Extending the Concert Framework to Verify Solana Programs

João Maria Correia Ramalho Marcelino Gomes

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. João Fernando Peixoto Ferreira
Prof. Pedro Miguel dos Santos Alves Madeira Adão

November 2022

Acknowledgments

I am extremely grateful to my parents, brother, and family for helping and supporting me throughout my life.

Many thanks to both my supervisors, Professor João Ferreira, and Professor Pedro Adão, for all the guidance, feedback, and encouragement they gave me throughout the thesis.

To my closest friends thank you for your support, friendship, and motivational speeches.

I would be remiss in not mentioning my classmates, thank you for your moral support.

Abstract

Smart contracts are programs stored on a blockchain that help the agreement between two parties without involving an external trusted party. Since smart contracts may carry huge amounts of cryptocurrency and cannot be modified once deployed to the blockchain, it is crucial to ensure their correctness and that they are bug and vulnerability-free. Solana is a recent blockchain that features smart contract capabilities. This blockchain is rapidly growing due to its low transaction fees and speed and is seen as an Ethereum competitor.

In this thesis, we adapt and extend the already existing smart contract verification framework ConCert developed in Coq. Our extension allows the embedding of Solana contracts by introducing Solana concepts, like accounts and ownership. It is also possible to extract smart contracts written in Coq to Rust, programs that are nearly ready to be deployed onto the Solana blockchain. The extended ConCert allows the verification of existing Solana contracts and the development of verified contracts.

The extended framework was evaluated by measuring the proofs accomplished in the model, measuring the complexity of contracts that the framework was able to embed and extract, the time to extract, and the overhead of the extraction. We observed that due to the expressiveness of our model we were unable to embed complex contracts and, as a result, were not verified nor extracted. However, we were able to embed and extract simple contracts with a negligible overhead in terms of extraction time but a noticeable overhead after the extraction.

The main contribution of this thesis is thus the possibility of writing verified contracts in Coq and extracting them to the Solana blockchain.

Keywords

Blockchain, Smart Contracts, Solana, Formal Verification, ConCert, Coq

Resumo

Smart Contracts são programas guardados numa blockchain que ajuda o acordo entre dois participantes sem envolvimento de um terceiro confiável. Uma vez que, *Smart Contracts* contém enormes quantidades de criptomoedas e não podem ser modificados uma vez enviados para a blockchain é crucial assegurar o seu correto funcionamento e que e estes não possuem bugs nem vulnerabilidades. Solana é uma blockchain recente que possui a funcionalidade de *Smart Contracts*. Esta blockchain apresenta um crescimento rápido em consequência da sua velocidade e baixa taxa de transação e, por estas razões, é visto como um concorrente à Ethereum. Nesta tese, adaptamos e estendemos a framework de verificação de *Smart Contracts* em Coq, ConCert. A nossa extensão permite a escrita de contratos Solana introduzindo ao modelo existente conceitos de Solana, tais como, *accounts* e *ownership*. É também possível extrair *Smart Contracts* escritos em Coq para Rust e, uma vez extraídos, e após pequenos correções estão prontos para ser enviados para a blockchain Solana. O ConCert estendido permite a verificação de contratos Solana existentes e o desenvolvimento de contratos verificados. A *framework* extendida foi avaliada através da avaliação das provas conseguidas no modelo, avaliando a complexidade dos contratos que a *framework* conseguiu representar e extrair, o tempo de extração e o *overhead* obtido na extração. Verificou-se que, devido à expressividade do nosso modelo, contratos complexos não são possíveis representar e, por consequência, não foram verificados nem extraídos. Contudo, contratos mais simples fora representados e extraídos. Nestes foi observado um tempo de extração insignificante e um *overhead* de extração considerável. Desta forma, esta dissertação introduz a possibilidade de escrever contratos verificados em Coq e extraí-los para a blockchain Solana.

Palavras Chave

Blockchain, Smart Contracts, Solana, Verificação Formal, ConCert, Coq

Contents

List of Figures

List of Tables

Listings

Acronyms

AST	Abstract Syntax Tree
BPF	Berkeley Packet Filter
CIC	Calculus of Inductive Constructions
DeFi	Decentralized Finance
DLT	Distributed Ledger Technology
DSL	Domain-Specific Language
EVM	Ethereum Virtual Machine
FT	Fungible Token
FSCL	Functional Smart Contract Language
NFT	Non-Fungible Token
PCUIC	Polymorphic Cumulative Calculus of Inductive Constructions
PDA	Program Derived Account
PoH	Proof-of-History
PoS	Proof-of-Stake
PoW	Proof-of-Work
STLC	Simply Typed Lambda Calculus
TPS	Transactions per Second

1 Introduction

Contents

1.1 Motivation

Blockchain technology has captured the interest of both researchers and the industry. A blockchain is a distributed ledger technology that allows transactions to be committed without a trusted third party. Ethereum was the first blockchain that separated the consensus layer from the execution of smart contracts.

Smart Contracts are programs that are executed in blockchains, like Ethereum, and allow transactions of resources between parties. Even though these programs tend to be short, it is easy to accidentally introduce errors during development. Smart contracts often control large amounts of cryptocurrency, as such, the presence of one single bug/vulnerability can lead to large financial losses. As a result of these high-value transactions, smart contracts are becoming more appealing to attackers, and as such, there have been some attacks that lead to huge losses, e.g. TheDAO [4], Parity's multi-signature wallet [5] and, more recently, Nomad's crypto bridge [6]. There has been an effort by the community to introduce formal methods and formal verification to this domain. Some examples are the formalization of Yul [7], Ethereum's intermediate language, and the Mi-Cho-Coq framework [8] that aims to verify tezos smart contracts.

Solana is a blockchain written in Rust that achieves consensus using the Proof-of-History mechanism. Both Solana and Ethereum have smart contract capabilities (used for Decentralized Finance (DeFi) and Non-Fungible Tokens (NFTs)), and with the recent rapid expansion of Solana's ecosystem, they have inevitably been compared. Solana is a recent blockchain hence, as far as we know, there are no published solutions that aim to verify smart contracts in its ecosystem.

The goal of this thesis is to formally verify smart contract properties for the Solana blockchain using the Coq proof assistant. For that, we will use the smart contract verification framework ConCert[1] [2,3,9], together with Coq and its specification language Gallina we will implement and verify smart contracts. These verified contracts can then be extracted into a set of target languages, including multi-paradigm languages, web development languages, and functional languages. Despite ConCert being prepared to extract for Rust, it is not yet prepared to extract to the Solana blockchain. In this work, we also extend ConCert so it is able to embed Solana contracts and subsequently extract them to Solana.

1.2 Problem

Once a contract is deployed onto the chain, in most cases, it is impossible to modify it. This feature combined with the lack of smart contract verification leads to unwanted bugs and vulnerabilities and, as a result, provides attackers with an attack vector. Without the use of smart contract verification techniques, bugs and vulnerabilities will keep on appearing and causing huge financial losses to both

[1] https://github.com/AU-COBRA/ConCert

companies and users of the blockchains. However, many different tools can and are being used to verify smart contracts for distinct blockchains [10].

Since Ethereum is the leading blockchain with smart contract capabilities, several tools can be used to verify contracts written in Solidity (Ethereum's smart contract language), e.g Etherscan[2] [11], an Ethereum blockchain explorer that also offers a source code verification service, Sourcify[3], is another tool for source code verification that is open-source and decentralized that aims to be the infrastructure for other verification tools, etc.

Solana is a recent Ethereum competitor that thrives on the premise of having high transactions per second and low transaction fees. However, being a recent blockchain, Solana, to the best of our knowledge, has no tools in its ecosystem capable of verifying smart contracts. We aim to fill this gap by extending an already existing framework, developed for a different blockchain, in a way that it is possible to encode most of the existing smart contracts and verify and deploy them to the chain. The ability to verify smart contracts before deploying them on the Solana blockchain makes it possible to reduce the number of contracts deployed with bugs and vulnerabilities onto the chain.

1.3 Objectives

Taking into consideration the number of bugs and vulnerabilities present in smart contracts and the huge financial losses they cause, the focus of this thesis is to develop a framework capable of embedding, executing, verifying, and extracting smart contracts for the Solana blockchain. More precisely, we aim to:

1. Extend the ConCert framework execution layer by allowing it to represent the Solana blockchain, along with some of its fundamental concepts, and Solana contracts;
2. Extend the ConCert framework extraction layer, more exactly, develop a new Rust extraction tailored to the Solana blockchain, such that they can be deployed onto it;
3. Implement smart contracts using the new execution layer, make use of them to exhibit the newly created Coq to Solana extraction and the new features in the ConCert framework, and formally verify some contract-specific properties.

1.4 Contributions

The main contributions of this thesis are:

[2]https://etherscan.io/verifyContract
[3]https://github.com/ethereum/sourcify

- An extension to the ConCert smart contract verification framework. More specifically, an extension of the already existing ConCert execution model to allow Solana programs to be written in Coq. Furthermore, this extension allows the verification of small correctness properties of the developed contracts.
- An extension to the ConCert Rust extraction. The current ConCert's Rust extraction is aimed at the Concordium blockchain and cannot be directly used for other chains. Our extraction extension allows for contracts developed in Coq to be printed in Rust that targets the Solana blockchain. The extracted code will then be deployed onto the chain.
- Two small case studies demonstrating that our extension is capable of developing and partly verifying contracts and extracting them to Rust.

All our code and case studies are available at `https://github.com/siimplex/ConCert`.

