
- Storage – In no way was blockchain created to replace centralized database systems. As it stands, blockchains are not able to store large datasets that many operators have. For instance, systems which allow users to upload files, cannot be created using only a Smart Contract platform. A decentralized system would allow developers to store the physical file off-chain in a centralized database, while uploading a file's hash on-chain, therefore allowing users to verify that their files were not, potentially, maliciously altered.

This shift in development methodology introduces new challenges for developers to be able to achieve seamless communication and interaction between off-chain and on-chain code of decentralized applications. Such challenges include:

1. First and foremost, identifying which segments of logic and data should be placed off-chain and which should be placed on-chain.
2. To achieve seamless communication between on-chain and off-chain code. On-chain code may not be immediately executed due to requiring time to reach con-

sensus in regards to its immutability within the canonical chain. Therefore this would require off-chain code to halt execution until the on-chain code is executed.

3. Data representation may differ between different software stacks and may require complex conversions to be carried out when moving data between on-chain and off-chain code.

4. The application would require to be coded in two or more different parts, most of the time using different technologies for each part, meaning that decentralized application development also incurs a learning curve for developers.

Consider Listing 1.1 that demonstrates the need to have a portion of the code running off-chain and another piece executing on-chain. Written in pseudocode, the function processes a new user within the system and continues with carrying out a background search on this user.

```
function ProcessNewUser(Person p)
{
  WriteOnChain(p.Id, p.Name, p.Surname, p.Role)
  WriteOffChain(p.Id, p.DateOfBirth, p.Salary)

  CarryOutBackgroundSearch(p);
}
```

Listing 1.1: Need to Have Off-Chain & On-Chain Code

From the code found in Listing 1.1, one can deduce the following:

- Date of Birth & Salary Variables - The values found within these variables should be stored off-chain because this is considered to be private user information. Therefore, due to the General Data Protection Regulation (GDPR), this data cannot be publicly shared.

- Name, Surname & Role Variables - These need to be stored on-chain to ensure immutability of these values.

- Background Search Function - This function should execute off-chain due to being computationally expensive. Mechanisms are set in place, Gas limits in the case of Ethereum, to discourage users from executing computationally expensive procedures on-chain as these would halt the execution environment from processing other transactions.

1.3 | Research Questions

This study will focus on answering the following research questions:

1. Whether a unified programming model framework can be applied to decentralized application development to address the identified challenges and provide programmers with the liberty to explore the movement of logical control-flow and data storage to on-chain and off-chain environments in a straightforward manner?

2. Will this abstraction process still impose any known or new difficulties for the developer?

3. Can these new difficulties be mitigated in some way or form? In doing so, we would be providing an in-depth comparative analysis between the traditional approach of building decentralized applications, and the one being proposed in this dissertation.

1.4 | Aims and Objectives

This dissertation aims to determine whether a unified approach to distributed application development reduces the overheads that need to be undertaken by developers when building such systems. In hope of achieving our aims, the following set of objectives need to be completed:

- Design a framework which would allow users to write, combined smart contract and system code.

- Build a decentralized application source generator that takes as input the unified smart contract and system code, and produces two separate codebases, one to be deployed off-chain and another to be deployed on-chain.

- Create a communication layer that will be used by the off-chain code to interact with and seamlessly execute on-chain code.

- Design and develop a real-world decentralized use case that will be used to evaluate the above. This use case should reflect everyday design decisions that such systems incur, such as storing only the objects' hash on-chain and the object itself in some form of centralized environment.

The use-case that would need to be developed using both traditional techniques, and also the proposed framework, shall involve a completely centralized system being shifted to a decentralized nature. However, due to issues such as GDPR, developers would have to opt for a hybrid system containing both. For instance, if we consider a social media platform such as Facebook, all their data is stored in a central database. This means that users have no way of verifying whether their data has not been maliciously altered. This can, however, be mitigated by storing a copy of the hashed data on-chain for future verifications.

The research questions will be evaluated through a process whereby a comparison will be made between code written in the traditional manner and code written using our framework. Software engineering metrics will be used to record overheads for both methods, including the number of lines added, deleted and modified, as well as the percentage of support and functional code. These metrics will be a result of the experiment taking place during the evaluation phase.

Any form of code created throughout this research is available through my GitHub page [1].

1.5 | Report Overview

The rest of this dissertation will be divided in the following manner. Chapter 2 provides a background to the domain being explored to appreciate the rest of the dissertation. Chapter 3 explains the design and implementation of the framework being proposed. Subsequently, in Chapter 4, this framework is evaluated in order to determine whether it answers the research questions outlined in this chapter. In chapter 5, related work is described and compared to what has been done in this dissertation. Finally, closing remarks and future work can be found in the final chapter.

[1] https://github.com/ryanfalzon/DLT-Dissertation

2 Background

This chapter provides an overview of the domains this research touches upon. With respect to blockchain technologies, an introduction to the underlying architecture and mechanism is given, as well as an analysis of smart contract applications. It was decided to position this research of a unified programming model for decentralized applications in the context of macroprogramming. Thus, a general overview of macroprogramming and aggregate programming is provided together with the challenges that one might face when using such approaches. The aforementioned will provide the reader substantial knowledge to understand technical approaches used during the design and implementation of the proposed framework.

2.1 | Blockchain

2.1.1 | Overview

Ever since Satoshi Nakamoto presented Bitcoin as a solution to the double-spending problem in 2008, blockchain, a type of DLT, has drastically grown in popularity. Even though Satoshi Nakamoto did not create blockchain technology, he is responsible for conceptualizing the first blockchain (Narayanan et al., 2019). The earliest literature of a data structure that loosely resembles blockchain technology was in a computer science dissertation with the title "Computer Systems Established, Maintained, and Trusted by Mutually Suspicious Groups" by David Chaum in 1982 (Sherman et al., 2019). In 1991, Stuart Haber and W. Scott Stornetta proposed a cryptographically secured chain of blocks to store timestamps without being easily manipulated securely (Narayanan et al., 2019). To improve the system's speed and efficiency, they allowed blocks to support

Merkle Trees as a means of storage so that each block can hold more than one certificate (Bayer et al., 1993).

The work presented by Satoshi Nakamoto in his whitepaper describes a system where peers participating within a network can communicate and exchange value within the ecosystem without the need of any type of intervention from third-party intermediaries. Held within a cryptographically secure digital ledger accessible by virtually anyone within the network are the network transactions. This digital ledger takes the form of a continuously growing chain of blocks. A consensus algorithm is used to ensure that all transactions placed within a mined block are correct and valid (Nakamoto, 2008).

Blockchain owes thanks to cryptocurrencies, mainly Bitcoin, for the mass increase in interest it gained. However, blockchain has become something more significant than just a use case for cryptocurrencies. The introduction of smart contracts, which refers to having executable code within the network, has opened various other doors. Smart contracts facilitate transactions that, in a centralized nature, would require an intermediary to overlook the process whereby certain conditions must be met in order for the process to continue. In this case, the difference is that the intermediate entity is replaced with pieces of code located in the chain that automatically executes when predefined conditions evaluate to true (Raskin, 2016).

2.1.2 | Blockchain Architecture

A blockchain is composed of a chain of blocks linked together with a cryptographic link. This cryptographic link is none other than the hash of the previous block (Yaga et al., 2019). A hash is a unique output that results from a one-way unknown process applied on an arbitrary input. This implies that a hash is unique and that there exists no process one can follow to go back to the original input from its resulting output. By reasoning, the first block, otherwise known as the genesis block, does not have a parent block, thus having an empty previous block hash (Swan, 2015). Figure 2.1 provides a simple, blockchain architecture diagram.

Consider a malicious user trying to manipulate the data within a particular block α. Upon editing the slightest details from block α, its hash requires re-calculating. This implies that the next block β, whose previous block hash is that of block α, requires re-hashing. The process continues until the last block of the chain is re-hashed. This turns out to be a lengthy and challenging process for an individual to undertake and makes tampering with the blockchain not only near impossible but immediately detectable by other peers within the network.

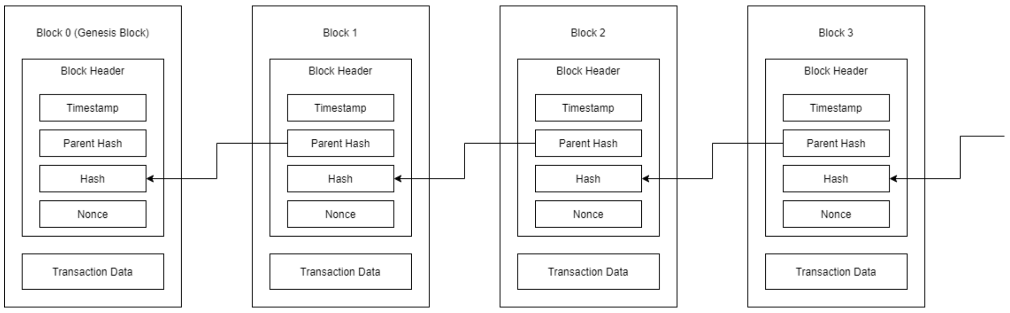


Figure 2.1: Blockchain Architecture

Amongst the various properties that one can find within a block header, one can find the parent hash, timestamp, nonce and consensus algorithm version. These can be considered as common between the numerous blockchain systems that are currently live. However, properties such as the consensus algorithm to validate blocks within the network and the access policy, are properties, amongst others, that differentiate one network from another. A vast range of consensus algorithms exist, and depending on the use case that a blockchain is being built for, an adequate consensus algorithm would be needed. The most used algorithm is the Proof-of-Work technique, in which miners need to provide performance power to safeguard the network and guarantee that the block they are submitting to the chain is valid. Being mentioned first in Nakamoto (2008)'s work, this approach consumes more energy due to a larger number of machines one would need in order to mine a block successfully. As time goes by, we are seeing the adoption of Proof-of-Stake algorithms more commonly now, just as Ethereum has adopted another version of their main chain to operate under a Proof-of-Stake environment. Miners enter a form of lottery and place their cryptocurrency at stake in such a mechanism, thus guaranteeing that a block is valid. When one considers blockchain access policies, two options are available, either a public or private network. Within a private network, only individuals who have access to the network can interact with the chain. In contrast, in a public network, anyone who downloads a copy of the ledger can view and submit transactions.

All in all, the blockchain data structure guarantees the integrity of the data found within each and every block through immutability, transparency and decentralization. By maintaining the previous block hash within every block, a cryptographic link is created whereby it makes it virtually impossible to manipulate data found in the network. Furthermore, the consensus mechanism ensures that peers within the network communicate within a trustless environment without the need for a central authority governing the network. Finally, all interactions occurring within the blockchain network is available to anyone through the digital ledger, thus providing transparency to its users.

2.1.3 | Smart Contracts

2.1.3.1 | Overview

The term Smart Contract was first coined by American computer scientist and cryptographer Nick Szabo in 1994. In his paper 'The idea of smart contracts' Szabo expressed the need to have a digital space, which allows for self-executable code in a trustless environment. He continued by defining a smart contract as a set of computer instructions shared with all network entities, having only interested parties agree to a set of predefined rules. These rules, or conditions as they are now more commonly referred to, are evaluated upon executing the contract, having a specific action occur if these conditions are satisfied (Szabo, 1994). Due to not having the right technology at the time, the idea remained dormant until recently. Together with the benefits that blockchain technology brings, the idea of Szabo's smart contracts enable self-verifiable, self-executable, and tamper-proof contracts.

A significant benefit that smart contracts bring forth as opposed to standard legal contracts is that once a smart contract is deployed to the blockchain, nobody can alter or remove that contract. Moreover, the ability to view the actual contract provides transparency to participants within the network of what the true nature of the contract is. All this is possible, thanks to the cryptographic nature utilized by blockchain networks which safeguards transparency through its append-only structure.

Consider a scenario where an individual is applying for a home loan from his local bank to be able to purchase a house. By nature, this is quite a lengthy process whereby the individual must go through a multistage process involving several different intermediaries, in order to get fund approvals and subsequently be able to purchase the house. Lack of trust between the involved parties can be considered as the root cause for this process. By leveraging the benefits of blockchain technology, smart contracts offer the trusted execution engine required to carry out such a transaction in the context of a trustless environment without the need of any intermediaries.

Depending on how blockchain networks operate, individuals who invoke smart contract functions may be subject to a fee to facilitate execution. More commonly known as Gas, this fee is set in place to allocate resources on the EVM to allow the code to be executed in a decentralized environment. Furthermore, Gas also ensures that execution is bound to finish, thus reducing the risk of having a malicious Smart Contract blocking other contracts to start execution.

2.1.3.2 | Programming Languages

Bitcoin Script Bitcoin Script is used to write scripts that can be stored on the Bitcoin network. The execution model for Bitcoin Script is more commonly referred to as the Unspent Transaction Output (UTXO) model. In this model, a smart contract is defined as a number of conditions that need to be met to spend the transaction value. Due to the limited number of operations that can be executed in such an environment, the types of applications are therefore also restricted. Bitcoin Script was designed as a non-Turing complete stack-based low-level language with reverse polish notation. With respect to syntax, Bitcoin Script loosely resembles Forth[1] (Moore & Leach, 1970) syntax such that any program terminates. Moreover, Bitcoin Script limits developers in terms of the script file size itself (Tyurin et al., 2019).

```
0 <Signature One> <Signature Two> 2 <Public Key One>
<Public Key Two> 2 OP_CHECKMULTISIG
```

Listing 2.1: Bitcoin Script Example

Listing 2.1 provides an example of a multisig transaction implemented in Bitcoin Script. Operations are processed from left to right, whose results are populated within a last-in-first-out stack. The above example requires two signatures, $< SignatureOne >$ and $< SignatureTwo >$ each originating from a different public key, $< PublicKeyOne >$ and $< PublicKeyTwo >$ respectively, to approve the transaction. To carry out this check, the $OP\_CHECKMULTISIG$ operation is used.

Solidity Solidity (Wood, 2014) is one of the most commonly used Smart Contract language for the Ethereum platform thanks to its simple code syntax. It is a high-level, statically typed, and object-oriented language whose syntax closely resembles that of the JavaScript[2] language. The contract architecture consists of a contract declaration that can have user-defined properties and functions. Reference from function declarations can be made to both locally defined properties and functions, as well as those present within other Smart Contracts. Ethereum smart contracts are compiled down to bytecode and executed on the Ethereum Virtual Machine. Limiting the duration of execution is a Gas mechanism where users are required to submit some cryptocurrencies to allow for execution. This Gas structure ensures two things; the first being assurance that validators will get paid even if something goes wrong during execution; the second

[1]First used in the 1970s, Forth is a procedural, stack-oriented programming language.
[2]JavScript is one of the core technologies used for the World Wide Web (WWW) and conforms to the ECMAScript specification.

is the fact that execution cannot exceed what is allowed through the prepaid amount (Parizi & Dehghantanha, 2018).