1.5 Dissertation Outline

This document is structured as follows. Chapter 2 presents the background of blockchain and smart contracts, an analysis of the blockchain Solana and of the programming language Rust, and an overview of Coq and two frameworks MetaCoq and ConCert. In Chapter 3, an overview of existing works related to program extraction, execution of languages, smart contract verification, and verification in Solana. In Chapter 4, the pipeline of the proposed tool, and the developed execution model are presented and, lastly, the extraction to Solana and its details are discussed. In Chapter 5, the execution model is evaluated, as two case studies of implemented and extracted contracts and a comparison with the framework that served as the basis for this tool. Finally, Chapter 6 concludes this document, including limitations and future work.

2 Background

Contents

This chapter describes some essential concepts required to grasp the entirety of the thesis. This chapter begins by describing how blockchain technology work, its main components (Section 2.1), and how the smart contract functionality works (Section 2.2). Then, it describes the Solana blockchain (Section 2.3) followed by the language used to write its programs, Rust (Section 2.4). Lastly, the proof assistant used will be described followed by the main framework used by ConCert (Section 2.5), and finally a deep look at the ConCert framework and its layers (Section 2.6).

2.1 Blockchain

A distributed ledger is a digital infrastructure that enables the secure use of a decentralized database. A Distributed Ledger Technology (DLT) uses nodes to record, share, replicate, and synchronize transactions, these nodes are the participants of the network. New transactions are broadcasted by nodes and once consensus is achieved between the participants an update is shared throughout the network so they can update their copy of the ledger. The security and accuracy of the records stored in each ledger are assured thanks to the use of keys, digital signatures, and cryptographic protocols. Furthermore, by using consensus there is no central point of failure, and as such a malicious user must work considerably harder to attack the system. Hence, against individual attackers, the system is highly secure. This technology, if applied correctly, has significant potential to revolutionize many industries, such as music and entertainment, artwork, supply chains, health, and finance [12]. In finance, more specifically, it can reduce transaction costs, ease auditability and increase transparency.

A blockchain is a form of distributed ledger technology where the committed transactions are stored in a chain of blocks that is endlessly growing. This concept was first introduced in 1991 by Stuart Haber and W. Scott Stornetta [13] and later, in 2008, the Bitcoin model was proposed, but only in 2009, it was implemented by Satoshi Nakamoto. Bitcoin was the first decentralized blockchain implementation using a peer-to-peer network to solve the double-spending problem [8], i.e. when a single coin is spent simultaneously more than once.

A blockchain is an ever-growing list of blocks, containing one or more transactions, that are linked using cryptographic hashes. Each block carries the hash value of the previous block Figure 2.1 which, in addition to linking two blocks, also is a mechanism to prevent the tampering of blocks. A Unix timestamp is added when the block joins the network, serving as a source of variation for the block hash, making it harder to manipulate the blockchain. A nonce is calculated iteratively until it is lower than a target. If a miner finds a nonce that when hashed is lower or equal then the target difficulty, then the block can be added to the chain (this process is know as Proof-of-Work (PoW)). Finally, the block holds a hash of all of the transactions in the block as a Merkle tree root hash, allowing the participants of the peer-to-peer network to easily verify the integrity of the block, preventing block tampering.

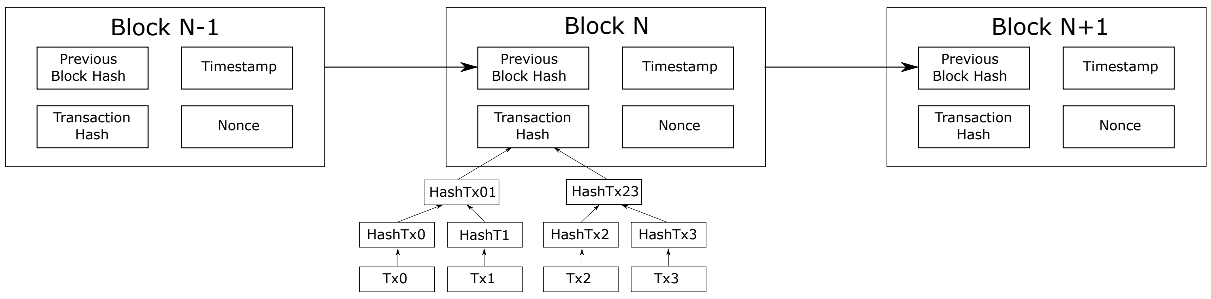

Figure 2.1: Simplified Bitcoin blockchain block structure, from Wikipedia [1]

With today's high computational capacity where a single computer can compute thousands and thousands of hashes per second using hashing as block security is just not enough to prevent tampering. As such, to mitigate this problem, blockchains use different consensus mechanisms like PoW and Proof-of-Stake (PoS) to regulate the creation of blocks. These systems vary depending on the blockchain, e.g. Bitcoin's PoW makes each miner compute a hash until it finds a correct one by using CPU power, Cardano's PoS uses considerably less CPU power and the validating capability depends on the stake in the network.

There are three types of Blockchain implementations, i.e., public, private, and hybrid. Public Blockchains, like Bitcoin, allow everyone to participate in it, be it by reading or sending transactions or joining the consensus process. If a node is honest and performs its duty as intended, it will be rewarded monetarily. Since anyone can join public Blockchains its main goal is to prevent the concentration of power. In a PoW public Blockchain, should a pool of miners gain control of 51% of the hash rate they could theoretically control what is being added to the chain [14]. Because the attackers possess the majority of the mining power they could add blocks to the chain which may include fraudulent transactions designed for financial benefit. Private Blockchains, like Hyperledger Fabric, value privacy allowing only vetted notes to participate. This type of Blockchain is not decentralized, it works as a closed database secure with cryptographic protocols and operates according to an organization. The organization behind the Blockchain can set the read and write permissions of the participants as it sees appropriate. Hybrid Blockchains amalgamate the best out of the public and private types resulting in a private Blockchain that has its ledgers' data verified with the help of a public Blockchain.

2.2 Smart Contracts

Smart contracts were proposed in the 1990s by Nick Szabo [15, 16], long before the creation of Bitcoin, and made great progress after 2013 when the altcoin Ethereum [17] appeared. Smart contracts are just like traditional contracts in the way that they both can be seen as an agreement with specific terms between two or more entities. These digital contracts are programs that keep the agreement terms between the buyer and the seller directly into lines of code, they automatically execute only when set predefined conditions are met.

The contracts are stored on a blockchain-based platform which allows the enforcement of terms of an agreement between parties without the need for a trusted third party. Once contracts are on the chain they inherit the chain's immutable property, in the sense that once created and deployed it is not possible to modify or tamper with the code of the contract.

According to Zibin Zheng [18], smart contracts compared to traditional contracts have three advantages, reduced fraud risks, lower administration and service costs, and increased business efficiency. As previously said, smart contracts gain immutability once deployed to the chain, hence they cannot be arbitrarily modified after being deployed which reduces the risks of tampering. Since blockchain records every transaction that is distributed throughout the network, financial fraud can be easily detected, furthermore reducing these types of risks. Blockchains achieve trust through the consensus mechanism without the need for a trusted third party. Smart contracts are automatically executed, i.e. does not need a third party, which allows for cutting costs. Since there is no need for an intermediary the agreement is automatically done once the preconditions are met, without the need for a third party.

The Smart Contracts life cycle can be split into four phases [19] (see Figure 2.2): creation, deployment, execution, and completion.

1. Creation: This phase has two sub-phases starting with reiteration and negotiation, and ending with contract implementation. Much like traditional contracts, the negotiation phase agrees on the objectives of the contract. Once the contents of the smart contract are agreed upon and finalized, they are converted into code.

2. Deployment: In this phase, the contract is stored in the blockchain. A small fee must be paid to the miners in exchange for this service, which per se is a way to prevent flooding of the blockchain ecosystem with smart contracts. Also in this phase, digital assets from the participating parties are locked by freezing their digital wallets.

3. Execution: The contract stored in the distributed ledger is read by the participating nodes, which verify its integrity and validate it. The compiler or interpreter then executes the code. After receiving inputs for the execution from one of the parties, a transaction is triggered and broadcasted to the network, which results in it being appended to the existing chain through the consensus mechanism.

4. Completion: After the execution of the smart contract the states of the involved parties are updated. Once the chain has the updated state information, the transactions sent from the contract

execution, and the consensus mechanism has verified asset transfer the digital assets are finally unfrozen. Thus, concluding the whole life cycle.

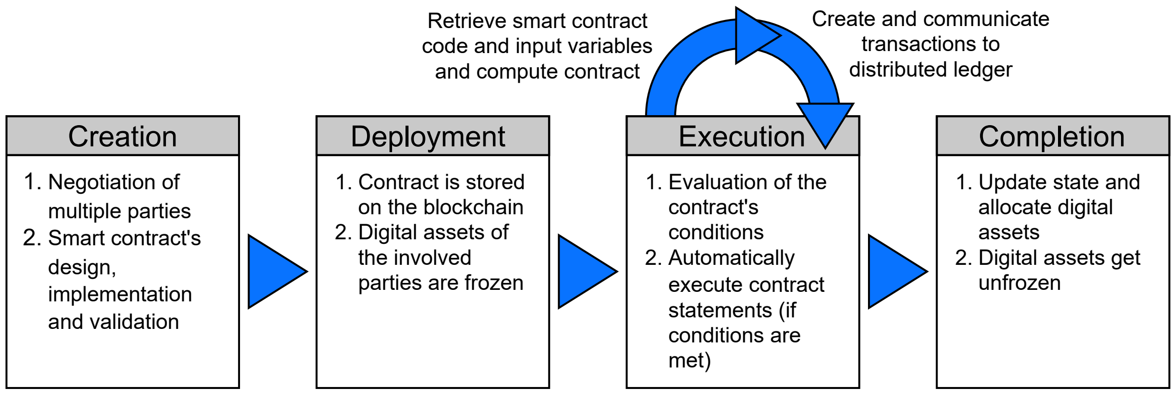
Retrieve smart contract code and input variables and compute contract
Create and communicate transactions to distributed ledger
Creation
1. Negotiation of multiple parties
2. Smart contract's design, implementation and validation
Deployment
1. Contract is stored on the blockchain
2. Digital assets of the involved parties are frozen
Execution
1. Evaluation of the contract's conditions
2. Automatically execute contract statements (if conditions are met)
Completion
1. Update state and allocate digital assets
2. Digital assets get unfrozen

Figure 2.2: The four phases of a smart contract life cycle: Creation, Deployment, Execution, and Completion

Ethereum was the first decentralized open-source blockchain to launch smart contract functionality. It is currently the most used altcoin and the second-best cryptocurrency only falling behind Bitcoin. This blockchain-based platform aims to give an alternative protocol to building distributed applications, or dApps, whilst providing different sets of trade-offs that emphasize development time, security, and the ability to efficiently interact with other dApps. To develop smart contracts and dApps, Ethereum provides a virtual general-purpose machine, Ethereum Virtual Machine (EVM), where the smart contracts are executed in stack-based machine language. But, since low-level is not that convenient to the programmer, most smart contracts are written in higher-level languages, for example, Solidity.

Ethereum is the current leader in DeFi but, in the past few years, other promising altcoins appeared, one of them being Solana.

2.3 Solana Blockchain

Solana [20] just like its competitor Ethereum, is a blockchain-based platform that allows the development of smart contracts and dApps. Solana has been steadily growing, in the year 2021 SOL (Solana's token) went up more than 10000% according to CoinGecko[1].

Solana claims to be the fastest-growing ecosystem in the crypto environment and the fastest blockchain on the planet. There are two main reasons for this, high Transactions per Second (TPS), which is owed to its PoS jointly with Proof-of-History (PoH) and its asynchrony, and secondly, it has a really low cost per transaction compared to its competitors, e.g., gas fees in Ethereum. PoS is a consensus algorithm

[1] https://www.coingecko.com/en/coins/solana

in which all the network stakeholders agree on the validity of the shared data and secure that data on the blockchain. The blockchain network can only move on to a new block of data once the previous one is secured. Solana, unlike Ethereum and Bitcoin's PoW, uses PoS with PoH. PoS can be seen as an evolution in consensus mechanisms because its less energy-intensive than PoW and provides a major increase in speed and efficiency.

Each node in a decentralized blockchain network processing transactions has its own internal clock. With the immense amount of nodes around the world, there ought to be some slight local clock discrepancies. This can prove problematic when the network needs to reach a consensus about which transactions occurred and their chronological order. Consensus mechanisms such as PoS and PoW both suffer from this time synchronization problem. Each node timestamps transactions according to its local system clock, the discrepancies from every node in the network, even the good ones, can lead to attacks from malicious actors. To solve this problem, Solana employs what is known as PoH. PoH is a stack of proofs that provides a way to prove that some data existed before the proof was created. This function runs in sequence and then stores the number of times it has been called and, just like PoW, the output of PoH can be recomputed and verified by external participants.

The Solana ledger stores records, called accounts, that can either hold data or an executable program. Each account is addressable by a key, that is referred to as a public key (pubkey), and may contain funds called Lamports - a Lamport is a fractional native token that is worth 0.000000001 SOL. In Solana, only the owner of an account is capable of modifying it. The accounts previously mentioned are attributed to a key pair and are addressable by its public key, however, there is another type of account called Program Derived Account (PDA). PDAs are accounts that are owned by programs, hence they are not controlled by a private key like the aforementioned accounts. The latter account type is the foundation for cross-program invocation, i.e., enabling the interaction between different Solana applications.

Traditional EVM-based blockchains combine both logic and state into a single contract. Solana takes a slightly different path. Solana's smart contracts are read-only [20], meaning that they do not contain any state data, only contain program logic. Once a contract is deployed on-chain it can be accessed by external accounts. During these interactions, if allowed, the program may store information on the accounts that initiated the interaction.

Program execution in Solana starts with a transaction being submitted to the cluster (Figure 2.3). A transaction consists of an array of signatures of a given message followed by the message itself. The message contains a header that contains the number of required signatures, the number of read-only account addresses, and the number of read-only account addresses that do not require signatures. This header is followed by an array of account addresses, a recent block hash - representing the last time the client observed the ledger - and finally an array of instructions. Each instruction specifies one program using its program id; the specified program can then return successfully or with an error code, in the latter the whole transaction fails.

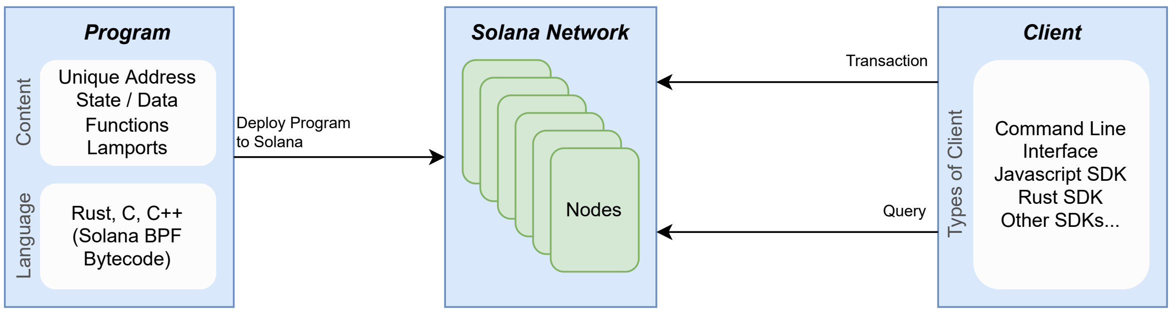

Figure 2.3: Representation of Solana contract content and workflow

Solana on-chain programs are compiled using the LLVM compiler infrastructure [21] to an ELF file containing a variation of the Berkeley Packet Filter (BPF). Hence smart contracts can be written in any programming language that can target the LLVM compiler, such as C, C++, and Rust. Using these programming languages it is possible to develop high-performance smart contracts, moreover, Rust solves issues of memory safety and thread concurrency.

2.4 Rust

Rust[2] is a statically-typed, multi-paradigm, general-purpose programming language that focuses itself on performance and safety. It began in 2006 as a side project of Graydon Hoare [22], an employee at Mozilla, and its first stable version was officially released in 2015 [23]. After, and even before, its official release it began steadily rising in the programming language ranks.

Rust is a fast and memory-efficient language that has many benefits that are needed by all sorts of software. It capitalizes on having no runtime or garbage collection allowing it to focus on other crucial services, thus increasing its speed. Rust has some similarities with C, for instance, both languages do not have garbage collection and they share some keywords and commands. Furthermore, in terms of syntax Rust is similar to C and C++. These languages are also comparable in terms of performance. When compiled, Rust code, native machine code is generated, this code is meant to have the same level of performance as C and C++. However, the program will only be compiled if there is no unsafe memory usage.

Rust controls its memory management through its *ownership* and borrowing system. In this system, each value has a unique owner but ownership can be transferred between objects. With this system, every bit of memory can be tracked and it is the reason why Rust does not need memory garbage-collection. Rust features several tools that facilitate the development process, such as `Cargo`, `Clippy`,

[2] `https://www.rust-lang.org/`

and `Rustfmt`. `Cargo` is Rust's build system and package manager, it is responsible for downloading, compiling, and distributing packages (called *crates*). `Clippy` is a collection of *lints* that is used to catch mistakes thus improving the overall correctness, performance, and readability of the code. `Rustfmt` is a tool used to format code in order to meet Rust code formatting standards.

Smart Contract development takes advantage of most of Rust's properties. Rust is type-safe, memory safe and it generates small binaries hence, allows the development of safe and efficient smart contracts that also contain low overhead.

2.5 Coq and MetaCoq

Coq[3] is a formal proof management system that allows interactive theorem proving. It provides a higher-level language called Gallina that is based on Calculus of Inductive Constructions (CIC) [24] - a formal language that combines both a functional programming language and a higher-order logic - and an environment that allows reasoning about behaviors. Common Coq applications include defining functions and evaluating them, stating mathematical theorems and interactively formally proving them, certifying properties of existing programming languages, and extracting certified programs to other programming languages such as OCaml, Haskell, and Scheme.

Coq's architecture can be split into two levels [25]. The first level is relatively small and is based on a language with few primitive constructions, typing rules, and computation. On top of this level, there is a richer environment that helps design theories/lemmas and proofs. Everything done in this second level will ultimately be safe because the safe level will check every proof and definition that was designed. Furthermore, the latter level offers several features, such as libraries, extendable notation, and tactics that can be used for proof automation.

MetaCoq[4] [26] is a project formalizing Coq in Coq constituted by several subprojects. MetaCoq provides meta-programming capabilities allowing manipulation of Coq code at the meta-level. At the MetaCoq project's center is the Template Coq quoting library for Coq which is extended with additional features: PCUIC; Safe Checker; Erasure and Translations. The most important features for our project are Template Coq, PCUIC, and Erasure which are used by the ConCert Framework explored in Section 2.6.