The following code shows an implementation of a multisig wallet smart contract in Solidity. One can see the resemblance that exists between Solidity and JavaScript. Furthermore, in comparison to the Bitcoin Script implementation above, one can appreciate the simplicity that Bitcoin Script offers in comparison to a Solidity Smart Contract.

```
contract MultiSigWallet {
  address payable one;
  address payable two;

  mapping(address => bool) signed;

  constructor(address payable _one, address payable _two) public
      payable {
    require(msg.value >= 0);
    one = _one;
    two = _two;
  }

  function sign() public {
    require (msg.sender == one || msg.sender == two);
    require (signed[msg.sender] == false);
    signed[msg.sender] = true;
  }

  function withdraw() public payable {
    require (signed[one] && signed[two]);
    require (address(this).balance > 0);
    uint amountToSend = address(this).balance / 2;
    one.transfer(amountToSend);
    two.transfer(amountToSend);
    signed[one] = false;
    signed[two] = false;
  }
}
```

Listing 2.2: Solidity Code Example

Marlowe Marlowe (Seijas & Thompson, 2018) is a Haskell embedded Domain Specific Language (DSL) that is used to implement smart contracts for the Cardano blockchain. However, just like modern programming languages, Marlowe is considered to be "platform-agnostic" and therefore, both UTXO and account-based platforms can benefit from the Marlowe programming model. By combining Haskell primitives with a number of basic constructs, programmers are given the ability to write complex contracts in a simple way. This is done by splitting a smart contract in smaller and reusable components which can communicate between them. Marlowe contracts are more notably used to model financial contracts in the style of Jones et al. (2000).

```
contract :: Contract

contract = When [Case (Deposit "alice" "alice" ada price) inner]
        10
        Close

inner :: Contract

inner =
  When [ Case bobChoice
        (When [ Case carolChoice
              agreement ]
            60
            Close)
    ]
    40
    Close

agreement :: Contract
agreement =
  If
    (bobChosen ‘ValueEQ‘ Constant 1)
    (Pay "alice" (Party "bob") ada individualPrice
      (Pay "alice" (Party "carol") ada individualPrice Close)
    )
    Close

choiceName :: ChoiceName
choiceName = "choice"

choice :: Party -> [Bound] -> Action

choice party = Choice (ChoiceId choiceName party)

```

```
bobChoice, carolChoice :: Action
bobChoice = choice "bob" [Bound 0 1]
carolChoice = choice "carol" [Bound 0 1]

bobChosen, carolChosen :: (Value Observation)
bobChosen = ChoiceValue (ChoiceId choiceName "bob")
carolChosen   = ChoiceValue (ChoiceId choiceName "carol")

defValue :: (Value Observation)
defValue = Constant 42

price :: (Value Observation)
price = Constant 450

individualPrice :: (Value Observation)
individualPrice = Constant 225
```

Listing 2.3: Marlowe Example

By keeping the same example as previously discussed smart contract languages, that of a multi sig wallet, one can see how Marlowe offers a totally different instruction set from what Solidity offers. In the above contract, when Alice deposits funds into the smart contract, the contract is locked. The contract can be unlocked only when both Bob and Carol agree, subsequently having the original deposited funds split between them.

2.1.4 | Chain Interoperability

The lack of interoperability that exists between current blockchain systems may end up being a problem in the future. Interoperability between blockchain systems is the ability to have cross-platform communication between different blockchain networks without the need for intermediate entities. In essence, such a mechanism could allow developers to create smart contracts on the Ethereum platform which allows users to purchase a share of a digital asset that is found in a consortium network such as Hyperledger and pay using Bitcoin. Some might consider this as the next step for blockchain technology through the creation of a number of sidechains designed using a defined DLT standard which are hooked to each individual blockchain network (Lima, 2018).

While Lima sees a DLT standard as being the solution to achieving chain interoperability, Vitalik Buterin stated that although chain interoperability is not easily achievable, this can be done through three stages. The first being that of centralized or multisig notary schemes. Notary scheme makes use of trusted entities to carry out specific actions or processes on chain B when something occurs in chain A. On the contrary, multisig notary schemes would require multiple entities to authorize the transaction. This is normally achieved by having the individual entities sign a message using their private key which can later be validated using their public key.

Using sidechains and relays is the second strategy that Buterin describes. Sidechains refer to when separate blockchains are attached to a parent blockchain using a two-way channel of communication. These sidechains act as an intermediary between the parent blockchain and any other non-linked blockchain to facilitate trustless communication between the two. Instead of relying on an intermediary to acquire information about another blockchain, relays refer to those blockchains that are specially built to handle such communication. This is a more direct approach as opposed to notary schemes which involve a certain level of trust.

Finally, Buterin explains how hashed time-locks can be used to facilitate atomic swaps in hopes of achieving interoperability. The notion of hashed time-locks is not new and has been one of the core properties of Bitcoins' Lightning Network deployed in 2019. Such a technique involves time-bound transactions and would require the recipient to acknowledge a transaction by creating a cryptographic proof within the specified time-frame, otherwise the transaction would be declined. Atomic swaps allow for swapping assets across different blockchain systems. A hash of a secret acts as a trigger to two different chains, which once revealed, sequentially unlocks the transactions on the respective chains (Buterin, 2016).

2.2 | Macroprogramming

2.2.1 | Overview

Back when the notion of high-level languages did not exist, developers were forced to create hand-coded programs that would execute directly on their machines. In addition to the performance and memory constraints that machines possessed at the time, developers quickly noticed how difficult it is to create and debug machine-level code, even with possession of substantial amounts of knowledge in the domain. This acted as a motivation to create what is now known as high-level languages. High-level languages are more straightforward to understand as they abstract programming details to a level that is closer to natural human languages. Furthermore, such languages allow developers to focus on what needs to be done instead of how the machine should do it. Each high-level language requires a compiler that translates the source code written in a high-level language to low-level machine code executed by the machine.

As application requirements started involving more computational power and specialized functionality that might not be readily available on a single system, developers began to utilize multiple machines for executing a single system. Doing so proved beneficial in terms of the features and functionality that each machine offered and also the benefits of utilizing multiple high-level languages to build a single application. By adopting such an approach, developers were able to build more robust systems. This technique is still used to this day where developers choose to have bespoke application code on both the client-side and server-side. In normal practices, client-side code tends to be less demanding than what is found on a server-side. This is done to cater for lower performance machines that clients might be running, more commonly referred to as end-user devices.

However, writing applications in this manner had its share of difficulties. Firstly, developers needed to create different parts of the application separately in different programming languages, which in the end, needed to run seamlessly together when deploying the solution. This increased the likelihood of bugs and indirectly made the process of debugging slower. Furthermore, communication layers would need to be created to facilitate exchanging messages between the different components. Therefore, a broader knowledge of programming capabilities needs to be possessed by the developer to carry out this form of development.

In the early nineties, a solution to the problems mentioned above was exploited: creating a unified programming model for developing software and hardware applications that span multiple systems. This meant that an individual with sufficient knowledge of the unified model and a good software development base could develop such applications using just the unified model and let the compiler decompile and build the individual components to run on the different environments. Moreover, developers did not need to worry about creating the intermediate communication layer between the system's different components (Page, 1996).

The terminology that describes this process, macroprogramming, was later coined by Newton and Welsh during the rise in wireless sensor networks' popularity. Their contribution relates to using macroprogramming to develop various sensor nodes using just one abstract programming language. Despite not being the first time that the term 'macro' was heard of, it was the first time that it was being used in this context [3]. In macroprogramming, the compiler acts as the defined rule by translating a line of code to its respective form that can be executed on its target platform (Newton & Welsh, 2004).

[3]A macro is defined as a rule or pattern that defines an input's mapping to a separate output.

2.2.2 | Aggregate Programming

While macroprogramming is more commonly used to refer to the creation of a unified programming model for wireless sensor networks, aggregate programming is the broader area of such a domain. Beal & Viroli (2016) describe aggregate programming as creating a layered solution to achieve a unified programming model. The first layer contains all the capabilities and features of each specific device, including communication medium, states and restrictions a device might have. By abstracting these software and hardware capabilities, a field calculus construct can be created. Subsequently, a set of resilient coordination operators are derived from the constructed calculus which is then made available through the fourth layer via user-friendly Application Programming Interfaces (APIs) that can be invoked by the developer to create the application code in the fifth and final layer. Beal et al. (2015) argue how this layered approach helps in hiding the complexities that programming Internet of Things (IoT) systems consisting of different edge devices has.

2.2.3 | Challenges of Macroprogramming

Level of Abstraction The first issue with creating a macroprogramming language is the level of abstraction that needs to be exhibited by the language. Therefore, in order to choose the correct model, one must take note of a number of differences that exist. The first being the physical attributes that lie with the languages to be abstracted. Such attributes include any memory or performance constraints that exist and also the fact that one system might support multiple types of nodes. Another set of differences that exist is related to how data is generated and processed, as this may differ from one system to another. Programmers should be allowed to toggle what happens with data at ease at the macroprogramming language level. The higher the level of abstraction, the more work must be overloaded to the runtime and the compilation framework. Thus, a balance is needed between the level of abstraction and the amount of processing the compilation framework should carry out. However, this should not hinder the process of making the unified model platform-independent to allow the developers to focus on what should be done rather than how it should be done (Pathak & Prasann, 2006).

Runtime System Design Heterogeneity of the systems is another issue that can be encountered. In the field of wireless sensor networks, this can be an issue as one can have devices communicating over Wi-Fi and others over Ethernet. On the other hand, in the blockchain domain, different networks use different methods of validation and verification of transactions and blocks. The runtime system should be able to consider this and carry out the necessary changes during runtime and not require the developer to include this in the macroprogramming code. Pathak and Prasann continue arguing that modularity is what makes a good runtime system as it helps in code generation and subsequently future updates and enhancements. A modular runtime design makes supporting more systems and improvements to the macroprogramming syntax simpler (Pathak & Prasann, 2006).

Code Generation As opposed to typical compilation carried out by a traditional compiler like javac (Used for Java), macro program compilation involves the generation of code for the individual systems which would later on be compiled using common compilers for binary retrieval. Because of this, this process is also referred to as code or source generation. This process is the root of many difficulties that may be experienced by developers when building macroprogramming languages, as source generation should not only translate the input code to the target code but rather make design decisions such as which data structure is best suited for each situation. Furthermore, it would be ideal to create a unified model for creating automatic unit tests for the generated code to ensure that adequate code coverage exists and no errors and bugs exist within the generated code (Pathak & Prasann, 2006).

Interoperability An essential component within a unified macroprogramming language is the communication layer that needs to exist between the different systems. If we consider the wireless sensor domain, devices within the network require a communication channel between each other. On the other hand, in the domain of distributed applications, the centralized components need to have a way to interact with components found on-chain and, in the future, the possibility to have different on-chain code communicate with each other. Mizzi et al. describe two forms of implementations for a communication channel; Either with the help of message queues or through a shared memory approach. The former makes the assumption that each system has a direct connection with the platform it needs to exchange messages with. On the other hand, the latter requires a portion of memory to be reserved for communication and acts as a form of shared memory which holds messages that need to be read by other systems. This approach, although it still uses message passing, raises the level of abstraction of the communication channel as no individual channels need to be built between the different systems supported by the unified model (Mizzi, 2019).

2.3 | Summary

A discussion and analysis were carried out on blockchain technology and macroprogramming in this chapter. With respect to the blockchain domain, an introduction was given, followed by a description of the architecture that such a technology adopts. Smart Contracts as a blockchain use case was also discussed together with a description of several smart contract programming languages such as Solidity and Bitcoin Script. Finally, background was given on macroprogramming and aggregate programming while also highlighting any challenges and issues that can be encountered by developers creating a unified programming model.

The aim of this chapter was to educate readers on the domains and technologies that will be used in the next chapters. Any subsequent work carried out builds on these technologies by making use of a macroprogramming approach to achieve a unified model for decentralized application development. To our knowledge, this is the first attempt of creating such a solution, however, similar approaches have been adopted in the field of wireless sensor networks. Different DLT dependent challenges were focused on achieving blockchain interoperability and easier development of IoT-enabled smart contracts. In Chapter 5 an overview of peripheral related work is provided to highlight their relevance to our work.

3

Design and Implementation

Applications developed in a decentralized manner make use of a mixture of programming languages and concepts to cater for the different locations, and their capabilities, where different components of the system reside. Components residing in different locations communicate together via API calls in order to facilitate message and data passing. This dissertation focuses on identifying a better approach to this development model to reduce both implementation and runtime overheads mentioned in the first chapter. To our knowledge, this is the first attempt at creating a unified macroprogramming model for this domain.

3.1 | Approach & Language Design

We aim at identifying a unified model that allows programmers to create combined source code, containing both off-chain code and on-chain code under one codebase as we believe that this is a more natural and easy way of programming an application. An aspect oriented approach was taken whereby code is tagged using some form of annotation to indicate which location framework it should be deployed to. These annotations will be used to tag whole classes and functions in scenarios where control-flow exists, but also individual variable properties when data-flow exists.

UniDAPP would in turn automatically generate the code for each respective execution location, having it ready to be deployed by the programmer. By using this approach, developers are able to experiment with shifting logic and data control code between different locations to achieve the best results in terms of costs and performance.

As will be discussed in subsequent sections, the source generator was designed in such a way as to have the framework easily and efficiently support other blockchain networks in the future. We see this as being a fundamental concept and requirement of

UniDAPP, as developers will be allowed to build decentralized applications that span across multiple networks, all while using one codebase.

Our ultimate goal is to create a framework which would abstract and hide the complexities that are coupled with decentralized application development.

3.2 | The Use Case

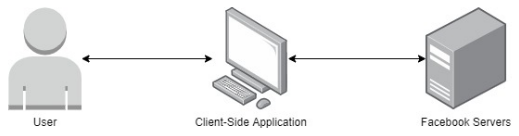
In order to evaluate the proposed framework, a use case needs to be identified. This use-case needs to include elements that would satisfy the criteria for a decentralized application and design decisions requiring different parts of the application to be deployed to different locations, i.e., off-chain and on-chain.

A social network application was chosen to serve as the use-case. Social networks such as Facebook and Instagram process client data on a daily basis, ranging from user interests to basic user information such as their date of birth. This often results in users being reluctant that their data is used for reasons that the client is unaware of or manipulated without the users' knowledge and consent. If one considers a scenario of two users, John and Mary, using the Facebook social network, where Mary works as a data engineer at Facebook. Figure **??** illustrates a post authored by John.

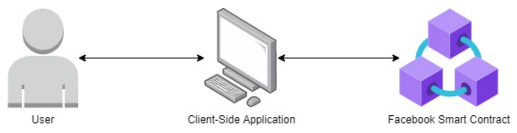
Given Mary's status and role at Facebook, she can access all user data found on the platform. Provided that Mary manages to bypass all security protocols that Facebook has to safeguard their data; in theory, she can change the content of John's post without his permission. Nowadays, organizations set up both technological and organizational guarantees to ensure that such scenarios never occur; however, we have seen cases where these have failed.

The described scenario can be overcome by shifting the social network on a decentralized application. However, due to the likes of GDPR, a Social Network operator should refrain from executing everything on-chain. This would mean that all the data will be available to virtually anyone, even individuals who are not social network members. This results from one of the core properties of public blockchain networks, whereby data is publicly available to anyone operating a node for the network. This issue causes the dilemma of central control versus decentralized, but trusted, execution. For a very long time, centralized intermediaries have been trusted with acting as an overseer and controller of our data and operations. However, with the introduction of DLTs, the need for a trusted centralized intermediary is removed and users are granted the ability to control their own data.

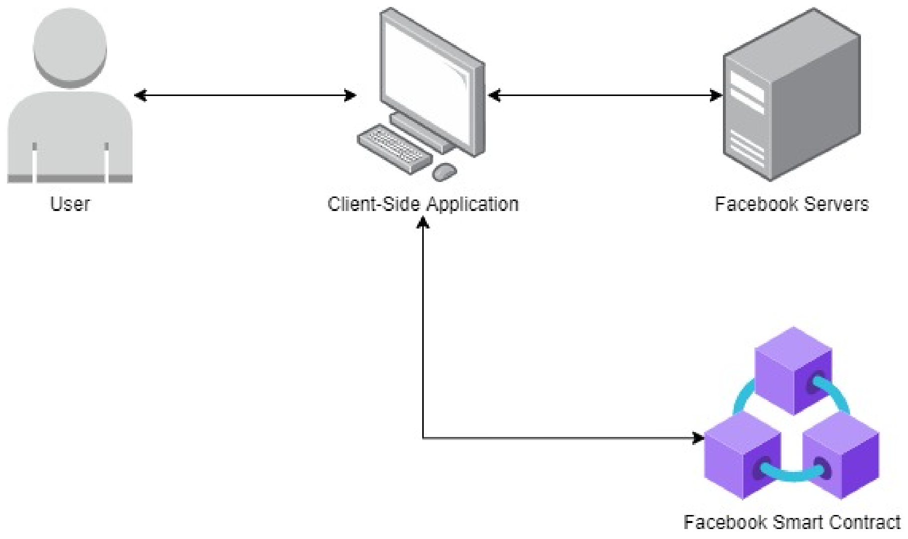
Having said this, social network operators will then have the option to migrate specific data properties and code execution to an off-chain environment. This would allow operators to comply with GDPR as sensitive client data would be available only to individuals enrolled in the social network, but still offering the benefits that on-chain smart contracts have. Architectural diagrams illustrating how these three types of systems differ can be seen in Figures 3.1(a), 3.1(b), 3.1(c). If such an approach were to be adopted, Mary would still be able to change the post authored by John. However now, John would be in a position to be able to show proof that his post has been changed through blockchain verification since the history is preserved on a publicly available ledger.