Template Coq [27] is a plugin for Coq that adds meta-programming functionalities. That is, it takes Coq terms *quotes* them, which results in an Abstract Syntax Tree (AST) that is represented as an inductive data type based on the Coq kernel's term representation. Since the related data types in Template Coq and Coq's kernel are very close, the *quoting* and *unquoting* procedures are straightforward. These meta-programming facilities allow for plugin development in Coq and, since it is fully developed in Coq,

[3]https://coq.inria.fr/
[4]https://github.com/MetaCoq/metacoq

it is possible to verify the plugin's implementation.

Polymorphic Cumulative Calculus of Inductive Constructions (PCUIC) is a variant of CIC which as stated previously, is the foundation of Coq's higher-level language Gallina. PCUIC representation is a cleaned-up version of the kernel representation used by Template Coq, moreover, it is equivalent to the one of Coq. MetaCoq allows the translation between the two PCUIC and Template Coq. This representation allows for formalizing the meta-theory of Coq in Coq itself. A PCUIC term can go through the Safe Checker subproject which allows for an inexpensive verified reduction of the term, followed by a conversion check and a type check.

Safe Checker is a sound type-checker for PCUIC. To achieve this sound type-checker both a verified reduction machine and a conversion checker had to be implemented.

Erasure is an erasure procedure for untyped lambda calculus that yields the same result as the extraction plugin in Coq. This procedure takes as an input a PCUIC term and produces an untyped lambda calculus term. During this procedure, the PCUIC term passes through many verified optimizations and reductions transforming higher-order constructs to first-order blocks to ease translation to other programming languages.

2.6 ConCert Framework

ConCert[5] is a smart contract verification framework written in Coq for Coq. ConCert [2] allows the embedding of functional languages into Coq by using meta-programming. In ConCert, there are two ways of representing functional programs: as an AST, deep embedding, and as a Coq function, shallow embedding. Each representation has its advantages: deep embedding is fitting for meta-theoretical reasoning whilst shallow embedding is fit for proving the properties of concrete programs. To connect these two representations of functional programs ConCert uses some of MetaCoq's facilities.

The overview of the structure of the ConCert framework can be split in two, the pipeline used to embed an Functional Smart Contract Language (FSCL) contract to a contract in Coq (in Figure 2.4) and the pipeline used to extract a contract written in Coq (in Figure 2.5).

The ConCert framework is split into three layers. The Embedding Layer, which features the embedding of smart contracts into Coq, λ_{smart}, together with its proof of soundness, and the Execution Layer and the Extraction Layer. The latter two are the layers most relevant to our work.

2.6.1 Embedding Layer

The Embedding layer contains an embedding of smart contracts into Coq along with its proof of soundness, using the MetaCoq project. This layer contains the definition and the interpreter of the previously

[5]https://github.com/AU-COBRA/ConCert

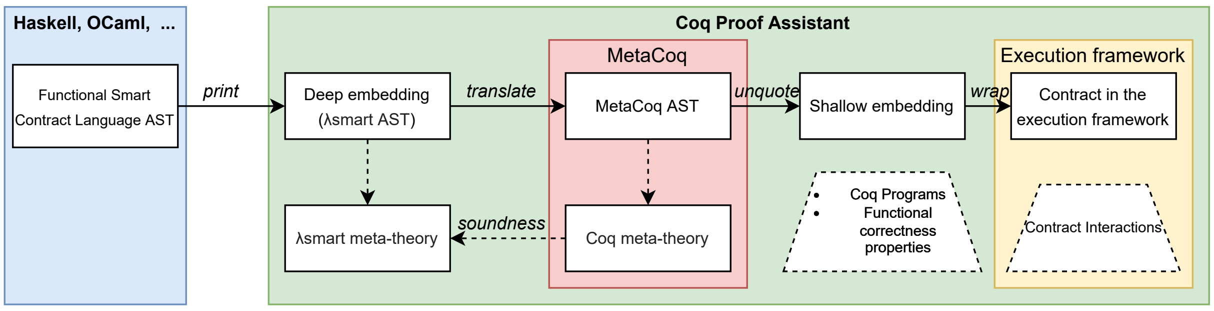

Figure 2.4: Simplifed pipeline from a FSCL to a contract in Coq, using the ConCert Framework [2]

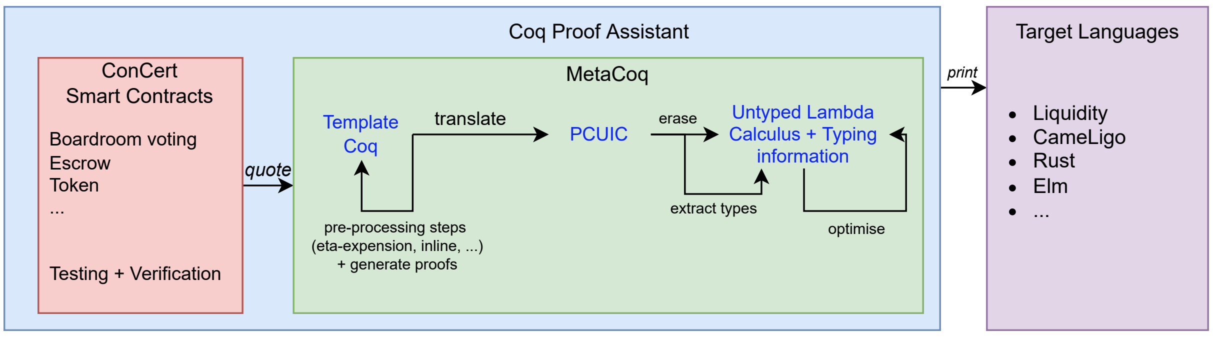

Figure 2.5: Simplifed pipeline of the extraction of a contract written in Coq, using the ConCert Framework [3]

mentioned language, λ_{smart}, moreover, this layer shows the soundness of the translation from the λ_{smart} language to PCUIC AST.

We start the embedding procedure (Figure 2.4) with an AST of a program written in a functional programming language, like Haskell or OCaml. The first step is to print this program. Printing is a recursive procedure that converts the AST (of a functional programming language) into a string built from the constructors of the corresponding inductive types in Coq (deep embedding). ConCert comes with a language that is the embedding of a core subset of the ACORN smart contract language in Coq, represented by λ_{smart}, which comes with some features of a functional language. The translation of a λ_{smart} AST to a MetaCoq AST is done by structural recursion, and the resulting term is a PCUIC representation. This translation is done using a global environment that includes all inductive type definitions present in λ_{smart} expressions. With the translation done, it is still left to show that these translations are sound. To prove soundness, i.e. to show that the semantics of λ_{smart} expressions agree with their corresponding translation to MetaCoq, ConCert resorts to a comparison between the results of the evaluation of λ_{smart} expressions with the weak-head call-by-value evaluation relation provided by MetaCoq.

Finally, this layer provides a few embedded contracts available both in deep embedding (as ASTs) and in shallow embedding (as Coq functions). The latter one allowing for straightforward reasoning about contracts' properties.

2.6.2 Execution Layer

The Execution Layer provides a model with which is possible to reason about contract execution traces, making it possible to state and prove the properties of interacting smart contracts. In this model smart contracts are comprised of two functions: `init` (Listing 2.1) and `receive` (Listing 2.2). The `init` function is called after the smart contract is deployed onto the blockchain.

```
    init : Chain → ContractCallContext → Setup → option State
```

Listing 2.1: Contract init function

The first parameter of type Chain represents the blockchain, the second parameter ContractCallContext contains all the data about the call to the contract, i.e., who initiated the transaction, the address that sent the call, the address of the contract being called, the balance of the contract being called and the number of tokens being passed in the call. The third parameter of type Setup contains user-defined information to be used in the function. Lastly, this function returns None if it fails, and returns the initial state if it succeeds.

After being deployed, each time the contract gets called the `receive` function is executed.

```
    receive : Chain → ContractCallContext → State → option Msg
                 → option (State * ActionBody)
```

Listing 2.2: Contract receive function

The first two parameters are the same as the ones in the `init` function. The third parameter, with type State, contains the current state of the contract. Msg is a message type previously defined by the user. Finally, if the function succeeds it returns the resulting state with a list of actions to execute if any. The action can only be one of three types: transfers, calls to other contracts, or contract deployment.

Since both `init` and `receive` are Coq functions it is possible to reason about them and prove contract properties. But reasoning about an isolated contract is not enough, since a contract can call other contracts. To simulate the scheduler of a real blockchain the implementation of ConCert uses a relational specification of a scheduler, which is defined as follows[6].

```
ChainedList (Point : Type) (Link : Point → Point → Type)
      : Point → Point → Type :=
    clnil : forall p : Point, ChainedList Point Link p p
    | snoc : forall from mid to : Point,
        ChainedList Point Link from mid →
        Link mid to → ChainedList Point Link from to.

Definition ChainTrace := ChainedList ChainState ChainStep.
```

Listing 2.3: ChainedTrace and ChainList definition

[6] https://github.com/AU-COBRA/ConCert/blob/master/execution/theories/Blockchain.v

where ChainStep contains the three different steps, `step_block`, `step_action`, and `step_permute`, which correspond respectively to adding a block to the blockchain, executing one of the previously mentioned actions, and changing the order of the actions to be executed (changing the action scheduled for execution).

2.6.3 Extraction Layer

Finally, the Extraction Layer [3] provides facilities to extract a smart contract written in Coq into a program in a FSCL. To extract to different FSCLs, ConCert maps the abstractions of the Execution Layer to the corresponding abstractions in the target FSCL. This layer works as an interface between the smart contracts written in Coq and the extraction functionality. ConCert currently allows the extraction to Liquidity [28], CameLIGO [29] (both smart contract languages), Midlang, Elm (web programming), and Rust (a multi-paradigm language with a functional subset) [30]. To extract smart contracts they must go first through a process of erasure, and to achieve this ConCert uses MetaCoq erasure with extensions. MetaCoq's verified erasure procedure will erase any type of term into a box, and in specific cases, it can erase an entire term into a box. ConCert [2] erasure procedure extension generates type annotations, which are needed to help with the typing of the target languages. Without these type annotations, there would be ambiguities that cannot be solved by the type checker. Also in this first extension, the erasure procedure for type schemes allows for handling type aliases, i.e., Coq definitions that return a type, which is present in the standard library. The second extension of MetaCoq's verified erasure is *deboxing*, which consists of an optimization process that removes boxes that were left behind by the erasure step. After the erasure process, with extensions and the optimization process, the optimized code is pretty-printed to the target language.

2.6.4 Rust Extraction

Rust[7] is a multi-paradigm general-purpose language, it is aimed to be fast and guarantee memory safety. Rust is a low-level language that provides a lot of control. With all these features Rust is an appealing smart contract programming language that is already being used in Blockchains such as Solana[8]. Extracting to Rust is not easy and it encompasses some challenges. A few challenges are met while extracting data types, for example, recursive data structures in Coq cannot be directly extracted to Rust, and to work around this indirection is used. Another major challenge comes from the different memory models used by Rust and Coq. Rust does not have a garbage collector while Coq's memory allocation is done implicitly. There are a few more such as handling polymorphic functions (functions with type parameters), partial applications, internal fixpoints, etc. Currently, ConCert allows the extraction to

[7] https://github.com/rust-lang/rust
[8] https://github.com/solana-labs/solana

Rust, more specifically, to the Concordium Blockchain. However, this extraction does not allow us to extract to the Solana blockchain, which is one of our goals.

3 Related Work

Contents

Blockchain and Smart Contracts are growing fields of research. Both technologies have been studied a lot by different research communities and their applications keep expanding. Their increasing applications bring many challenges, leading to many solution proposals and studies that further develop these technologies. This section will delve into some studies and challenges related to Blockchain and Smart Contract extraction and verification. As previously said, to the best of our knowledge, there are not any articles about Smart Contract verification in Solana.

In Section 3.1 we look at some related work that is directly related to the extraction that was previously presented. In Section 3.2, we will observe works from different authors that compile a dependently typed language to a low-level representation. In Section 3.3, we look at different smart contract verification techniques. Finally, in Section 3.4, we look at different works related to the blockchain.

3.1 Extraction to Statically Typed Languages

Here extraction means getting source code in a target language, the source code must be accepted by the language compiler and must also be able of being integrated into the targeted systems. Coq [31, 32] is one of the several proof assistants that facilitates the extraction of functional languages such as Haskell and OCaml (from Coq proofs or programs). Agda is another proof assistant capable of extraction, it has mechanisms for defining custom backends, but the GHC backend is the most approved.

The ConCert framework [2] extends Coq's proof assistant extraction [3, 28] using MetaCoq's erasure [30]. The current ConCert extraction allows the extraction to a multi-paradigm general-purpose language (Rust), functional smart contract languages (Liquidity, Simplicity), and a functional language for web development (Elm). Coq proof assistant's current extraction is not verified due to unverified optimisations that are done during the extraction process. This separation makes it difficult to compare it with the formal procedure given by Letouzey [31], on the other hand, ConCert's separation between erasure and optimisation facilitates such comparisons.

Isabelle [33] is a generic proof assistant that focuses on formal verification (e.g. proving the correctness of systems and proving properties of programs). Isabelle/HOL is an application within the generic framework that provides higher-order logic theorem proving. Isabelle/HOL can produce code for a subset of ML (OCaml, Haskell). Hupel and Nipkow [34] extend the possible extractions and develop a verified compiler from Isabelle/HOL to CakeML and, just like MetaCoq, it implements meta-programming capabilities for quoting.

The MetaCoq project [35] aims to formalise projects in Coq and features a verified erasure subproject that is the basis for the ConCert framework extraction.

Šinkavors and Cockx [36] present an approach for embedding and extraction procedures in Agda that combines both shallow and deep embedding. This combination is possible by leveraging the power of Agda's *reflection* (similar to TemplateCoq's quote/unquote mechanism), this allows to keep benefits from both shallow and deep embeddings. The approach presented by the authors is fine-grained, i.e., the embeddings do not cover the entire language and, as such, it is best suited for Domain-Specific Languages (DSLs). At the moment, there are no guarantees that the semantics of the original implementation is preserved in the extracted code because Agda's meta-theory is not formalised. Since Coq is also a theorem prover equipped with similar capabilities as Agda, it could prove interesting to adapt this approach, and, with MetaCoq's formalisation, it would be possible to reason about semantic preservation.

3.2 Execution of Dependently Typed Languages

Related works in this section are related to compiling code from a dependently-typed language to low-level code.

In [9] Nielsen and Spitters present a model specification of smart contracts in Coq, that features inter-contract communication and allows modeling both depth-first execution blockchains and bread-first execution blockchains. This work later served as the basis for the execution layer of ConCert in [2].

Œuf [37] is a verified compiler of a restricted subset of Coq's specification language Gallina. The source language is a lambda-lifted Simply Typed Lambda Calculus (STLC), with neither pattern-matching nor user-defined inductive types.

In [38] Pit-Claudel et al. present a new approach to sound program extraction of **DSLs!** (**DSLs!**) embedded in Gallina to a correct-by-construction program written in an imperative intermediate language that can be compiled to assembly code. This approach uses Coq's tactic language to drive compilation, avoiding the need for an embedded language before compiling, which differs from ConCert's approach (and the approach used by this project). Furthermore, this approach extraction is implemented directly onto the proof assistant itself, as opposed to ConCert's approach which extraction relies on plugins.

The CertiCoq project [39] is a verified compiler for Coq which closes the gap between certified high-level systems and compiled code in machine language. CertiCoq uses the MetaCoq project quoting capabilities, it uses Template-Coq at the first step and verified erasure at the first step. After a few more steps C code is produced and later compiled with CompCert certified compiler [40] to the machine target language.

3.3 Smart Contract Verification

In the presented solution, smart contracts are verified by using the Theorem Proving technique in the Coq proof assistant, but this technique is not the only one that can be used to verify smart contracts.

There are several smart contract verification techniques, Model Checking, Theorem Proving, Program Verification, Symbolic Execution, and Runtime Verification and Testing. According to the survey [10], Model Checking (i.e. automatically verifying a system model against its specification) is one of the most used techniques, and Program Verification. In our project, we use the Theorem Proving technique and thus it's the more relevant one for this project.

In Theorem Proving, the use of a proof assistant like Coq, Isabelle/HOL [41] or Agda [42], makes it easier to prove properties of smart contracts with theorems/lemmas. In [43] Arusoaie uses Coq proof assistant to formally verify financial derivatives (written in a purely declarative in a DSL - Findel) by developing an infrastructure that allows to formalise and prove properties that, if proved, will exclude several potential vulnerabilities. Mi-Cho-Coq [44], a Coq framework that formalises the Michelson language (used in the Tezos blockchain), implementing a Michelson interpreter and a way to encode language expressions. Furthermore, it uses the weakest precondition calculus defined as a proven correct function.

Chapman et al. [45] develop a System *F*, which is a typed λ-calculus that extends the simply-typed λ-calculus. Using Agda, Chapman et al. present one of the first System *F* that is intrinsically typed and formalised.

Yul (also called JULIA) is an intermediate-level language that can be compiled into bytecode. In [7] Li et al. compare the formalisation of Yul with machine-level and user-level languages. They further extend the existing Yul formalisation and verify properties guaranteed by the ERC-20[1] token contract using Isabelle/HOL. In [46], the EVM is formalized in Lem - language that can be compiled for interactive theorem provers, and some safety properties are proven using Isabelle/HOL.

Kongmanee in [47] presents models for smart contracts that are language and system independent. With the results from the NuSMV tool [48], they attempt to identify design flaws in smart contracts before their development.

Sergey et al. [49, 50] describe Scilla, an intermediate-level language for verified smart contracts (for Zilliqa blockchain) that facilitates the separation between smart contract communication and the programming aspect. Scilla aims to be minimalistic while achieving sufficient expressivity and tractability. Scilla semantics has been formalized using Coq proof assistant as a shallow embedding.

3.4 Solana Contract Verification

The related works in this category are concerned with discovering vulnerabilities and verifying smart contracts in Solana.

Pierro et al. [51] develop a web tool capable of verifying the ownership of smart contracts by comparing the source code of the contract, written in a high-level language, with the bytecode deployed in

[1] https://ethereum.org/en/developers/docs/standards/tokens/erc-20/

the Solana blockchain. This tool yields some interesting results, the program address advertised by the owner corresponds to the bytecode in the blockchain, thus increasing the trust of the users on the blockchain.

Tavu in his thesis [52] investigates real-smart contracts to discover common vulnerabilities. These vulnerabilities are generally known by Solana developers, and to automate this process verification tools can be used. The author uses different tools with different approaches and run they against real-world contracts, finding vulnerabilities in some of them.