(a) Centralized System



(b) Decentralized Smart Contract System



(c) Hybrid System

Figure 3.1: Use Case Architecture

3.3 | Proposed Framework

Ellul & Pace (2019) proposed that annotations within a unified programming model allow developers to tag code fragments with the location where the said piece of code should execute. The idea here is that one program, written in one language, can be passed through a source generator to be processed down to separate code files. Ellul & Pace provide a way of annotating both control flow and data flow code segments in the same manner.

As we aim to reduce the overheads that developers are exposed to in such an environment, we propose UniDAPP, a framework developers can utilize to create decentralized applications using a unified programming model. Without deviating away too much from native C# syntax, three types of annotations were available for use; tag annotations, in-line annotations, and block annotations. Despite Ellul & Pace using one form of annotation for both control flow and data flow in their initial proposition, it became evident that we would be creating a more readable and understandable codebase by utilizing a mixture of these types of annotations.

Throughout this chapter, the term <LOCATION> is used in instances of annotation definitions and examples. This term refers to the location where the annotated code should execute. The current version of UniDAPP supports two locations, Desktop and Ethereum, meaning that developers can either execute code in a centralized desktop environment or on the Ethereum chain. One location should be passed within parenthesis within the annotation itself.

Figure 3.2 provides a high-level illustration of UniDAPP. One can see how the unified smart contract and system code is passed through a dApp source generator. After this file is parsed and processed accordingly, the centralized system code and the smart contract code are automatically generated and ready to be deployed. It is important to note that the communication channel illustrated in this figure is not necessarily a single communication channel, nor is it shared, it is just a means of exchanging messages between one system and another.

It is also important to note how UniDAPP can also be used on existing systems. In scenarios where the code for an existing centralized system coded in .NET is available, this can be used as a starting point and newer features can be added to this source code. This feature is made possible due to the fact that our source compiler is able to parse .NET code and interpret it accordingly.

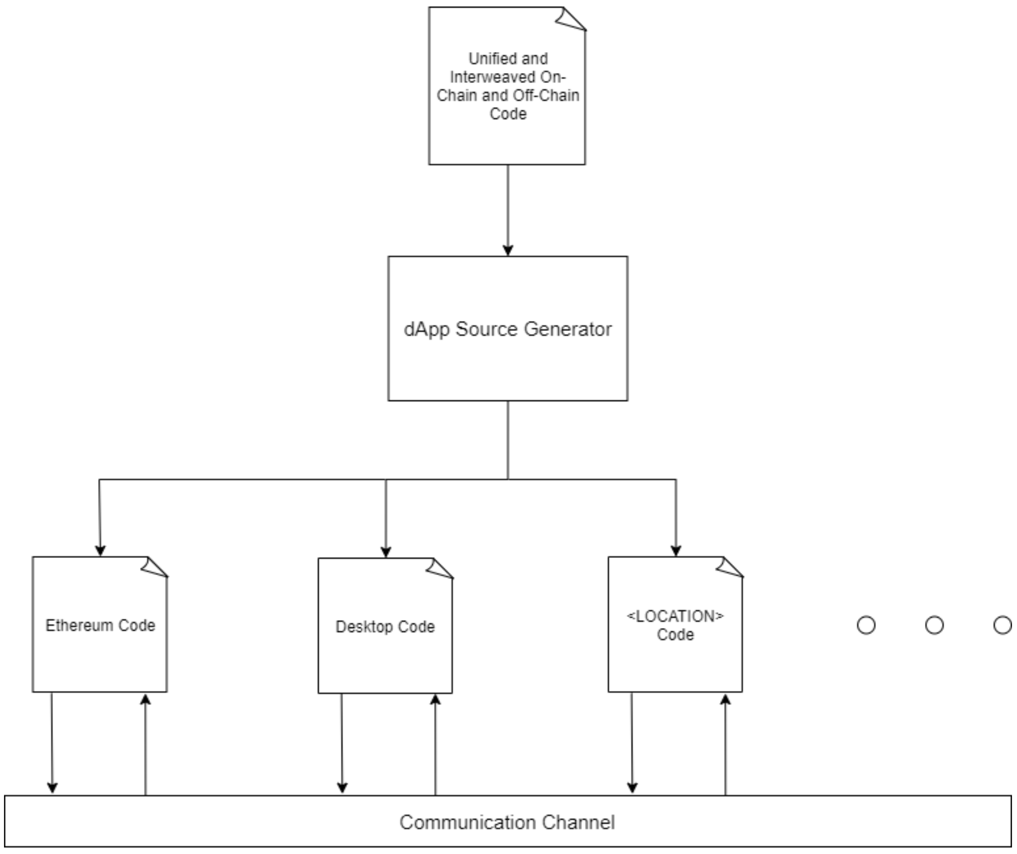


Figure 3.2: UniDAPP Framework

3.3.1 | Annotation Grammar

In this section, three annotations will be discussed; tag annotations, in-line annotations and block annotations. We will dive deeper in the annotations' grammar and semantics to understand the design decisions that were taken.

3.3.1.1 | Tag Annotations

These annotations highly resemble standard annotations found in the .NET languages, were an attribute enclosed within a set of braces is added on top of a line of code. This means that a specific action needs to be carried out on the expression found underneath the tag. These labels are intended to be used for data flow syntax, i.e. class, function and variable definitions and must follow the formal definition found below.

$\langle start\rangle$ ::= '`[XOn(`' $\langle location\text{-}definition\rangle$ '`)]`';

$\langle location\text{-}definition\rangle$ ::= $\langle location\rangle$ | '`OutOnly`';

$\langle location\rangle$ ::= '`Desktop`' | '`Ethereum`';

Such annotations provide a concise and straightforward manner to annotate class function and variable declarations. In the case of functional logic, such an annotation would not be ideal as it would require developers to annotate every line of code. During development, it became evident that when defining custom types, developers might require that certain properties are kept read-only, meaning that only retrieval of such variables would be allowed. Hence, in addition to a location definition, tag annotations within the context of our framework accept the '*OutOnly*' attribute. Such an attribute would instruct the source generator to exclude the annotated variable from any parameter listings. Listing 3.1 provides an example of using this annotation.

```
[XOn(All)]
public class Profile
{
  [XOn(All)]
  public int Id;

  [XOn(Ethereum)]
  [OutOnly]
  public Bytes32 Hash;

  [XOn(Desktop)]
  public string Name;

  [XOn(Desktop)]
  public string Email;
}
```

Listing 3.1: Tag Annotation Example

From the above code block, one can see that the Profile class will be available on all systems that the framework supports, which at the time of writing, are Desktop and Ethereum environments. Variables such as the name and email will be only available on desktop environments, hence ensuring that no private user information is available on the blockchain network. On the other hand, the profile hash is annotated to be stored on-chain and can only be read.

3.3.1.2 | In-Line Annotations

The use of in-line annotations was reserved for identifying classes that are custom-defined data objects. Using in-line annotations provides a clear and concise way of identifying data models from other system logic. Such annotations are required to follow the following grammar rules.

⟨*start*⟩ ::= '`XModel(`' ⟨*contract-name*⟩ '`)`';

Similar to the previously discussed annotation, this tag requires a parameter to be passed. This parameter should be the name of the Smart Contract, where the model should be placed. As multiple Smart Contracts definitions can exist within the framework, models would not be required to exist within all of them. Code Listing 3.2 below is an example of how one could go about using this form of annotation.

```
[XOn(All)]
public class Profile : XModel("SocialNetwork")
{
  ...
}

[XOn(Ethereum)]
public class SocialNetwork
{
  ...
}
```

Listing 3.2: In-Line Annotation Example

3.3.1.3 | Block Annotations

Finally, for annotating function scopes, block annotations were chosen as the ideal tagging structure over the previously discussed techniques. Block annotations offer the ability to gather several consecutive expressions into one scope, thus avoiding unnecessary overuse of annotations. Such annotations are required to follow the following grammar rules.

⟨*start*⟩ ::= '`@XOn(`' ⟨*location-definition*⟩ ⟨*parameters*⟩ '`) {`' ⟨*scope*⟩ '`}`';

⟨*location-definition*⟩ ::= '`Desktop`' | '`Ethereum`';

⟨*parameters*⟩ ::= '`, `' ⟨*parameter*⟩ | ⟨*parameters*⟩ '`, `' ⟨*parameter*⟩;

Finally, Listing 3.3 shows how one would go about using block annotations. It is important to note that as opposed to other class segments, functions do not require a preceding annotation. This allows developers to have full control of where each expression within a function is stored.

```
 1  [XOn(All)]
 2  public class SocialNetwork
 3  {
 4    [XOn(Desktop)]
 5    public IDatabaseConnector _databaseConnector;
 6
 7    $>mapping(int => string) public Profiles;
 8
 9    public void Register(Profile profile)
10    {
11      @XOn(Desktop, profile)
12      {
13        _dataBaseConnector.AddProfile(profile.Id, profile.Name, profile
              .Email);
14      }
15
16      ~@XOn(Ethereum, profile)
17      {
18        Profiles[profile.Id] = profile.Id.Hash();
19      }
20    }
21  }
```

Listing 3.3: Block Annotation Example

Just like previously discussed annotations, the location parameter indicates where the code found within the block annotation scope should be placed. This parameter is followed by a comma delimited list of parameters that indicate the input to the function created in the respective location. The '$\sim$' in line 11 will be discussed further down.

3.3.1.4 | Additional Aspects

Parent Inheritance The notion of parent inheritance works throughout all the described annotation structures. When code fragments are not annotated, the source generator reverts back to the annotation provided to its parent. This reduces the number of unnecessary annotations required to be written by the developer. In Listing 3.4 since all properties incorporated within the Profile class will be automatically placed on the Ethereum platform as per the annotation applied to the class declaration.

```
[XOn(Ethereum)]
public class Profile
{
  public int Id;

  public string Hash;
}
```

Listing 3.4: Parent Inheritance Example

Asynchronicity Additionally, another essential aspect of such annotations is the '$\sim$' symbol placed before block annotations. This symbol dictates whether the said block should be executed synchronously, in the case that the symbol is used, or asynchronously, if the symbol is not added. Synchronous execution refers to when processes are executed one after the other, which results in the caller system blocking until a response is received. On the other hand, asynchronous calls refer to when processes are executed in parallel. The 'orchestrator' system, i.e. the system that should make these synchronous and asynchronous calls, is set to be the first annotated location identified within a function scope. Therefore, if we consider Listing 3.3 and apply this reasoning, the Ethereum scope should be executed synchronously with respect to the Desktop system. Despite supporting asynchronous calls, due to time constraints there does not exist a way of reading the response and feedback retrieved via such calls. In such cases asynchronous calls are only used in cases where the rest of the code is independent of what happens within the asynchronous call. However, it may be the case that no response is received because something from the asynchronous call failed, thus, such calls might not be reliable to use.

Error Handling In terms of error handling, syntax errors are caught by the compilation stage of the source generator. When a syntax error occurs, the compiler execution fail and an informative message is returned to the user explaining while compilation failed. Due to time constraints, the scope of this research does not explore the scenario where business runtime errors occur when using a system built using UniDAPP. An example of such an error would be a require statement residing within a smart contract function requiring that a certain value is, for instance, larger than 0. In such a case, the current iteration of UniDAPP simply catches that error at the communication layer and continues execution to avoid having the application abruptly crash.

3.3.2 | Annotation Semantics

If we consider the code fragments found in Listings 3.1, 3.2 and 3.3 which were provided as an example in the previous sections, the below listings would be the resulting code generated by the framework. Listing 3.5 provides C# code generated for a Desktop environment, while Listing 3.6 provides Solidity code generated for an Ethereum Smart Contract.

```
public class Profile
{
  public int Id { get; set; }
  public string Name { get; set; }
  public string Email { get; set; }
}

public class SocialNetwork
{
  public IDatabaseConnector _databaseConnector { get; set; }

  public void Register(Profile profile)
  {
    _databaseConnector.AddProfile(profile.Id, profile.Name, profile.
       Email)
    await XCall("Ethereum", "SocialNetwork", "Register", profile.Id);
  }
}
```

Listing 3.5: Generated C# Code

```
pragma solidity >=0.4.22 <0.7.0;
contract SocialNetwork{
  struct Profile{
    uint128 Id;
    bytes32 Hash;
  }

  mapping(uint128 => bytes32) public Profiles;

  function Register(uint128 id) public{
    bytes32 hash = keccak256(abi.encodePacked(id));
    Profiles[Id] = hash;
  }
}
```

Listing 3.6: Generated Solidity Code

The framework would generate two separate files, one containing C# code intended to be deployed on a centralized desktop environment and another containing an Ethereum Smart Contract written in Solidity. The framework takes care of creating class and contract definitions for the respective files. Moreover, data type mappings are warranted accordingly. For instance, an integer denoted by the data type ‘int’ in a C# codebase is automatically mapped to a ‘uint’ in the case of Solidity syntax. This was done in effort to keep syntactical notation of the annotation framework as close to .NET C# syntax. Furthermore, while .NET languages allow functions to accept objects as parameters, Solidity only allows developers to pass in primitive types. If we take into consideration Listings 3.5 and 3.6, while the off-chain ’Register’ function accepts a parameter of type ’Profile’, the on-chain function was generated in such a way whereby it accepts all on-chain properties found within the ’Profile’ object in their primitive types. The only exception would be the ’hash’ property as this was annotated with the ’OutOnly’ annotation, meaning it would only appear in return statements and not parameters.

As can be seen in Line 13 of Listing 3.5, the Ethereum code block was annotated as being a synchronous process. In order to call an Ethereum Smart Contract from the desktop environment, XCall is used. XCall is a run-time library built to achieve seamless communication between the different locations supported by the framework and enables a standard and generic way of communication This library exposes a function that takes the following arguments:

- Location – Refers to the network the XCall library should initiate communication with.
- Contract Name - Refers to the name of the smart contract found on the specified platform.
- Function Name – Refers to the name of the function found within the Smart Contract.
- Parameter List – A comma separated list of parameters that the function accepts.

3.4 | System Architecture

Figure 3.3 illustrates the main components that comprise the framework being proposed. The source generator is a console application that takes as input the path to a file that contains the annotated code. The annotations need to indicate the intended deployment location of the code block beneath them. Consequently, separate files are generated containing the generated code for off-chain and on-chain deployments.