4 Solana Programs Written in Coq

Contents

This chapter presents the proposed solution—a tool for smart contract verification by construction for the Solana blockchain. First, the pipeline of the tool is presented (Section 4.1). Afterward, an overview will delve deeper into the Solana Execution Model written in Coq (Section 4.2). Lastly, it is shown an overview of the extraction mechanism used to extract Coq contracts to Rust Solana programs (Section 4.3).

4.1 Pipeline

In order to develop the proposed tool in this thesis, it is necessary to grasp the context of Blockchain and Smart Contract Technologies. More specifically, the Solana Blockchain and its Program structure (which will be delved further into in Section 4.2).

This thesis aims to provide a tool that can be used by Solana Smart Contracts developers' giving them the possibility to develop verified contracts. Hence, contributing to the overall safety of the Solana ecosystem.

The pipeline is, in fact, similar to the original ConCert Framework pipeline, and covers the majority of the process to produce a verified contract by construction to the Solana Blockchain. The process starts with the representation of a smart contract in Coq (the contract must have been designed and developed before entering the pipeline) and ends with the extraction to the target language, in our case Rust. The tool pipeline can be seen in Figure 4.1 where the contributions have been highlighted in italic bold and green.

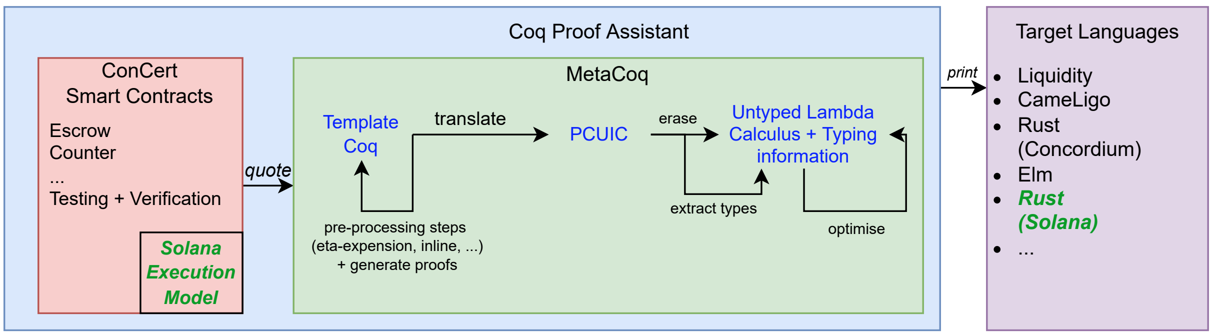

Figure 4.1: The tool pipeline, starting from a written contract in Gallina and ending with the extraction to the target language.

To start the pipeline, it is necessary to write a smart contract in Gallina using the new Execution Model. During this phase, it is possible to test and verify the properties of the written contracts. The remaining passes, i.e. quoting, translation, erasure, and optimisations, were all borrowed in their entirety from the original ConCert's pipeline. This denotes that MetaCoq [35] is also an integral part of this tool. Furthermore, the certifying passes and optimisations are still applied to mean that the new execution model did not affect this transformation/optimisation phase.

To finalize the pipeline process, the optimised code can be pretty-printed using the already existing pretty-printer developed directly in Coq. Like ConCert, this tool's target languages include Rust, which is both a general-purpose multi-paradigm language and a smart contract language. The original framework's Rust pretty-printer is directed to the Concordium Blockchain. This tool uses a new pretty-printer to Rust (based on the already existing one) that has been fine-tuned to be able to print deployable contracts for the Solana Blockchain.

Once this process is done, we get a Rust program that has the same functionality as the one written in Coq. After some minor changes (borrow checker and lifetime related), the Rust program is accepted by the compiler and can be deployed to the chain and interacted with.

4.2 Solana Execution Model in Coq

This work is applied in the context of ConCert, a smart contract verification framework in Coq that aims to give blockchain users the possibility to write smart contracts and test and verify their properties.

4.2.1 Basic Implementation

To extend the existing ConCert execution model it is necessary to consider Solana core concepts [1] such as accounts, transactions, programs, rent, and calling between programs. Moreover, it is also necessary to consider Solana's Rust program support.

Solana's programs export a known entrypoint that is called when calling a program. This entrypoint takes a byte array that contains the serialized program parameters' (`program id`, `accounts`, and `instruction data`), which are deserialized and then used as parameters to call a user-defined instruction process function, returning the result.

As such, a Solana contract can be represented in Coq (Listing 4.2) as a function that receives the blockchain information (type `Chain`), an array of accounts (type `SliceAccountInformation`) and instruction data (`option Msg`): `Chain` (Listing 4.1) represents the view of the blockchain that a contract can access and interact with (e.g. current chain height, finalized height, etc);

```
Record Chain :=
  build_chain { chain_height : nat; current_slot : nat; finalized_height : nat; }.
```

Listing 4.1: Solana contract definition in Coq

`SliceAccountInformation` is a type alias for an array of `AccountInformation` records; and `option Msg` is the instruction data that has been deserialized into the `Msg` type. When this function is called it

[1] https://docs.solana.com/developers

can either be successful or it can fail with an error type, `ProgramError`, which differs depending on the contract error.

```
Record Contract (Msg State : Type) `{Serializable Msg} `{Serializable State} :=
  build_contract {
    process : Chain → SliceAccountInformation → option Msg
                → result unit ProgramError;
  }.
```

Listing 4.2: Solana contract definition in Coq

The Coq record `AccountInformation` (Listing 4.3) is a direct replica of the Solana Rust struct `AccountInfo` (Listing 4.4), with the exception of the field `rent_epoch` which is not relevant to this model. The field `account_state` in the execution model has type `SerializedValue` which is an intermediary storage type to facilitate polymorphism for different contract states [9]. This type is used in the definition of `WeakContract` (Listing 4.5) which is a weaker representation of the previously seen `Contract` type (Listing 4.2). Similar to ConCert, this representation allows dealing with the contracts in a generic manner. However, it makes reasoning harder, and for this reason, the `Contract` type is preferred to reason with, since it has no `SerializedValues` and has concrete types in their place.

```
Record AccountInformation :=
  build_account {
    account_address : Address;
    account_is_signer : bool;
    account_is_writable : bool;
    account_balance : Amount;
    account_state : SerializedValue;
    account_owner_address : Address;
    account_executable : bool;
  }.
```

Listing 4.3: Coq's AccountInformation definition

```
pub struct AccountInfo<'a> {
  pub key: &'a Pubkey,
  pub is_signer: bool,
  pub is_writable: bool,
  pub lamports: Rc<RefCell<...>>,
  pub data: Rc<RefCell<...>>,
  pub owner: &'a Pubkey,
  pub executable: bool,
  pub rent_epoch: Epoch,
}
```

Listing 4.4: Solana's AccountInfo definition

In addition to accounts and their state, it needs to be possible to interact with other programs and accounts. It is possible to do so by using monads and helper functions that ease the use of actions throughout contract behavior.

```
Inductive ActionBody :=
  | act_transfer (to : Address) (amount : Amount)
  | act_call (to : Address) (msg : SerializedValue)
  | act_deploy (c : WeakContract)
  | act_special_call (to : Address) (body : SpecialCallBody)
    with SpecialCallBody :=
      | transfer_ownership (old_owner account new_owner : AccountInformation)
      | check_rent_exempt (account : AccountInformation)
      | check_token_owner (account : AccountInformation)
    with WeakContract :=
      | build_weak_contract
          (process :
                Chain → (* chain *)
                SliceAccountInformation → (* accounts *)
                option SerializedValue → (* instruction data *)
                result unit ProgramError).
```
Listing 4.5: Solana Action Body definition in Coq

In this design the contracts can interact with the blockchain and other programs by transferring lamports (`act_transfer`), calling programs (`act_call`), deploying programs (`act_deploy`), and using special calls (`act_special_call`) (Listing 4.5).

Special calls is a newly added constructor that facilitates the execution of different context-specific functions. This way these context-specific actions do not clutter the `ActionBody` inductive. Moreover, it facilitates reasoning and extraction to Solana. In this design, there are three special actions: `transfer_ownership`, `check_rent_exempt`, and `check_token_owner`. Special action `transfer_ownership` will transfer ownership of one program, i.e. change the program's owner address. In addition to the destination address, it needs three accounts: the current contract owner, the contract that will transfer ownership, and the contract's new owner. Secondly, the special action `check_rent_exempt` checks if an account is rent exempt, i.e., if it has more than two years' worth of rent in lamports. Solana's rent concept was implemented using only this action which does not modify the blockchain environment. This check is usually done to check if an account will remain on the chain or if it will be deleted before the main functionality of the contract is started. Moreover, Rent is one of the many useful sysvars Solana exposes to its users[2]. Lastly, the special action `check_token_owner` is straightforward as it checks if the queried account's owner is the token program. And, just like the previous one, this special action does not modify the environment.

The `token program`[3] is an implementation of Fungible Token (FT) and NFTs on the Solana blockchain. As such it is not part of the core Solana concepts. However, it is an integral part of developing Solana contracts in Rust. Moreover, in Rust, the transfer ownership action requires the use of the token program.

Unlike the original ConCert execution model where they obtained a clear separation between contracts and their interaction with the chain, this separation is no longer that clear with this design. Actions are called in the middle of the program and their effects take place when they are executed.