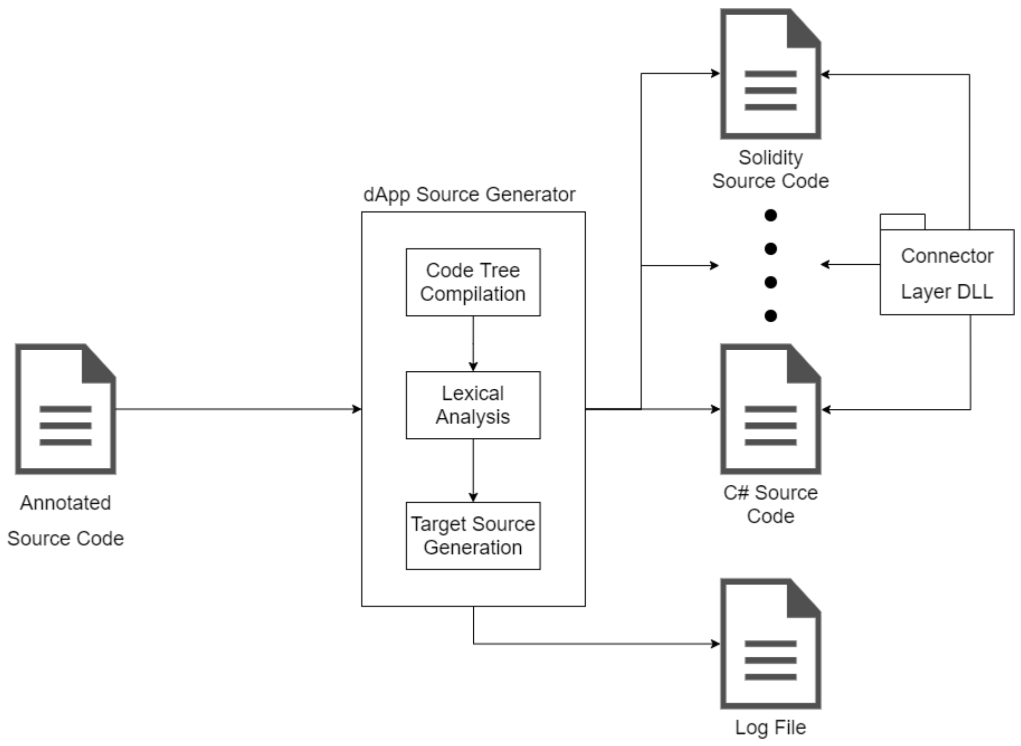


Figure 3.3: System Architecture

The process by which this is achieved is divided into three parts:

- Abstract Syntax Tree (AST) Compilation – During this stage, the input file is parsed down into an AST. This is achieved by taking the code found in the input file and creating a hierarchical tree consisting of nodes, where each node resembles a piece of code from the input file. This tree helps achieve a simpler procedure to iterate the input code and above all, provides all the required details to primarily analyse the tags associated with each piece code, and subsequently generate the segregated code files to be deployed to their respective target platform.

- Lexical Analysis – During this stage, the source generator parses the code tree to determine where each piece of code needs to be executed depending on its annotations. This step produces similar code trees to the previous step for each location that was identified from the annotations. However, these code trees only contain the necessary details needed to generate the source code in the following step.
- Code Generation – During this stage, the source generator takes the parsed data from the previous step, and generates specific code syntax depending on the location of where it needs to be executed.

Subsequently, the connector layer is a class library that allows for seamless communication between off-chain code and the different code executions happening on different on-chain networks. Methods exposed through this class library are referenced by the code generated by the source generator. Furthermore, this class library exposes several utility functions and extensions that allow for simpler development of decentralized applications.

For further information on how the processes mentioned in this section operate, please refer to Appendix A. The code for both the Source Generator, and also the connector library, is available on my GitHub page [1].

3.5 | Implementation Issues

Achieving Dynamic Source Generation The framework should provide ease of extensibility, allowing for other developers to support further locations to which the input source code can be decompiled down to. This ensures that developers are not limited to just desktop or Ethereum networks for example, but will be able to build decentralized applications to, let's say the NEO network. While creating a structure whereby annotations can be dynamically added was easily achieved, the source generation procedure carried its own set of challenges. These mainly revolved around the aspect of having different coding structures supported by the different platforms that the framework supports. For instance, while a C# class can consist of a number of properties and a number of functions, a Smart Contract declaration within Solidity can also have a list of Structs associated with it. Therefore, the procedure within the source generation section where all parsed data nodes are linked to their parents, had to be developed in

[1]https://github.com/ryanfalzon/DLT-Dissertation

such a way that can dynamically accept all types of syntax structures, irrespective of the platform that the data is being constructed to.

Decompiling Custom Model Types Creating custom models to represent the data stored within a database is essential in any platform that allows for application development. This is not an exception when developing decentralized applications. The limitation when developing the framework presented herein is the aspect of return types and parameters. This issue arises due to programming languages, like Solidity, that do not accept custom model types as return types and parameters to functions but rather opt to have the individual properties listed one after the other. Because of this, the lexical analysis procedure needs to keep track of the defined custom models and produces only the required properties from these models to be placed instead of return types and function parameters. An example of this is provided in Listings 3.8 and 3.9 which contain the annotated source code and the generated Solidity code respectively. Despite this, we believe that as more languages are supported, the framework's inherent support would increase as well.

```
[XOn(Ethereum)]
public class Profile
{
  public int Id;

  public string Hash;
}

[XOn(Ethereum)]
public class SocialNetwork
{
  public void Register(Profile profile)
  {
    @XOn(Ethereum, profile)
    {
      ...
    }
  }
}
```

Listing 3.7: Annotated Source Code

```
contract SocialNetwork{
  struct Profile{
    uint128 Id;
    string Hash;
  }

  function Register(uint128 id, string memory Hash) public{
    ...
  }
}
```

Listing 3.8: Generated Solidity Code

4

Evaluation and Validation

4.1 | Overview

To evaluate the approach and language proposed herein, we created an experiment which requires the development of several tasks using both the traditional approach and also using UniDAPP. We used the Social Media platform described in Chapter 3.2 as a real-world scenario of a decentralised application that requires the execution of user-defined stories and tasks defined in this chapter. These user-defined stories will allow developers to explore the outcomes of shifting logic and control flow to and from off-chain and on-chain locations by defining a set of development requirements to create a decentralized system that adheres to GDPR.

The traditional system and its subsequent tasks use a .NET code base for off-chain execution and Solidity to program the on-chain smart Contract code placed on the Ethereum network. On the other hand, the annotated code is written solely using UniDAPP, automatically generating the separate code bases for the individual platforms.

Certain overheads were measured from both approaches to assess whether the research questions laid out in Section 1.3 are answered. In the case of the annotation approach, we will be measuring the code written by the developer and not the generated code.

- Lines of Code – A percentage of added, deleted, and modified code from the total changes. This measurement will be taken for both the traditional and annotated approaches.
- Support and Functional Code – A ratio of support vs. functional code was read for both the traditional and annotated approaches. Functional code refers to logical

code derived from functional requirements such as data validation. Meanwhile, support code is written with the intent to allow the system to operate. A key example of such code is the ability to communicate to on-chain smart contracts from a centralized system.

- Expressiveness of Abstraction – The ability to have platform-specific code written in a generic form and then be decompiled down to their respective platform code-base. As opposed to the previous measurements, which are quantitative, this one is a qualitative measurement.

These measurements are based on related work laid out in Chapter 5. Due to time constraints, it was decided to leave out the usability aspect of the framework and identified this as a possible future work in Chapter 6.

4.2 | Experiment

We start first by defining each user story and their subsequent tasks, hence allowing the reader to understand what the expected outcome of each user story is. User stories are presented in the first person from the point of view of the user. The user is not limited to be the end-user of the system, but rather the person or entity that requires the user story to be completed. For instance, if a user story describes a new feature needed to be added to the system, in this case, the user is the end-user. However, if, on the other hand, the story describes a bug fix, the user, in this case, is the software developer.

The metrics calculated from the resulting code using the annotation framework will be compared to those when using the traditional approach. This allows us to analyse the usability from an unbiased point due to considering both traditional and annotation techniques.

4.2.1 | User Story 1: Centralized to Decentralized Framework

4.2.1.1 | Description

As a software developer, I want to shift the social application from a centralised nature to a decentralised one to satisfy the need for transparency that end-users desire. The tasks that would need to be carried out to complete the user story successfully are the following:

- Outline Smart Contract architecture, including crucial functionality and publicly available data.
- Migrate custom profile and post object models to on-chain structures. This process will require creating user-defined structs within the smart contract declaration that resemble the centralised application's custom model classes.
- Migrate data and control code to smart contract.

4.2.1.2 | Results

	Traditional		Annotated	
	Lines	Percentage	Lines	Percentage
Additions	105	50.24%	52	43.33%
Deletions	104	49.76%	56	46.67%
Modifications	0	0.00%	12	10.00%
Total	209		120	
	−42.58%			

Table 4.1: User Story 1

4.2.1.3 | Discussion

A percentage decrease of 42.58% in the number of lines that needed to be modified was achieved when opting to use the annotated framework to develop the first user story. Besides having lower addition and deletion figures, the number of modified lines increased. When using the traditional approach, the developer needed to delete the whole C# code base and re-code everything using Solidity, including contracts, functions and model declarations, aspects that did not need changing. When using the annotation framework, these declarations do not need changing as the source generator automatically generates the required code for the target location. This shows the framework's potential when developers are tasked with shifting data and control flow from a centralised environment to a decentralised one.

4.2.2 | User Story 2 - Hybrid Framework

4.2.2.1 | Description

As a software developer, I want to have specific data properties stored off-chain rather than on-chain. All data related to profiles and posts should be stored off-chain. On the other hand, profile and post hashes are created and stored on-chain to ensure that these have not been modified. In doing so, the application would be compliant with GDPR. The tasks that would need to be carried out to complete the user story successfully are the following:

- Modify user-defined on-chain structs to remove unnecessary object properties.
- Create custom model classes on the off-chain platform to store the data removed from the on-chain platform.
- Create the necessary code to store off-chain data while modifying the on-chain code to keep the desired model properties.
- Create a way of having a line of communication between off-chain and on-chain executions to call the desired smart contract code found on the Ethereum platform.

4.2.2.2 | Results

	Traditional		Annotated	
	Lines	Percentage	Lines	Percentage
Additions	172	73.19%	139	71.28%
Deletions	45	19.15%	32	16.41%
Modifications	18	7.66%	24	12.31%
Total	235		195	
		−17.02%		

Table 4.2: User Story 2

4.2.2.3 | Discussion

In the second user story, despite having a lower percentage decrease when using the annotated framework, that of 17.02%, one can still see that additions and deletions decreased. At the same time, the number of modifications carried out increased. This lower percentage occurs due to one of the limitations of the current version of UniDAPP. This limitation, which involves additional code lines to be written to retrieve data from on-chain code, will be further discussed in Section 4.3.2.2. However, if this limitation is addressed in future versions of the annotation framework, one would see an increase of 27.23% rather than the 17.02% measured in this experiment.

4.2.3 | User Story 3 - Centralized Privacy Settings

4.2.3.1 | Description

As a software developer, I want to create privacy settings functionality in a centralised environment. The privacy settings should allow users to toggle both profiles and posts to either a public or private option. The tasks that would need to be carried out to complete the user story successfully are the following:

- Create custom model classes on the off-chain platform to store the data of a privacy setting. The data should include a boolean flag for both profiles and posts, true referring to a public setting and false referring to a private setting, as well as a profile identifier.
- Create off-chain code that should verify and store the privacy settings accordingly.

4.2.3.2 | Results

	Traditional		Annotated	
	Lines	Percentage	Lines	Percentage
Additions	23	100.00%	26	100.00%
Deletions	0	0.00%	0	0.00%
Modifications	0	0.00%	0	0.00%
Total	23		26	
	13.04%			

Table 4.3: User Story 3

4.2.3.3 | Discussion

A 13.04% increase in lines of code was incurred when developing the third user story. The development of the tasks involved the creation of a centralised feature. Therefore, the traditional approach's code was identical to the one created in the annotation framework. The additional lines that were needed in the annotated codebase were the annotations that were added to indicate that the code should be stored or executed off-chain. This means that UniDAPP does not offer any specific benefits when compared to traditional approaches of developing either a centralised or decentralised feature.

4.2.4 | User Story 4 - Decentralized Privacy Settings

4.2.4.1 | Description

As a software developer, I want to store all the data generated by the privacy settings in a decentralised environment. While this will not hold any private user data in a public domain, thus being compliant to GDPR, the users will have the ease of mind that their privacy settings have not been tampered with.

- Migrate custom privacy settings object models to on-chain structures. This process will require creating a user-defined struct within the smart contract declaration that resembles the centralised application's custom model class.
- Migrate data and control code to smart contract functions.

4.2.4.2 | Results

	Traditional		Annotated	
	Lines	Percentage	Lines	Percentage
Additions	20	74.07%	14	60.87%
Deletions	7	25.93%	8	34.78%
Modifications	0	0.00%	1	4.35%
Total	27		23	
		−14.81%		

Table 4.4: User Story 4

4.2.4.3 | Discussion

In contrast to the above, the last user story required the previous story's functionality to be shifted from a centralised environment to a decentralised one. In this case, a decrease of 14.81% in modified lines of code was experienced. This small percentage can easily be improved and have a higher gap between the more traditional and annotation frameworks. This can be achieved by identifying how to create interchangeable logic code that would only require changing the annotation of its location. This will help reduce the number of added and deleted lines while adding a small percentage to the number of modifications due to the annotation change. An example of how this can be achieved can be seen in the listing below.

4.3 | Further Discussions

4.3.1 | Functional & Support Code

One can see how when using our annotation framework, the percentage of support code to functional code reduces throughout all the use cases. Use case 1 sees a 100% decrease in support code, while in use case 2, an 84.26% decrease was achieved. Despite both traditional and annotated approaches having 0% support code in use case 3, support code in use case 4 drops by 41.3% when using UniDAPP. This shows how developers can focus more on the underlying logic, rather than writing support code, when using the annotation framework.

	Support		Functional	
Use Case 1	4	1.91%	205	98.09%
Use Case 2	46	19.57%	189	80.43%
Use Case 3	0	0.00%	23	100.00%
Use Case 4	2	7.41%	25	92.59%

Table 4.5: Traditional Approach

	Support		Functional	
Use Case 1	0	0.00%	120	100.00%
Use Case 2	6	3.08%	189	96.92%
Use Case 3	0	0.00%	26	100.00%
Use Case 4	1	4.35%	22	95.65%

Table 4.6: Annotated Approach

4.3.2 | Expressiveness of Abstraction

4.3.2.1 | Cross-Platform Code Compatibility

While most of the Solidity programming language highly resembles syntax found within the C# language, there are forms of Solidity syntax written in a different grammar. This isn't just specific to C# and Solidity as some language features are unique to platforms and some that have commonalities with others. Hence we believe that it is important that common ones are included and represent the different platforms to a sufficient level and yet uncommon ones can be supported by allowing for platform specific code.

Let us consider the scenario where a developer needs to create a mapping and store it on-chain. Traditionally, one would need to create solidity syntax similar to the below:

```
mapping(bytes32 => Post) public Posts;
```

However, since the above is not part of the .NET syntax, when using UniDAPP, developers must escape it using the '\>' escape character. However, in future versions of the source generator, such instances should be handled by finding the closest counterpart in the .NET domain and have the source generator decompile that line to its respective form based on the provided annotation. Hence, the mapping syntax above could be written as a native C# dictionary in the annotated code, only to be decompiled down to a Solidity mapping by the source generator as below:

```
[XOn(Ethereum)]
Dictionary<int, Post> posts = new Dictionary<int, Post>();
```

4.3.2.2 | Chaining Annotations

One limitation of the current version of UniDAPP is the flexibility in combining annotations with individual pieces of .NET code. This poses certain overheads that with traditional techniques are not experienced by developers. Suppose one considers a scenario where a function is created to retrieve a specific profile from an application that resides on a decentralized platform. In this case, specific properties might need to be retrieved from an off-chain location, while others from an on-chain location. Traditional techniques would require developers to create an off-chain function that closely resembles the code in Listing 4.1 below.

```
1  public Profile GetProfile(int profileId)
2  {
3    var profile = GetProfileQuery<Profile>.Execute(new {Id = id});
4
5    Connector connector = new Connector(credentialManager.PublicKey,
         credentialManager.PrivateKey);
6    profile.Hash = connector.Call("getProfileHash", id, hash);
7
8    return profile;
9  }
```

Listing 4.1: Get Profile - Traditional Approach

While most profile details are retrieved from an off-chain SQL database, line 6 of Listing 4.1 shows how the profile hash is retrieved from the Ethereum platform and assigned to a property found within the Profile object. Arguing how this can be overcome by retrieving the profile hash from the SQL database would contradict using a distributed application platform since one would be solely relying on data residing on a centralised platform that could be susceptible to malicious attacks. By using the annotation library, the development of such a function would be similar to the code found in Listing 4.2.

```
public Profile GetProfile(int profileId)
{
  ~@XOn(Desktop, profileId)
  {
    var profile = GetProfileQuery<Profile>.Execute(new {Id = id});
    profile.Hash = GetProfileHash(profileId);
    return profile;
  }
}

public string GetProfileHash(int profileId)
{
  return @XOn(Ethereum, profileId)
  {
    Profile profile = profiles[profileId];
    return(profile.Hash);
  }
}
```

Listing 4.2: Get Profile - UniDAPP (Current Version)

As can be seen above, the annotated block found between lines 13 and 17 had to be placed within its separate function, which is called to initialize the profile hash variable in line 6. This needs to be done since UniDAPP fails to support the functionality of having the return value from annotated code blocks initialize a variable. This might cause an unwanted overhead for the developer as they would still be required to create two separate functions, just as one would need to do in traditional approaches.

One option to mitigating such a limitation would be to develop the required connectors that would allow developers to populate variables with the return value from an annotated code block. However, this would require one to have nested annotated scopes that, in such a scenario, would have different locations of execution. An example of this is provided in Listing 4.3 below.

```
public Profile GetProfile(int profileId)
{
  ~@XOn(Desktop, profileId)
  {
    var profile = GetProfileQuery<Profile>.Execute(new {Id = id});
    profile.Hash = @XOn(Ethereum, profileId)
    {
      Profile profile = profiles[profileId];
      return(profile.Hash);
    }

    return profile;
  }
}
```

Listing 4.3: Get Profile - UniDAPP (Incorrect Approach)

However, such an approach would confuse developers who might be undertaking code reviews and analysis, as it would provide uncertainty of where the code should be executed. At first glance, one would expect that anything placed within the annotated desktop code should be executed on a desktop platform, despite having the annotated Ethereum code included within this scope. Alternatively, a more intuitive solution would be to allow unannotated variable declaration outside of annotation scopes to use such variables from within annotated scopes. Listing 4.4 provides an example of this solution.

```
1   public Profile GetProfile(int profileId)
2   {
3     ~@XOn(Desktop, profileId)
4     {
5       var profile = GetProfileQuery<Profile>.Execute(new {Id = id});
6     }
7
8     var profileHash = @XOn(Ethereum, profileId)
9     {
10      Profile profile = profiles[profileId];
11      return(profile.Hash);
12    }
13
14    @XOn(Desktop)
15    {
16      profile.Hash = profileHash;
17      return profile;
18    }
19  }
```

Listing 4.4: Get Profile - UniDAPP (Future Update)

As shown in line 8, a variable for holding the profile hash is created and initialised using the annotated Ethereum scope's return value. In doing so, the developer creates a shared variable that can be used across annotated scope blocks found within the enclosing function. Such an approach can be considered more intuitive for developers as such practices are already adhered to in traditional programming languages.

4.3.3 | Framework Extensibility

Despite not being a primary aim of this dissertation, throughout the development of the framework, it became evident that having such a framework supporting additional code locations is essential. This would mean that systems can be developed using a greater variety of target platforms, other than .NET and Ethereum Smart Contracts. Having the framework support a new location would require the addition of the respective location generator and extending the connector library to allow it to support communication to and from the new location. The location generator refers to the transformation process required to be carried out on the annotated parsed code, in order to have it readily available for deployment on the target location. On the other hand, implementing the connector library is dependent on what frameworks are currently available to interact with the desired location. For instance, connections to .NET platforms are generally made via API calls, while interactions, such as Smart Contract calls, to the Ethereum network, can happen via JSON RPC calls to supported clients such as Geth or Parity. Even though this is neither a framework limitation nor a drawback for its users, it is still something one needs to consider when creating decentralized applications. As opposed to traditional development of such systems, where this work would need to be incurred by the system developers, when using the annotation framework, this work would not be required as it would already be made available by the annotation framework developers.

Consider a use case where educational institutions would opt to shift academic certificates from a centralised environment to a decentralised one. Like the social application use case, processing of private user data, such as student names and marks, is essential. Hence, in such a case it would not be ideal to shift to a completely decentralized system to avoid having sensitive data stored on-chain. An alternative solution would be to use a hybrid system where both public and private chains are used to leverage the technology's benefits. Figure 4.1 below illustrates how such a system would operate.

Let's assume that a portal written using .NET code is made available to educational institutes to be able to upload certificates and also to students to view their certificates. Educational institutes upload the raw certificate to a Hyperledger private chain which is only accessible by approved entities. This makes the private data consolidated to only trusted individuals. However, since private networks can be perceived as still being controlled by a particular entity, a hash of the data uploaded on the private network is created and subsequently anchored to a public network, thus allowing users to ensure that their data has not been tampered with.

However, this comes at a cost as developing such a system would require coding on three separate platforms, .NET, Hyperledger and Ethereum. A framework of the likes being proposed would benefit developers since they would be allowed to design such a system while using one codebase. Furthermore, due to the benefits explained thus far, changing the location where specific data properties reside, would be far more straightforward in one environment than manipulating three individual environments. It is also important to note that due to blockchain still being in its infancy, many application might not last the face of time, therefore, it might not be worthwhile to invest time in building such an application.

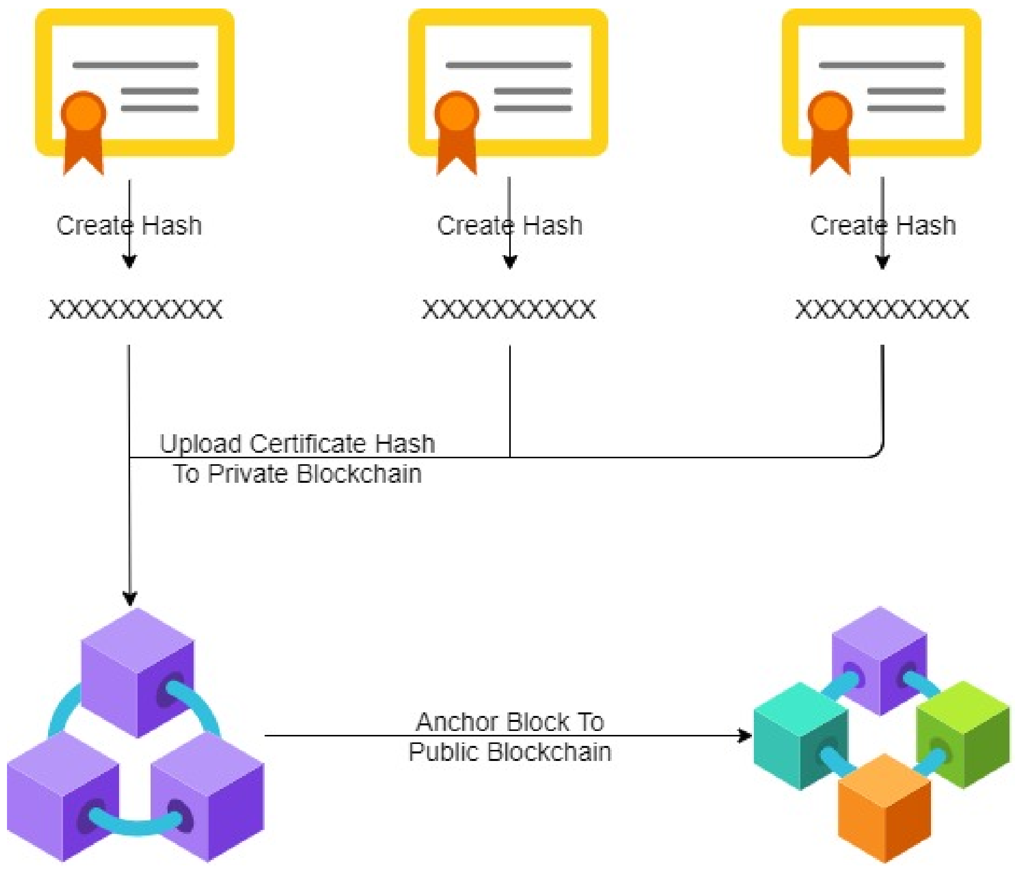

Create Hash
Create Hash
Create Hash
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
Upload Certificate Hash To Private Blockchain
Anchor Block To Public Blockchain

Figure 4.1: Hybrid System Example

4.4 | Summary

In this chapter, we evaluated the annotation-based framework proposed in this dissertation. The research questions set out in the first chapter were validated against the results achieved from the experiment. The experiment consisted of developing a number of user stories for the predefined use-case in both the traditional manner and the proposed annotation framework. For both instances, changes applied to the code were noted at each step. This was done to find the percentage difference of changes between the two.

- **Research Question 1**: Results showed how our annotation framework allows developers to explore the change in execution location for both logic and data-flow in a more effortless manner. However, when adding a wholly centralized or decentralized feature, our framework showed no benefits than traditional approaches.

The annotation framework also reduced the amount of support code to functional code that needed to be written.

- **Research Questions 2 & 3**: Cross-platform code compatibility and the ability to have chained annotations were identified as possible drawbacks of the current version of the framework, and would most likely induce an additional overhead on the developer. Section 4.3 provides a detailed explanation to possible solutions of these drawbacks.

5

Related Work

Although related work on a unified model for decentralized application is limited, several advancements have been carried out in the field of macroprogramming. These mainly relate to wireless sensor networks. Despite this, efforts have been made in the blockchain domain to develop suitable models for developing cross-chain smart contracts, and IoT enabled blockchain networks. Therefore, throughout this section, an overview of work on macroprogramming within the field of wireless sensor networks will be presented.

5.1 | Macroprogramming for Wireless Sensor Networks

Regiment Regiment, a functional language whose syntax closely resembles Haskell's, aims to provide an easier way of programming wireless sensor networks. The network is depicted as a set of spatially distributed time-varying signals, where each set represents a region of nodes within the network. Compiling a Regiment program will create an intermediate language-like model between Regiment and the languages supported by the individual sensor nodes. This intermediate language, coined as the Token Machine, handles communications and delegations between the individual sensor nodes by capturing only those operations that each sensor supports. This approach differs from our annotation framework. While we proposed a unified model that translates down to the separate code blocks written in their target language, Regiment is a language that compiles down to an intermediate language (Newton & Welsh, 2004).

```
dosum :: float, (float, int) -> (float, int)
fun dosum(temp, (sumtemp, count)) {
  (sumtemp + temp, count + 1)
}
tempreg = rmap(fun(nd) {sense("temp", nd)}, world);
sumsig = rfold(dosum, (0, 0), tempreg);
avgsig = smap(fun((sum, cnt)) {sum / cnt}, sumsig);
BASE <- avgsig
```

Listing 5.1: Wireless Sensor Network Programmed Using Regiment (Newton & Welsh, 2004)

Listing 5.1 provides an example of a program written in Regiment to find the average temperature within the sensor network. The program makes use of *rmap*, *rfold* and *smap*, which are constructs defined in the Regiment language. While *rmap* helps obtain a reading from all sensors within the specified region, *rfold* is used to aggregate these temperatures into a single variable. Finally, *smap* is applied on the resulting signal to retrieve the average temperature.

Kairos Kairos, an extension to the Python language, allows programmers to state the global behaviour of wireless sensor networks explicitly. This is achieved by tackling the abstraction process in three stages; nodes, one-hop neighbours and remote data access abstraction. The Kairos programming model also relies on another fundamental aspect which is eventual consistency. Gummadi et al. (2005) argue that the state of each individual intermediate node cannot be guaranteed. However, computational convergence is bound to happen in the eventuality of a failure. A shared memory model is utilised where a shared node state is maintained through a message passing technique. This approach provides a centralised view of the whole network, thus making programming such networks easier. Despite this, it still does not offer programmers the ease of mind of not needing to understand such a system's underlying mechanisms by leaving such handles to the framework and automatically optimising communication patterns for a defined topology (Gummadi et al., 2005).

```
void buildtree(node root)
  node parent, self;
  unsigned short dist_from_root;
  node_list neighboring_nodes, full_node_set;
  unsigned int sleep_interval = 1000;

  // Initialization
  full_node_set = get_available_nodes();
```

```
    for (node temp = get_first(full_node_set); temp != NULL; temp =
        get_next(full_node_set))
      self = get_local_node_id();
      if (temp == root)
        dist_from_root = 0; parent = self;
      else dist_from_root = INF;
      neighboring_nodes = create_node_list(get_neighbors(temp));
    full_node_set = get_available_nodes();
    for (node iter1 = get_first(full_node_set); iter1 != NULL; iter1 =
        get_next(full_node_set))
      for(;;) //Event Loop
        sleep(sleep_interval);
        for (node iter2 = get_first(neighboring_nodes); iter2 != NULL;
            iter2 = get_next(neighboring_nodes))
          if (dist_from_root@iter2+1 < dist_from_root)
            dist_from_root = dist_from_root@iter2+1;
            parent = iter2;
```

Listing 5.2: Wireless Sensor Network Programmed Using Kairos (Gummadi et al., 2005)

A complete program in Kairos for constructing a routing tree from a given node is given in Listing 5.2. The use of macroprogramming constructs such as *get_available_nodes*(), which gets a list of available nodes found within the network, and *get_neighbors*(), which gets a list of one-hop neighbours from the given node, are used. These constructs make it easier to construct the routing tree as opposed to writing the individual sensor code required to gather the required data.

COSMOS Another approach to macroprogramming wireless sensor networks was that of COSMOS. The COSMOS platform consists of the mOS operating system and the mPL programming language. mPL allows distributed data processing to be specified in terms of data flow in the form of functional components written using a subset of the C language. Once developed, these functional components can be reused across other systems. mOS, on the other hand, serves as a heterogeneous runtime environment for programs written in mPL. One of the benefits that COSMOS provides is the ability to introduce additional abstraction features without the need to change the underlying structure of mOS. Our annotation framework was built using the same thought process with the hope that in the future, the proposed framework will be able to support additional locations that developers can target their code to (Awan et al., 2007).

```
// Logical Instances
accel_x   : accel(12);
disp      : disp1, disp2;
cpress_fc : cpress;
thresh_fc : thresh(250);
max_fc    : max;
fft_fc    : fft;
ctrl_fc   : ctrl;

// Refining Capability Constraints
@ on_mote = MCAP_ACCEL_SENSOR : thresh, cpress;
@ on_srv = MCAP_UNIQUE_SERVER : ctrl;

IA {
  timer(30) -> accel;
  accel     -> cpress[0];
  cpress[0] -> thresh[0], max[0];
  thresh[0] -> fft[0];
  fft[0]    -> disp1;
  max[0]    -> ctrl[0], disp2 | max[1];
  ctrl[0]   -> thresh[1];
}
```

Listing 5.3: Wireless Sensor Network Programmed Using COSMOS (Awan et al., 2007)

From Listing 5.3, one can see how programmers make use of components such as *accel*, *cpress* and *thresh*, written using native C syntax, to define logical instances. Despite this, component interaction and dataflow, found within the IA scope, are defined using the abstraction model provided by mPL language.