The type `ActionBody` in its actions only has the destination address on other parameters depending on the action. Later, it is introduced the action representation which consists of the `ActionBody` and two addresses, origin and from, which in most cases will be the same.

```
Record Action :=
  build_act {
    act_origin : Address;
```

[2]https://docs.solana.com/developing/runtime-facilities/sysvars
[3]https://spl.solana.com/token

```
    act_from : Address;
    act_body : ActionBody; }.
```

Listing 4.6: Solana `Action` definition in Coq

The `Action` type resembles what is usually considered a transaction, but just like the authors of ConCert, the `Action` and `Transaction` (respectively Listing 4.6 and Listing 4.7) types are different [9]. The two aforementioned types differ from each other because an `Action` is done by a user and modifies the blockchain state. The latter can be seen as a fully specified action and additional information. For example, a `Transaction` in case of an `act_deploy` contains the program address that was created after the action has been executed.

```
Record Tx :=
  build_tx {
      tx_origin : Address;
      tx_from : Address;
      tx_to : Address;
      tx_amount : Amount;
      tx_body : TxBody;
  }.
```

Listing 4.7: Transaction definition in Coq

```
Inductive TxBody :=
  | tx_empty
  | tx_deploy (wc : WeakContract)
  | tx_call
     (msg : option SerializedValue)
  | tx_special_call
     (body : SpecialCallBody).
```

Listing 4.8: Transaction Body definition in Coq

Solana's transaction model allows for transactions to contain more than one instruction (or action) per transaction. To ease reasoning on transactions and their effects on the chain, our model restrained each transaction to a single instruction (or action). Moreover, the transaction format has been simplified from the Solana docs, specifically the transaction array of signatures which are used to check if a user authorized that transaction. Since, just like ConCert, we are not modeling most of the cryptographic aspects of Solana and focusing on the accounting aspects.

Finally, the Solana calling between programs concept can be achieved using the `act_call` in a program. Program-derived accounts are not explicitly implemented in the execution model like the other core concepts. However, they are implicitly used in the special action `transfer_ownership` when this action is converted into actual Rust code (Section 4.3 delves deeper into this concept and conversion from Coq inductive to Rust code).

In summary, the `Contract` and `WeakContract` definitions were modified. The record `AccountInformation` was added to represent the accounts. The record `Tx` remained the same, but the inductive type `TxBody` was modified according to the new actions added in the inductive `ActionBody`.

4.2.2 Semantics of the Extended Execution Layer

In this subsection we start by looking at the `Environment` which contains all the information related to the blockchain and accounts, then we present the `ActionEvaluation` which is used to evaluate the actions and their effects on the chain. Finally, we present the *ChainHelpers* class that is used to aid contract implementation and extraction.

The `Chain` type mentioned previously (in Listing 4.1) has some information about the blockchain but it is not enough to allow the blockchain to execute actions. It needs to be possible to look up deployed contract information. ConCert's original `Environment` definition allows the look up of the contract's functions and state. But, since Solana has some concepts that were not in the original design the environment was extended with look up functions for program ownership (`env_account_owners`) and program balance (`env_account_balances`) (Listing 4.9).

```
Record Environment :=
  build_env {
    env_chain :> Chain;
    env_account_balances : Address → Amount;
    env_account_owners : Address → option Address;
    env_contracts : Address → option WeakContract;
    env_contract_states : Address → option SerializedValue;
  }.
```

Listing 4.9: Environment definition in Coq

With this definition, we can define functions that update the environment. In Listing 4.10 it is defined as a function that allows changing the ownership of a contract, this is done by using a map. By sending the contract address, the address of the new owner, and the environment a new updated environment is produced where the contract is now owned by the new owner. Furthermore, the remaining environment fields can be updated in a similar way.

```
Definition set_account_owner (addr : Address) (new_owner : Address)
                                                      (env : Environment) :=
    env<|env_account_owners ::= set_chain_account_owner addr new_owner|>.
```

Listing 4.10: Account Owner setter definition in Coq

Like most of the concepts and definitions described in this section, action evaluation is also based on the original action evaluation in ConCert. When actions are executed it is necessary to evaluate the effects of the actions. This is defined as a "proof-relevant" relation `ActionEvaluation` in Coq, with type `Environment → Action → Environment → Type` [9].

This relation is defined by four cases: transfer tokens, contract deployment, contract calls, and special calls. In Listing 4.11, it is presented the details of the transfer case. The complete definition of this relation can be seen in Listing A.2.

```
Inductive ActionEvaluation
          (prev_env : Environment) (act : Action)
          (new_env : Environment) (new_acts : list Action) : Type :=
  | eval_transfer :
      forall (origin from_addr to_addr : Address)
            (origin_acc from_acc to_acc : AccountInformation)
            (amount : Amount),
      amount >= 0 →
      amount <= env_account_balances prev_env from_addr →
      account_address origin_acc = origin →
      account_address from_acc = from_addr →
      account_address to_acc = to_addr →
      address_is_contract to_addr = false →
      act = build_act origin from_addr (act_transfer to_addr amount) →
      EnvironmentEquiv
        new_env
        (transfer_balance from_addr to_addr amount prev_env) →
      new_acts = [] →
      ActionEvaluation prev_env act new_env new_acts
  | eval_deploy :
    ...
  | eval_call :
    ...
  | eval_special_call :
    ...
.
```

Listing 4.11: Action Evaluation relation in Coq

In the transfer case, the amount must be greater than zero and the sender must have enough money. The origin, from, and to addresses must correspond to some `AccountInformation` address. When these conditions are met, an `act_transfer` action is created and the old environment is affected and evaluated to a new environment where the balance of both accounts has been updated according to the amount sent. The transfer case does not result in any further action as it can be seen by the variable `new_acts`. And to finalize the evaluation relation we use `EnvironmentEquiv` to achieve equality between environments.

The special call case is evaluated as any of the other cases but has a specific evaluation relation `SpeciallCallBodyEvaluation` to evaluate the body of this call (Listing 4.12) it can be seen in Listing A.1.

```
Inductive SpecialCallBodyEvaluation
          (prev_env : Environment) (act : Action)
          (new_env : Environment) (new_acts : list Action): Type :=
  | eval_transfer_ownership :
     forall (origin from_addr to_addr acc_addr new_owner_addr : Address)
            (wc : WeakContract),
       env_account_owners prev_env acc_addr = Some origin ∨
         env_account_owners prev_env acc_addr = Some from_addr →
       env_contracts prev_env to_addr = Some wc →
       EnvironmentEquiv
         new_env
         (set_account_owner acc_addr new_owner_addr prev_env) →
       new_acts = [] →
       SpecialCallBodyEvaluation prev_env act new_env new_acts
  | eval_check_rent_exempt :
     ...
  | eval_check_token_owner : forall (origin from_addr : Address),
        ...
  .
```
Listing 4.12: Special Call Body Evaluation relation in Coq

For instance, in the transfer ownership special call, the origin and from address must be both the same address and owner of the contract and the contract must be deployed Then the environment is updated, more specifically its account ownership map is updated with the new information concluding the effects of this Special Call.

Now with the `Environment`, there is enough information to evaluate actions. ConCert further augments this type to keep track of actions to execute and also define the meaning of a step in a chain with three inference rules. This part of the execution model was not required to suffer any major changes and, as such, it was kept almost fully. The only exception was a few small changes to accommodate previously mentioned additions to the chain and some lemmas and theorems that needed to be adjusted to be accepted (Section 5.2 delves deeper into proofs that were modified and accomplished).

Finally, to help with the extraction process the class `ChainHelpers` was created. This class contains the function signatures that are to be used during contract implementation (Listing 4.13). The next section (Section 4.3) further explains the use of this class.

```
Class ChainHelpers :=
  build_helpers {
    next_account : SliceAccountInformation → Z →
                     result AccountInformation ProgramError;
    deser_data (A : Type) : SerializedValue → result A ProgramError;
    deser_data_account (A : Type) : AccountInformation → result A ProgramError;
    ser_data {A : Type} : A → SerializedValue;
    ser_data_account {A : Type} : A → AccountInformation →
                                  result unit ProgramError;
    exec_act : WrappedActionBody → result unit ProgramError;
  }.

Global Opaque next_account deser_data deser_data_account
              ser_data ser_data_account exec_act.
```
Listing 4.13: `ChainHelpers` class and functions in Coq

4.3 Solana Extraction from Coq

Coq's extraction mechanism targets functional programming languages like OCaml and Scheme. ConCert builds upon this mechanism and focuses on the area of smart contract languages (Liquidity) and general-purpose languages with a functional subset (Rust).

The extraction presented in this thesis is an extension of the extraction in ConCert. More specifically, the new extraction adds a new Rust pretty-printer that targets the Solana blockchain and makes some minor changes to the actual Rust extraction procedure.

4.3.1 Execution Model Helper Functions

To facilitate the extraction procedure to Solana, several helper functions and definitions were implemented. The aforementioned functions are needed because some of the existing functions from the execution model core cannot be used by the extraction due to visibility issues. Moreover, these functions are used as placeholders in contract development, and after extraction, they are replaced by calls to Rust functions that were previously developed. Most of them do not directly affect the chain/environment and as such do not have any lemmas or theorems to prove their effects.

There are two classes of helper functions that were created to facilitate the extraction process: `AccountGetters` (Listing 4.14) and `ChainHelpers` (Listing 4.13). To ensure that both of these classes' functions are visible by the extraction, all of them are declared as global.