PyoT Similar to UniDAPP, PyoT is a macroprogramming framework for developing a distributed application in the context of an IoT environment. By abstracting actuators and sensors as CoAP resources, more commonly referred to in their work as software objects, they could hide the complexities that such networks are surrounded by. These software objects can then be programmed using either a web-based UI or through shell commands. Furthermore, the programmer defines where pieces of code should be executed, leaving the interpreter to distribute the code accordingly, similar to the proposed annotation framework. They conducted experiments aimed at measuring the execution time, scalability, newly created overheads, and real-world implementations. This resulted in the conclusion that although there exists an execution time overhead, the framework is still a flexible and scalable one (Azzara et al., 2014).

```
temps = Resource.objects.filter(title='temp')
results = [temp.GET() for temp in temps]
avg = sum (results) / len(results)
if avg > 24
  Resource.objects.get(title='fan').PUT('on')
```

Listing 5.4: Wireless Sensor Network Programmed Using PyoT (Azzara et al., 2014)

Listing 5.4 provides an example of how PyoT could be used to activate a fan given that the average temperature read from the sensors exceeds a specific value. The *Resource* construct defines a list consisting of the different sensors that is available within the wireless sensor network. Each element within this list contains defined functions, such as $GET()$, that retrieves the information available from the sensor, and $PUT()$ which sends signals to the sensor with the aim to manipulate the sensors' state.

5.2 | Macroprogramming for Blockchain Systems

Porthos Porthos is a DSL, embedded in Haskell, that allows for programming of smart contracts across several blockchain networks. A smart contract written in Porthos will enable developers to define the location where each portion of the smart contract should execute. A compiler creates separate files containing the code needed to be deployed to the respective chain, for example, a Solidity file for Ethereum and a GO Chaincode file for Hyperledger. Coupled with the Porthos DSL, a messaging routing mechanism acts as a communication medium that relays messages from one network to the other. This is achieved by listening for on-chain events and triggering a specific action once an event happens. Through the experiments carried out, one could see how by raising the level of abstraction, such a system could be developed without worrying about the complex code for communication, thus needing to focus only on the contract logic. While the concept of having annotated code is the same as what is being proposed in this research, the two are distinct as Porthos was aimed at providing a solution for chain interoperable smart contracts. Simultaneously, the annotation framework works at solving the challenges faced with distributed application development (Mizzi et al., 2019).

```
1  savings :: Participant -> Time -> Contract
2  savings recipient expiryTime =
3    repeatCommit "save" (ETH, isCommitTo recipient)
4      (onTimeout expiryTime (releaseAll end))
```

Listing 5.5: Time-Locked Savings Plan Using Porthos (Mizzi et al., 2019)

Listing 5.5 above provides an example contract written in Porthos. Placement of code depends on the asset being used. Since in line 3 the asset in question is ETH, Porthos automatically translates the required code to be executable on the Ethereum network.

D'Artagnan Similar to the framework we proposed, D'Artagnan is a high-level macroprogramming language, embedded in Haskell, that allows developers to describe blockchain-connected IoT devices that provide some form of data to a smart contract dependent system. A single program written using D'Artagnan is passed through the framework, which in turn processes and outputs the code for the smart contracts, edge nodes, and IoT devices. The blockchain edge node acts as a listener for events occurring on-chain to fire up the necessary action to retrieve data from IoT devices and return to smart contracts. To increase the level of abstraction that the framework offers, Haskell primitives can be used in the macroprogram. Like our approach, D'Artagnan provides programmers with the ability to place code within one of three possible arrangements; IoT-focus, Edge-focus or Blockchain-focus. This allows developers to experiment with different locations where data flow and control logic can be placed to reduce the amount of Gas needed to execute a smart contract for smart contract code and achieve the best performance from IoT devices. Moreover, the communication required to pass messages from the blockchain network where the smart contract resides to the IoT devices and vice-versa is all taken care of by the D'Artagnan framework. This is another similarity with what is proposed in this research, where the communication between the different locations that developers can place their code is handled by the underlying mechanism of the proposed framework (Mizzi et al., 2018).

```
payPerCycle :: (Int, Stream Bool) -> Stream Int
payPerCycle (fee, inUse) = ifThenElse (inUse, (liftS fee, liftS 0))

payByConsumption :: (Int, Stream Int) -> Stream Int
payByConsumption (fee, usage) = liftS fee .*. usage

meter :: Stream Int -> Stream Int
meter feed = let x = pre 0 x .+. feed
  in x
```

Listing 5.6: Smart Rent Application Using D'Artagnan (Mizzi et al., 2018)

In Listing 5.6, a smart rent application written using D'Artagnan is given. Such an application calculates electrical consumption based on the meterage acquired from household appliances. The beauty of such a solution is that the same application can be deployed to different environments containing varying types of edge devices. The D'Artagnan framework automatically generates the code that would be required by each nodes, thus making it extremely easier for developers to manage as opposed to writing different low-level code for individual nodes to accommodate the different platforms they might support.

AlkylVM Ellul & Pace (2018) present AlkylVM as a means of integrating resource constrained devices to blockchain systems. Their proposed solution consisted of the *Aryl Blockchain Node* and the *Alkyl Virtual Machine*. In essence, the *Aryl Blockchain Node* connects to an Ethereum network similar to common Ethereum nodes. The Aryl node is tasked with continuously searching for contract events that would require input from IoT devices, extract and pass application logic from identified events to IoT devices, and also with writing blockchain transactions containing the final result. The Alkyl Virtual Machine, on the other hand, is a virtual machine running on all IoT-enabled devices that the Aryl Blockchain node has visibility of. The importance of the Alkyl Virtual Machine is that it is able to execute the instruction extracted from the contract events. Programs needing to execute on the Alkyl Virtual Machine are written using Alkyl, which is a strongly typed C-like language. This model not only allows resource constrained IoT devices to interact with Blockchain networks, but also keeps track of the Intermediate Representation being executed off-chain, on a trustless environment.

```
On PaymentEvent(string encryptPin, uint32_t mins) {
  string pin = Decrypt(encryptPin);
  for each (device in devices) {
    device.SetPin(pin, block.timestamp + (mins * 60))
  }
}
```

Listing 5.7: Smart Rent Application Using AlkylVM (Ellul & Pace, 2018)

```
char* pin;
time_t expires;
uint8_t index;
bool valid;

public void SetPin(char* pin, time_t expires) {
  this.pin = pin;
  this.expires = expires;
  this.valid = true;
}

deviceevent void KeyPress(char key) {
  if (pin[index] != key) {
    valid = false;
  }

  index++;
  if (index == 4) {
    index = 0;
    if (valid) {
      SystemCall(Unlock);
    } else {
      SystemCall(IncorrectBeep);
      valid = true;
    }
  }
}
```

Listing 5.8: Smart Rent Application Using AlkylVM

While Listing 5.7 is an example of a script executing on the Aryl Blockchain Node, Listing 5.8 illustrates an example of a piece of code meaning to be executed on an AlkylVM-enabled IoT node. The code allows users to guess the pin, set by the Smart Contract event. All interactions, i.e. entering a valid or invalid pin, with the IoT device are recorded on the Ethereum network by using the *'SystemCall'* function.

iContractML Hamdaqa et al. (2020) presented a DSL for modelling and deploying smart contracts to multiple blockchains. By using a model editor, developers are able to create an abstract model of a smart contract, which can later be deployed to different blockchain platforms without having to modify the underlying code of the smart contract. This is made possible by using a source generator which takes as input the smart contract model, and creates the smart contract code depending on where the user wants to deploy the smart contract. Hamdaqa et al. (2020) evaluated this approach by creating three distinct smart contracts using iContractML and deploying them to the respective blockchain platform. By calculating metrics introduced by Guizzardi et al. (2005), i.e. the lucidity, soundness, laconicity and completeness they were able to deduce whether the proposed framework holds up to their research questions. Their results showed that despite iContractML being able to generate the configuration files required, these are not enough to solely rely on iContractML to deploy blockchain smart contracts, thus still requiring the intervention of developers prior to deployment.

iContractML is relatively similar to what we are proposing. The technique of model-once-deploy-anywhere favours for code reusability, however does not address issues that developers might face when utilizing the smart contract created by iContractML from a centralized desktop environment for instance. These issues, however, can be addressed or mitigated through the use of our annotation framework.

5.3 | Summary

Framework	Written In	Written For	Evaluation Description
Regiment	Haskell	Wireless Sensor Networks	Feasability of the basic Regiment primites through a highly restricted subset of the language was explored.
Kairos	Python	Wireless Sensor Networks	Measurement of performance and convergence time when using a system built using Kairos and one built in Python.
COSMOS	C	Wireless Sensor Networks	An evaluation on the performance of the system after a pre-defined test case had been built using COSMOS.
PyoT	Python	Wireless Sensor Networks	Tests were performed in an emulated environment and a real testbed with the aim of measuring execution time, architecture scalability, task distirbution overhead and performance.
Porthos	Haskell	Chain Interoperability	Expressiveness of abstraction, security analysis and extensibility were noted when building a number of use cases using Porthos.
D'Artagnan	Haskell	IoT Enabled Blockchain	Code generated automatically by D'Artagnan was compared to one coded manually.
AlkylVM	C	IoT Enabled Blockchain	An example was provided as to how one would go about creating an IoT-enabled smart contract application.
iContractML	OBEO Designer	Smart Contract Deployed On Multiple Networks	The generated smart contracts of the pre-defined use cases was analyzed to asses whether the artifacts created are enough to be deployed to a target blockchain.

6

Conclusions

6.1 | Achieved Aims and Objectives

This dissertation's main aim was to assess whether a unified model can be applied to distributed application development to address several challenges experienced by developers during such a process and subsequently identify any new difficulties and ways of how these can be overcome. Our approach involved designing and creating an annotation library that enables users to write unified smart contract and system code by making use of annotations to distinguish between where code should be placed and executed.

A framework was designed whereby once the developer creates the annotated code, this is passed to our interpreter for processing. The interpreter parses the file and identifies each annotation that was listed by the developer. This process automatically creates separate files, each containing code intended to be executed on the tags' respective location. For instance, generating C♯ code to replace the annotated code to be run on a desktop environment in one file and Solidity code to replace the Ethereum annotated code in another file. A layer acting as an intermediary between the different target locations the framework supports was also developed to facilitate easy and efficient exchange of messages.

This framework was evaluated by comparing the development of several user stories that needed to be developed using traditional distributed coding techniques and the UniDAPP framework. Our experiments made it evident that the proposed framework reduces development overheads that developers need to incur to experiment with the placement of data and control flow code. This can be concluded from the reduced percentage of modifications required to be carried out to implement the user story when using UniDaAPP. Furthermore, from the experiment, one can see how the ratio of support code to functional code was reduced. This would infer that developers could focus their attention on more critical logical aspects of the code rather than support functionality.*

6.2 | Critique and Limitations

One of the limitations of the current version of the framework is that annotation chaining is not supported. This means that if off-chain code is dependent on the result obtained from an on-chain execution, a developer would need to split these functionalities up into different methods. Ideally, the ability to link multiple block annotations together would exist as this would reduce overheads for users.

Furthermore, something which emerged from the evaluation is that specific domain-specific code has not yet been fully standardized. This would mean that even though developers can develop decentralized applications under one framework, sometimes they would still be required to write code specific to the location being deployed to. A key example of this would be a mapping statement used in a Solidity smart contract. In .NET, there exists no direct counterpart to a solidity mapping; hence, when using UniDAPP, developers would need to write Solidity syntax within the UniDAPP file.

6.3 | Future Work

Being the first attempt at creating a solution to the problems laid out in Section 1.4, several improvements could be made to the proposed language framework to further improve its usability. These improvements are over and above the enhancements discussed in the evaluation section which could improve the results obtained thus far. The items presented hereunder are not limited to technical improvements, but also relate to future research that can be materialized.

First and foremost, one can explore whether a real-time syntax analyzer, similar to an 'intellisense', could be of benefit to developers using the annotation framework. As it stands, the developer is only notified of any syntax errors that exist in relation to the written annotations. This means that the source generator still proceeds with creating the target code despite having syntax errors in native C# and Solidity code. Thus, by creating a syntax analyzer that runs in the background when the annotated file is being created, helps notify developers that syntax errors exist within their code. This feature can be further enhanced by linking the syntax analyzer with the source generator to have it reject any annotated code containing some sort of syntax error.

An important aspect of decentralized application development is the process of testing the system. The on-chain portions of such systems inherit all benefits that on-chain transactions have, one of them being immutability, meaning that once an on-chain smart contract is published to the chain, no possible way of editing that smart contract would exist. This makes testing a crucial phase of developing such systems as it is within the interest of the developer that a smart contract is deployed without any form of bugs. Unless developers take a test-driven development approach with coding such a system, writing tests for a fully-developed system is a repetitive task and very difficult. A process whereby a form of verification technique can be carried out on the generated code can be automatically created and invoked. This process can involve approaches ranging from simple unit tests to runtime verification like LARVA (Colombo et al., 2009) or even static verification like KeY (Ahrendt et al., 2015).

Finally, further research can be carried out on the usability of the framework. The scope of this research was limited to flexibility in terms of logic and data storage. However, the framework from the point-of-view of the developer was not explored. Therefore the usability can be further explored by venturing into quantitative research and creating an experiment where a group of developers will develop the system using both traditional tools and using the annotation framework. Subsequently, developers would be interviewed to deduce and analyze their thoughts on the annotation framework. Doing so will provide a user-acceptance like understanding how elements of the framework can be improved or additional features that could be added that would not have been brought up without such an experiment.

6.4 | Final Remarks

The main aim of this thesis was to study the use of macroprogramming techniques coupled with annotations to create a framework for developing decentralized applications in an easier manner than what is currently the norm. It has been shown, that using the framework being proposed, developers would incur less development overheads when creating such applications. Nonetheless, by enhancing the framework with the features discussed in the Section 6.3, and also overcoming the issued identified in Section 6.2, we believe that the framework can be leveraged for its benefits to be used as the primary development tool for decentralized applications.

A

Implementation Details

A.1 | Abstract Syntax Tree

In order to be able to analyse the annotated code efficiently, the code located in the input file is parsed down to a hierarchical tree structure. This is done to create a visual representation of the code that needs to be traversed. Taking as an example the code found in Listing A.1, this would produce the AST that is illustrated in Figure A.1. One can notice how details such as access modifiers, parameters, annotations and property names are depicted as separate nodes within the source tree.

```
[XOn(All)]
public class Profile
{
  public int Id;
}
```

Listing A.1: Source Tree Code

A.2 | Lexical Analysis

Throughout this section, a detailed explanation of the lexical analysis process of the source generator will be provided. Here, the annotated code found in the input file is parsed down to identify where each code segment should be executed. This can be seen in the flowchart provided in Figure A.2.

The first step in the parsing process, is to create a source tree from the annotated code. This is done with the help of the Roslyn library which can create a source tree with multiple levels of hierarchy, thus allowing for easy analysis of each code block. Once the source tree is created, a lexical analysis procedure is carried out whereby each level of the tree is iterated to process each node and determine where it should be placed.

One of the objectives that the source generator needed to meet was the ability to support further on-chain networks in the future, easily and efficiently. This was achieved by creating a dynamic factory that returns the appropriate XChainGenerator based on the attribute defined by the user in the annotation being parsed. An XChainGenerator is a utility created for this research that stores the parsed data from the lexical analysis for future use. The class diagram in Figure A.3 shows the inheritance hierarchy of the XChainGenerators. An interface 'IXChainGenerator' contains a set of defined methods that each XChainGenerator must contain. This interface is inherited by an abstract class 'XChainGenerator' that defines and implements functionality that, irrespective of the location were the code needs to execute, needs to be carried out on the parsed code. Finally, separate generators implement the aforementioned 'XChainGenerator' which in turn implements the functions defined in the interface in their unique manner. All this is accessed through the 'XChainGeneratorFactory' that returns the appropriate XChain-Generator.

Let us consider a developer wishing to extend the framework by implementing support for HyperLedger Chaincode. In such a case, the developer would need to create a separate class '*XOnChainHyperledgerGenerator*' class which inherits and implements all methods in the '*XChainGenerator*'. This would allow the lexical analyzer to recognize annotated code with the '*Locations.Hyperledger*' tag. Together with what will be explained in Section A.3, developers would be able to fully integrate another target location within the framework.

A.3 | Source Generation

After a successful lexical analysis process, the final stage of acquiring the separated code base is to generate the source code from the parsed data. The lexical analysis provides a list of unlinked models that contain all the relevant information to generate the source for the individual frameworks the proposed solution supports. Figure A.4 offers a visual representation of the process by which the source is generated.

The first step of this process is to identify all top-level structures within the list of models. In essence the program would be filtering out all classes and contract decelerations. This assumption is made based on the decision that these decelerations are not children of any other form of structure. A consume function is invoked on each of these top level items to identify their children. This process is repeated on each identified child iteratively until the lowest level element is reached and there are no more children are left to be identified. Figures A.5 and A.6 illustrate a hierarchy diagram of how the identified types are structured for off-chain and on-chain Ethereum frameworks respectively. Continuing with the example provided at the end of the previous section, a developer extending the framework to support another target location would need to create similar models to the ones shown in Figure A.6.

A.4 | Locations Communication Channel

The Connectors runtime library allows for seamless communication between off-chain and on-chain code executions. Similar to the source generator, the library needed to be built in a generic way that would make it easy for the addition of further networks in the future. A factory mechanism was also adopted here whereby the appropriate connector is provided based on the parameter that is passed. Such a mechanism allows developers to extend the framework without needing to modify the underlying logic of the library itself, thus only requiring the developer to create new code to support the new target location.

Apart from this, a number of extension methods are also included in this library that allow simpler code to be written by the developer for both off-chain and on-chain code. These include the following:

- Hash – This function accepts any arbitrary object and, by using the SHA256 algorithm, produces a a unique hash comprising of a 32-bit string.
- ToHexString – This function accepts a byte array and produces a hexadecimal string representing that represents the byte array.
- Assert – This function asserts the passed condition and throws an exception if the condition is not met.
- IsNotNull – With the help of reflection, this function dynamically checks that none of the properties of the passed object are null or empty. If at least one of the properties is null or empty, false is returned. Otherwise, true is returned.

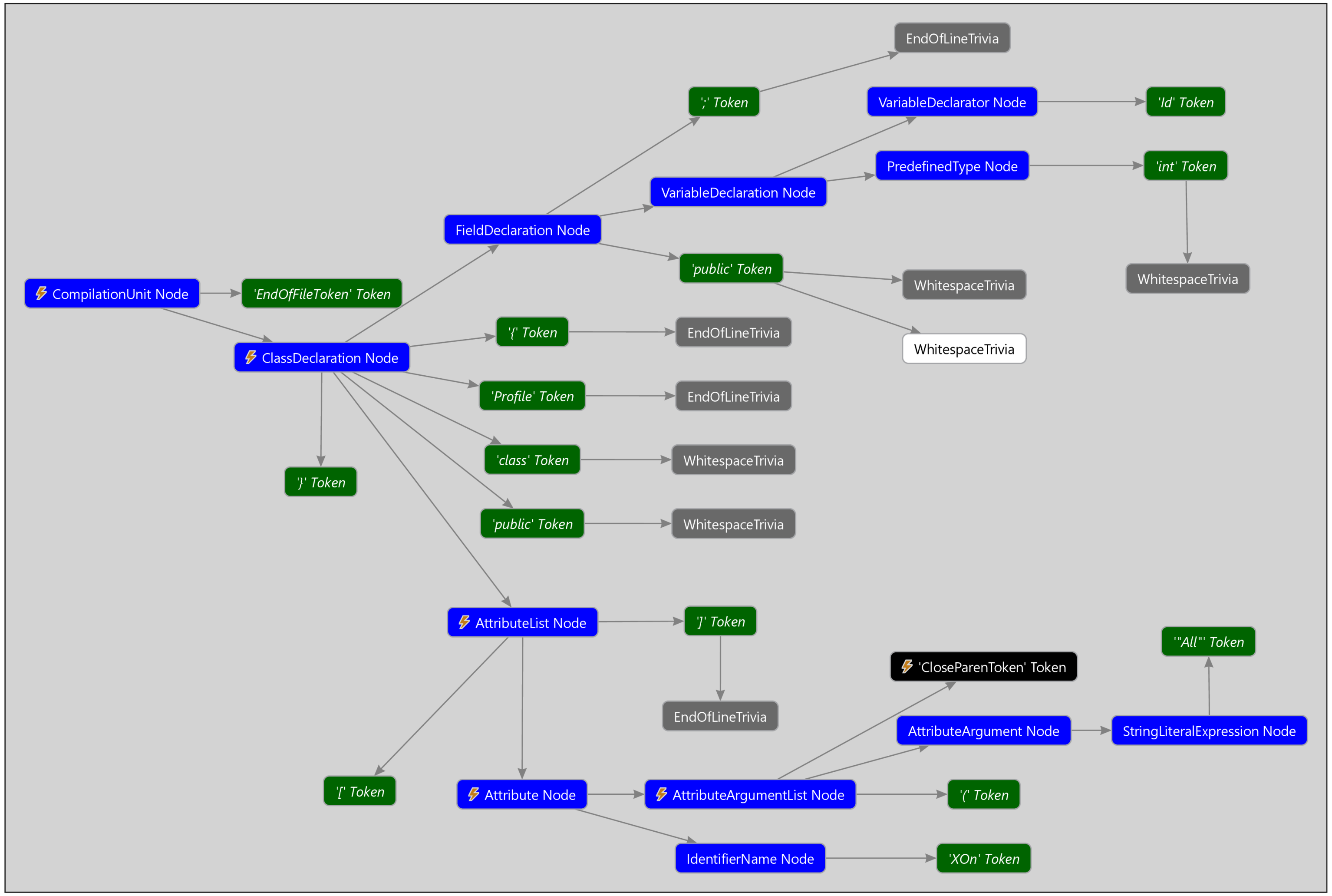


Figure A.1: Abstract Syntax Tree

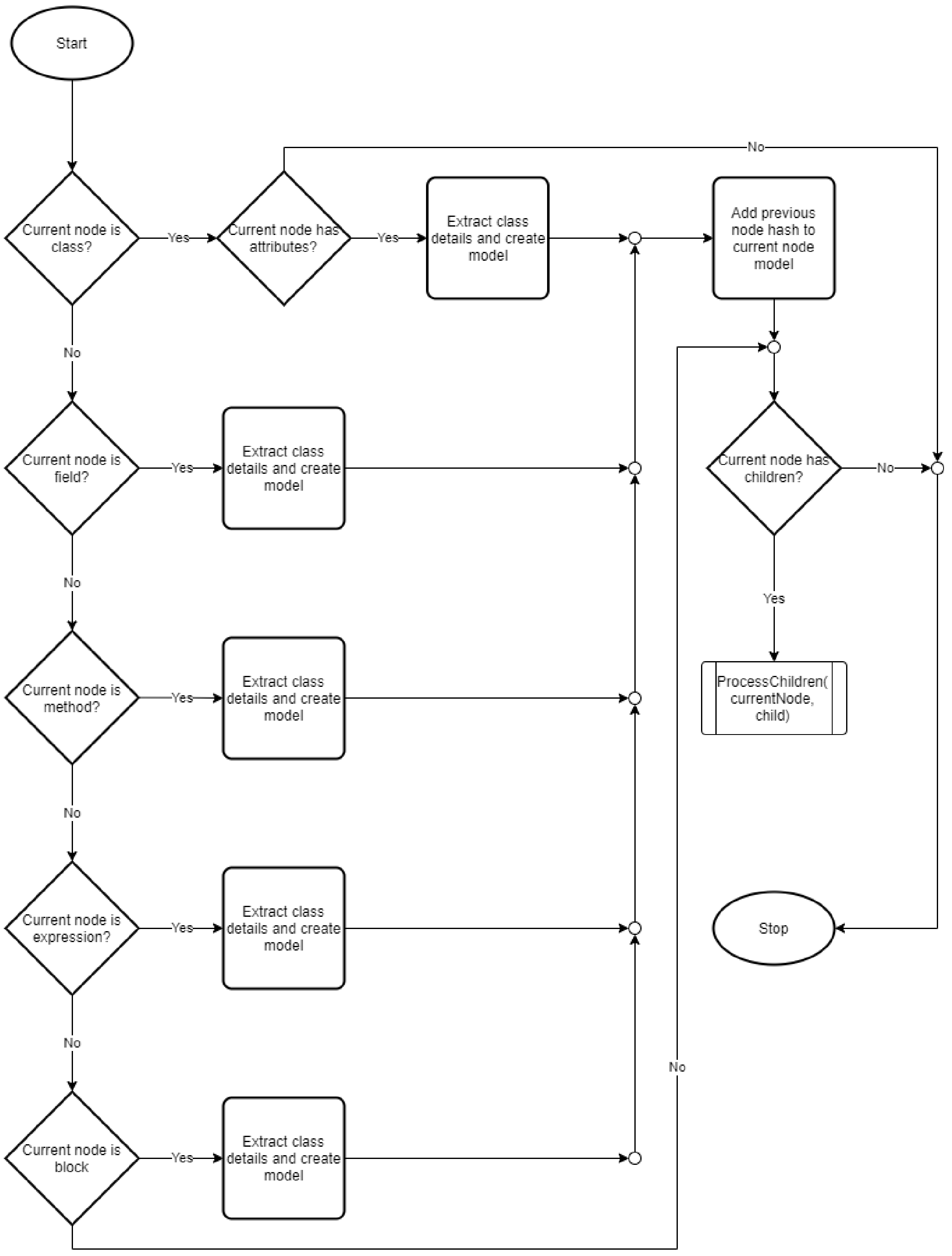

Start
Current node is class?
Yes
Current node has attributes?
Yes
Extract class details and create model
No
Add previous node hash to current node model
No
Current node is field?
Yes
Extract class details and create model
Current node has children?
No
No
Current node is method?
Yes
Extract class details and create model
Yes
ProcessChildren(currentNode, child)
No
Current node is expression?
Yes
Extract class details and create model
Stop
No
No
Current node is block
Yes
Extract class details and create model

Figure A.2: Lexical Analysis Flowchart

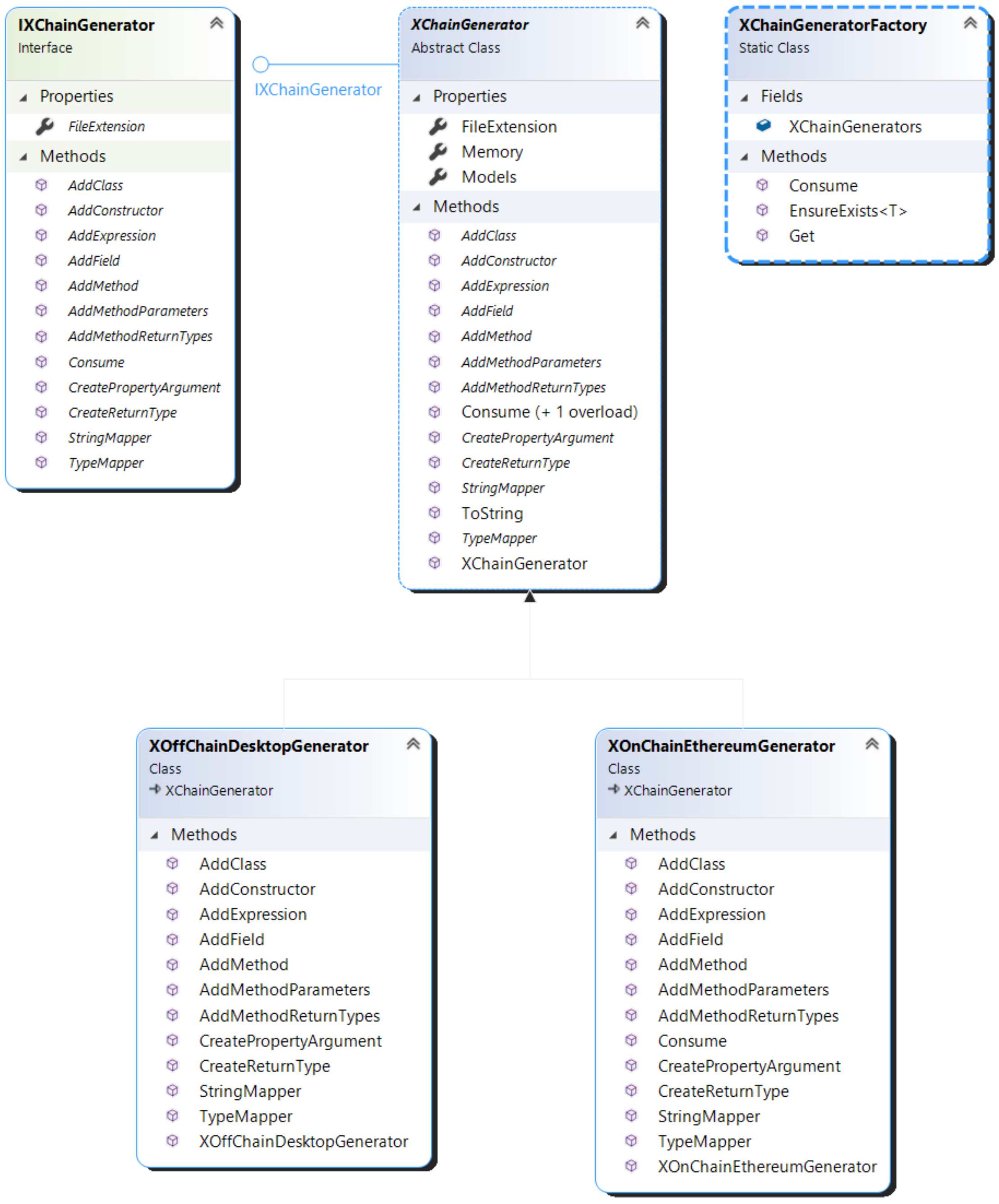


Figure A.3: XChainGenerators UML Diagram

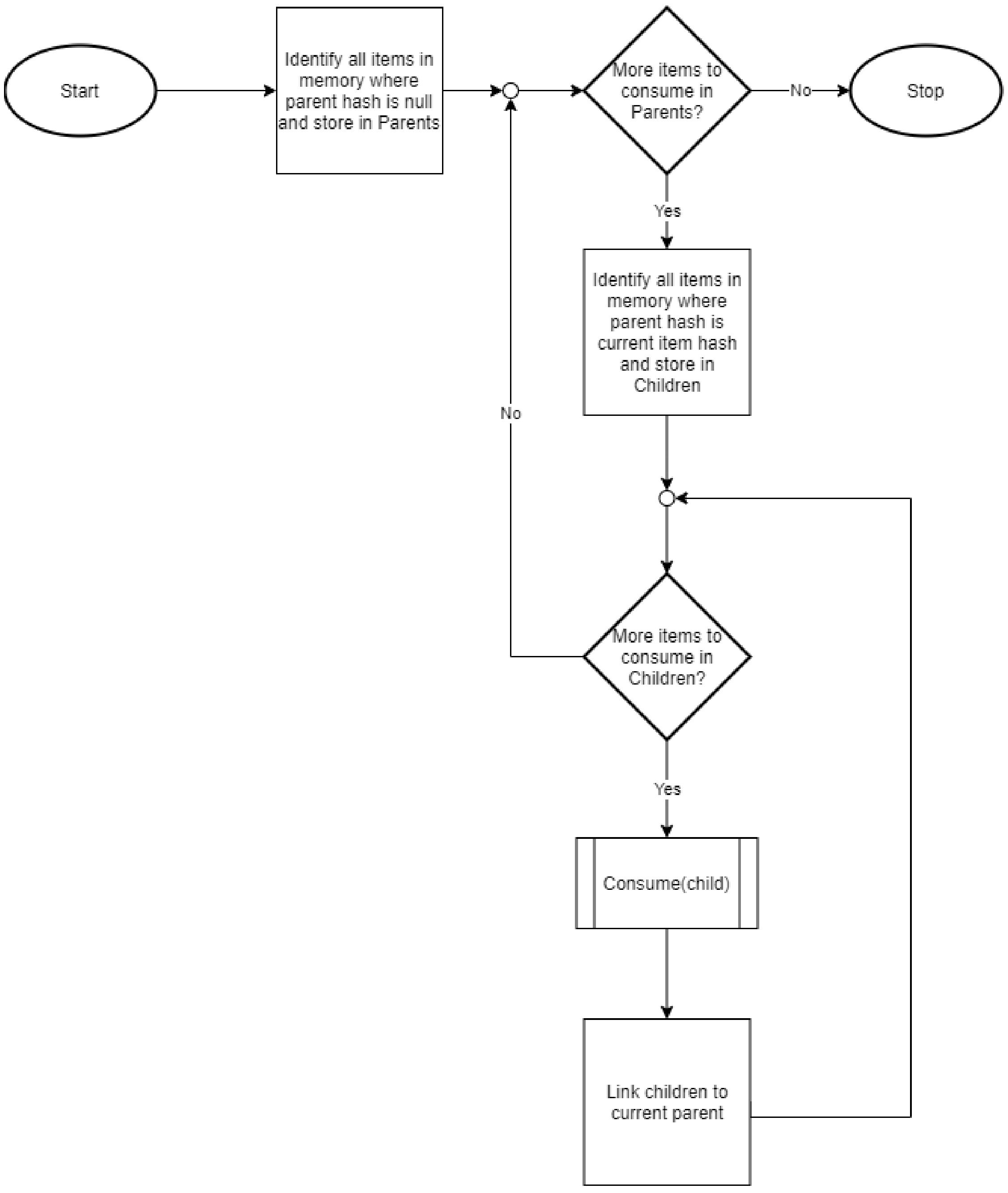

Start
Identify all items in memory where parent hash is null and store in Parents
More items to consume in Parents?
No
Stop
Yes
Identify all items in memory where parent hash is current item hash and store in Children
No
More items to consume in Children?
Yes
Consume(child)
Link children to current parent

Figure A.4: Source Generation Flowchart

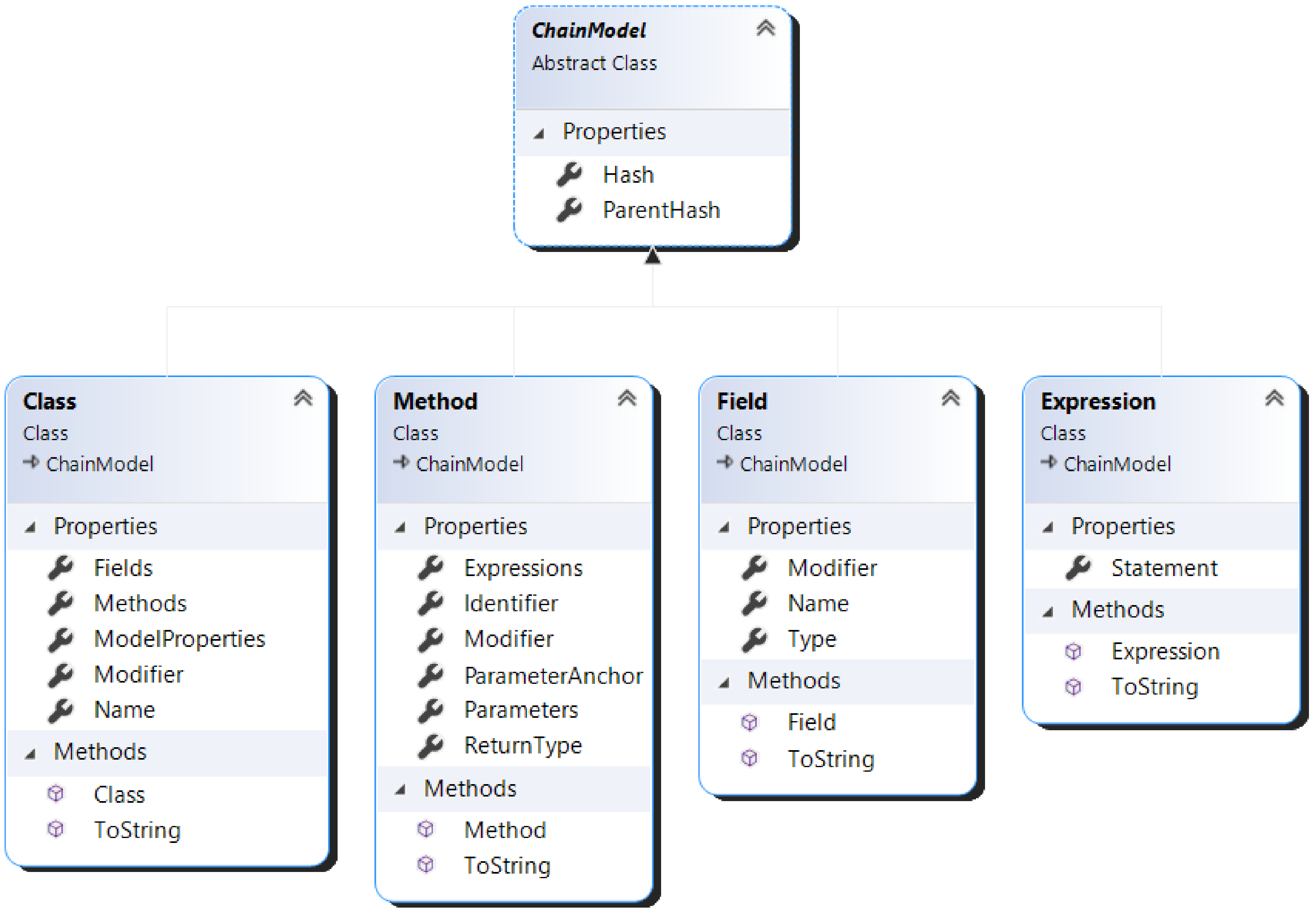


Figure A.5: Off-Chain UML Models

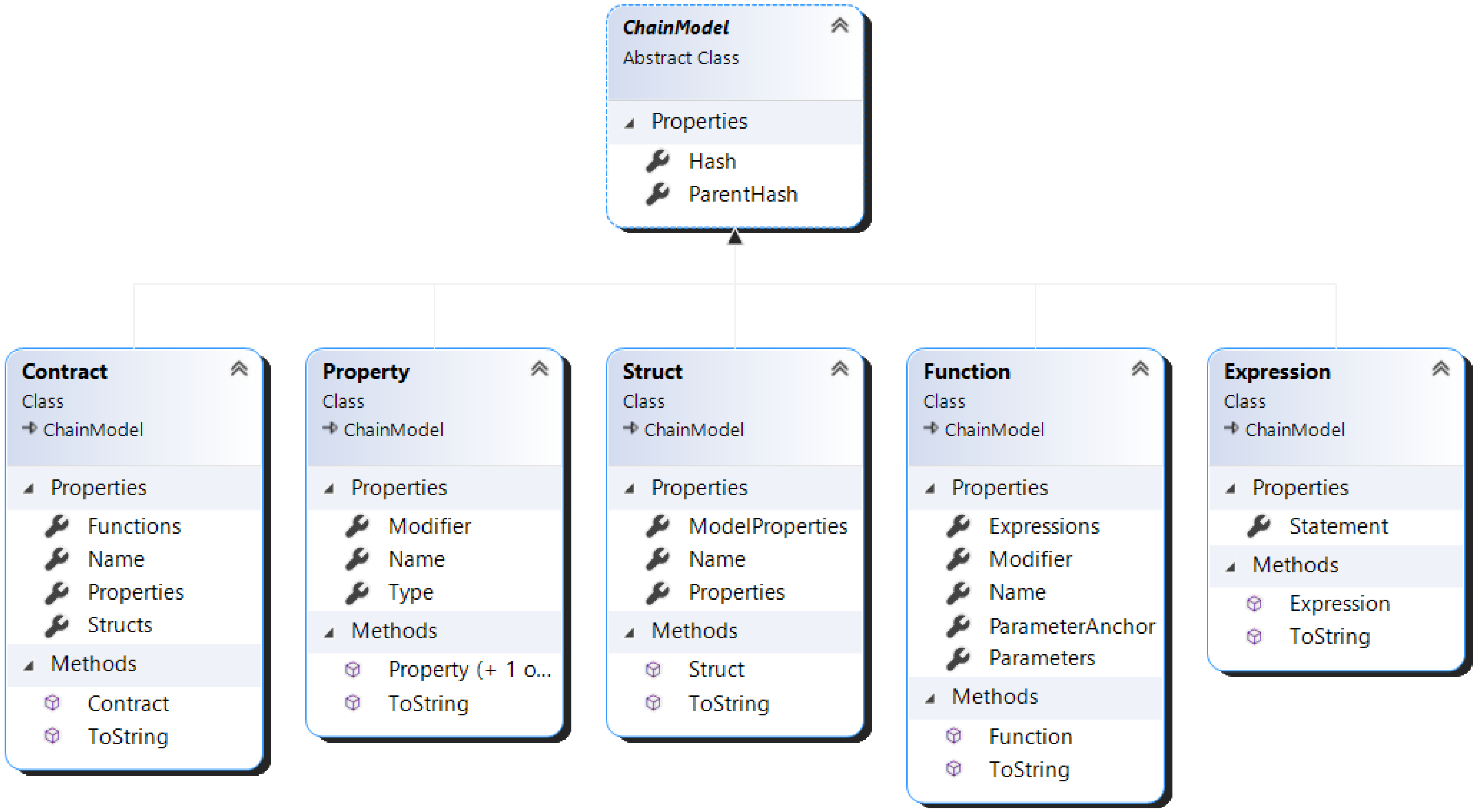


Figure A.6: On-Chain Ethereum UML Models

References

Ahrendt, W., Chimento, J. M., Pace, G. J., & Schneider, G. (2015). A specification language for static and runtime verification of data and control properties. In *International symposium on formal methods* (pp. 108–125).

Awan, A., Jagannathan, S., & Grama, A. (2007). Macroprogramming heterogeneous sensor networks using cosmos. *ACM SIGOPS Operating Systems Review*, *41*(3), 159–172.

Azzara, A., Alessandrelli, D., Bocchino, S., Petracca, M., & Pagano, P. (2014). Pyot, a macroprogramming framework for the internet of things. In *Proceedings of the 9th ieee international symposium on industrial embedded systems (sies 2014)* (pp. 96–103).

Bayer, D., Haber, S., & Stornetta, W. S. (1993). Improving the efficiency and reliability of digital time-stamping. In *Sequences ii* (pp. 329–334). Springer.

Beal, J., Pianini, D., & Viroli, M. (2015). Aggregate programming for the internet of things. *Computer*, *48*(9), 22–30.

Beal, J., & Viroli, M. (2016). Aggregate programming: From foundations to applications. In *International school on formal methods for the design of computer, communication and software systems* (pp. 233–260).

Buterin, V. (2016). Chain interoperability. *R3 Research Paper*.

Colombo, C., Pace, G. J., & Schneider, G. (2009). Larva-a tool for runtime monitoring of java programs. In *Ieee computer society* (pp. 33–37).

Cong, L. W., & He, Z. (2019). Blockchain disruption and smart contracts. *The Review of Financial Studies*, *32*(5), 1754–1797.

Data Protection Act. (2018). Chapter 586 of the Laws of Malta. Retrieved 2021-09-04, from `https://idpc.org.mt/wp-content/uploads/2020/07/CAP-586.pdf`

Ellul, J., & Pace, G. (2019). Towards a unified programming model for blockchain smart contract dapp systems. In *2019 38th international symposium on reliable distributed systems workshops (srdsw)* (pp. 55–56).

Ellul, J., & Pace, G. J. (2018). Alkylvm: A virtual machine for smart contract blockchain connected internet of things. In *2018 9th ifip international conference on new technologies, mobility and security (ntms)* (pp. 1–4).

Guizzardi, G., Pires, L. F., & Van Sinderen, M. (2005). An ontology-based approach for evaluating the domain appropriateness and comprehensibility appropriateness of modeling languages. In *International conference on model driven engineering languages and systems* (pp. 691–705).

Gummadi, R., Gnawali, O., & Govindan, R. (2005). Macro-programming wireless sensor networks using kairos. In *International conference on distributed computing in sensor systems* (pp. 126–140).

Hamdaqa, M., Metz, L. A. P., & Qasse, I. (2020). Icontractml: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. In *Proceedings of the 12th system analysis and modelling conference* (pp. 34–43).

Jones, S. P., Eber, J.-M., & Seward, J. (2000). Composing contracts: an adventure in financial engineering. *ACM SIG-PLAN Notices*, *35*(9), 280–292.

Lima, C. (2018). Developing open and interoperable dlt\/blockchain standards [standards]. *Computer*, *51*(11), 106–111.

Mizzi, A. (2019). Macroprogramming using an embedded dsl approach.

Mizzi, A., Ellul, J., & Pace, G. J. (2018). Macroprogramming the blockchain of things. In *2018 ieee international conference on internet of things (ithings) and ieee green computing and communications (greencom) and ieee cyber, physical and social computing (cpscom) and ieee smart data (smartdata)* (pp. 1673–1678).

Mizzi, A., Ellul, J., & Pace, G. J. (2019). Porthos: Macroprogramming blockchain systems. In *2019 10th ifip international conference on new technologies, mobility and security (ntms)* (pp. 1–5).

Moore, C. H., & Leach, G. C. (1970). Forth–a language for interactive computing. *Amsterdam: Mohasco Industries Inc*.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, 21260.

Narayanan, A., Bonneau, J., Felten, E., Miller, A., & Goldfeder, S. (2019). Bitcoin and cryptocurrency technologies. *Curso elaborado pela*.

Newton, R., & Welsh, M. (2004). Region streams: Functional macroprogramming for sensor networks. In *Proceeedings of the 1st international workshop on data management for sensor networks: in conjunction with vldb 2004* (pp. 78–87).

Page, I. (1996). Constructing hardware-software systems from a single description. *Journal of VLSI signal processing systems for signal, image and video technology*, *12*(1), 87–107.

Parizi, R. M., & Dehghantanha, A. (2018). Smart contract programming languages on blockchains: An empirical evaluation of usability and security. In *International conference on blockchain* (pp. 75–91).

Pathak, A., & Prasann, V. K. (2006). Issues in designing a compilation framework for macroprogrammed networked sensor systems. In *Proceedings of the first international conference on integrated internet ad hoc and sensor networks* (pp. 7–es).

Raskin, M. (2016). The law and legality of smart contracts.

Seijas, P. L., & Thompson, S. (2018). Marlowe: Financial contracts on blockchain. In *International symposium on leveraging applications of formal methods* (pp. 356–375).

Sherman, A. T., Javani, F., Zhang, H., & Golaszewski, E. (2019). On the origins and variations of blockchain technologies. *IEEE Security & Privacy*, *17*(1), 72–77.

Swan, M. (2015). *Blockchain: Blueprint for a new economy.* O'Reilly Media, Inc.

Szabo, N. (1994). Smart contracts.

Tyurin, A. V., Tyulyandin, I. V., Maltsev, V. S., Kirilenko, I. A., & Berezun, D. A. (2019). Overview of the languages for safe smart contract programming. , *31*(3), 157–176.

Wood, G. (2014). *Solidity.* Retrieved 2021-09-04, from `https://solidity.readthedocs.io/en/latest/`

Yaga, D., Mell, P., Roby, N., & Scarfone, K. (2019). Blockchain technology overview. *arXiv preprint arXiv:1906.11078*.