The `AccountGetters` consists of a total of seven getter functions that are used to get information from the `AccountInformation` record, one for each field.

```
Class AccountGetters :=
  build_account_getters {
    get_account_address : AccountInformation → Address;
    get_account_is_writable : AccountInformation → bool;
    get_account_balance : AccountInformation → Amount;
    get_account_state : AccountInformation → SerializedValue;
    ...
  }.

Global Opaque get_account_address get_account_is_writable
                get_account_balance get_account_state ...
```

Listing 4.14: Account Getters class and functions in Coq

These getter functions may seem redundant since it is possible to get the fields by using Coq basic record notation. However, using that notation the extraction automatically generates simple getter functions in Rust which will not have the desired behavior. One of the reasons is that the record `AccountInformation` will be converted/remapped into the Rust struct `AccountInfo` and as such the generated getters will not be correctly applied (this will be further discussed in Section 4.3.2). For each of these getters, there is a lemma that proves that given the same account the record getter will return the same as the getter defined in the helper class (for example in Listing 4.15).

```
Lemma get_account_address_correct `{AccountGetters} acc addr1 addr2 :
  account_address acc = addr1 →
  get_account_address acc = addr2 →
  addr1 = addr2 →
  account_address acc = get_account_address acc.
```

Listing 4.15: WrappedActionBody example in Coq

The `ChainHelpers` class consists of a broad range of functions from serialization and deserialization functions to a function to simulate action execution. Function `next_account` is used to access accounts inside of `SliceAccountInformation`, to replace the iterator usually used by Solana contracts. Functions `ser_data` and `deser_data` are used to serialize and deserialize any type of data. Functions `ser_data_account` and `deser_data_account` are similar to the previous two but are meant to be used with accounts, more specifically they are used to serialize data into an account and deserialize data from an account, respectively. Finally, function `exec_act` is used to simulate action execution in a contract. Looking closely at this function signature, it receives a `WrappedActionBody` instead of an `ActionBody`. `WrappedActionBody` is an inductive type like `ActionBody` however instead of each constructor receiving arguments with `Address` type, they receive arguments with `AccountInformation` (Listing 4.16).

```
Definition WrappedActionBody_to_ActionBody (wact: WrappedActionBody) : ActionBody :=
  match wact with
  | wact_transfer from to amount ⇒ act_transfer (account_address to) amount
  | wact_call to msg           ⇒ act_call (account_address to) msg
  | wact_deploy contract       ⇒ act_deploy contract
  | wact_special_call to body ⇒ act_special_call (account_address to) body
  end.
```

Listing 4.16: WrappedActionBody example in Coq

The execution model chain uses addresses for every action. However, most of Solana's actions require access to many account fields. As such, since it is harder to go from an address to an `AccountInformation`, the wrapped actions use `AccountInformation` instead. They can be easily converted into a `ActionBody` by getting the address(es) of the account(s) used in the wrapped constructor.

While developing contracts using these helper functions the respective classes, `ChainHelpers` and `AccountGetters`, need to be put implicitly into context. As a consequence, every function defined in the contract context will have its argument number increased. To use remaps on these functions the number of arguments must be reduced to the bare minimum (the reasons for this will be discussed in the next subsection, Section 4.3.2). In ConCert's contracts, it is necessary to use the class `ChainBase`. In order to prevent this class from increasing every function argument count, a specialization procedure is executed during extraction, resulting in the extracted functions not having `ChainBase` as one of their arguments. In a similar fashion, both helper classes suffer a similar specialization procedure which will remove the class type from the function signature. In fact, the specialization procedure in the extraction is actually a combination of the original `ChainBase` specialization and the added specialization for the `ChainHelpers` and `AccountGetters` classes.

4.3.2 Extraction Approach

The ConCert Framework extraction targets several languages, one of them being Rust for the Concordium blockchain. The extraction procedure is, as we have seen before, composed of transformations, translations, and optimizing passes. ConCert's contract extraction to Concordium builds upon this procedure with a pretty-printer made specifically so that the contracts are allowed on-chain. The new Solana extraction follows the same idea of having a pretty-printer precisely to print contracts that use Solana's crates/libraries.

The approach used can be split into three different parts: generated code from the contract, directly printed code, and remapped code.

A – Generated Code Code that is generated from contracts that are developed using the Execution Model. This code is generated using the Rust extraction present in the ConCert framework, which will convert all the Coq functions used in the contract into Rust methods.

Directly printed code and remapped code are akin to each other. The first type of code consists of functions that return chunks of code in string format, it either can be part of entire functions or a single line of code. The second type of code is tied to existing functions present in the Execution Model (or helper classes).

B – Directly Printed Code The extraction presented in this project has many functions that consist of directly printed code, i.e. program preamble, contract entrypoint and functions that convert actions into instructions.

The program preamble consists mainly of auxiliary functions that together with inductive remapping can be used by the contract. Moreover, this preamble contains the Rust crates imports needed for the proper functioning of the contract.

Solana's Rust program exports an entrypoint that is used by Solana runtime to call and invoke that program. In Solana Rust contract development one sets the entrypoint by using its macro and giving the function name as an argument. The given function (usually called `process_instruction`) must have three specific input arguments: program id, list of accounts, and an array of bytes with the instruction data.

```
Definition process_instruction_wrapper (process_name : kername) :=
  <$ "fn process_instruction<'ai>(";
     "    program_id: &Pubkey,";
     "    accounts: SliceAccountInformation<'ai>,";
     "    instruction_data: &[u8],";
     ") → ProgramResult {";
     "    let prg = Program::new(*program_id);";
     "    let msg : ContractInstruction = ";
```

```
    "        match ContractInstruction::try_from_slice(instruction_data) {";
    "            Ok(m) => m,";
    "            Err(_) => return Err(ProcessError::DeserialMsg.into())";
    "        };";
    "    let cchain =";
    "        " ++RustExtract.ty_const_global_ident_of_kername <%% Chain %%>
                ++"::build_chain(";
    "            PhantomData,";
    "            0, // No chain height";
    "            Clock::get().unwrap().unix_timestamp as u64,";
    "            0 // No finalized height";
    "        );";
    "    prg." ++RustExtract.const_global_ident_of_kername process_name
                ++"(&cchain, accounts, Some(&msg))";
    "}";
    "entrypoint!(process_instruction);" $>.
```

Listing 4.17: Entrypoint for the extracted contracts in Coq

Listing 4.17 shows the function used as the entrypoint for the contracts that are extracted using the new tool. This function is part of the directly printed code, i.e, this function returns a string so that it can be printed into a file. Similar to what is done in ConCert's Rust extraction, we start each interaction with the contract by creating a struct `Program` that contains an arena of memory that can be allocated and accessed. This is done to bridge the gap between the different memory models used by Coq and Rust since the latter is a programming language without a garbage collector. Also, in this context, the `Program` struct contains the address of the contract that is being interacted with to facilitate access to it through execution (Listing 4.18). The entire program/contract is extracted as methods of the `Program` structure, which makes it possible for every method to access the memory arena.

```rust
struct Program {
    __alloc: bumpalo::Bump,
    __program_id: solana_program::pubkey::Pubkey,
}
```

Listing 4.18: `Program` struct produced by extraction

The argument `instruction_data` is deserialized into the enum `ContractInstruction`, using the Borsh library[4]. This library is responsible for the serialization and deserialization of data throughout the contract execution. `ContractInstruction` is implemented in Coq as an inductive type (its values depend on the contract, examples in Section 5.3 and Section 5.4) that when extracted generates a Rust enum containing the same constructors and the same arguments. The generated enum contains all the possible messages that the contract is able to receive and interpret, and contains the necessary data for each call.

To simulate the actions used in the Coq contracts it must be possible to convert each constructor of `ActionBody` into Rust code that has an equivalent behaviour. Listing 4.19 shows the Rust function

[4]https://borsh.io/

that is extracted in order to convert actions and the instruction that are executed for each action. This conversion function however does not allow converting deploy or call actions. Moreover, if these appear in a contract its execution will result in an error due to their behaviour not being implemented in Rust. The constructor `ActionBody::Transfer` as per its Coq counterpart (`wact_transfer`) receives the receiver and sender accounts and the amount to be transferred. Then the amount is subtracted from the sender account (line 7) and added to the balance of the receiver (line 6).

```
1  fn convert_action(&'a self, act: &ActionBody<'a>)
2          -> Result<(), ProgramError> {
3      let cact = if let ActionBody::Transfer
4          (donator_account, receiver_account, amount) = act {
5          if **donator_account.try_borrow_mut_lamports()? >= *amount {
6              **receiver_account.try_borrow_mut_lamports()? += amount;
7              **donator_account.try_borrow_mut_lamports()? -= amount;
8          } else {
9              return Err(ProcessError::Error.into());
10         };
11     } else if let ActionBody::SpecialCall(to, body) = act {
12         return self.convert_special_action(to, body);
13     } else {
14         return Err(ProcessError::ConvertActions.into());
15     };
16     Ok(())
17 }
```

Listing 4.19: Function that converts `ActionBody` into code

The constructor `ActionBody::SpecialCall`, like its Coq counterpart `wact_special_call`, receives the account to call and the body of the special call. Unlike any other action, to convert a special action the function `convert_special_action` is called giving the body as an argument and it works similarly to the `convert_action` function. For example, to check if an account is rent-exempt it is necessary to use the special action `CheckRentExempt`. Once this action is called it will be converted into the code shown in Listing 4.20.

```
if let SpecialCallBody::CheckRentExempt(account_to_check) = body {
    let rent = &Rent::from_account_info(to_account)?;
    if !rent.is_exempt(account_to_check.lamports(),
                       account_to_check.data_len()) {
        return Err(ProcessError::Error.into());
}
```

Listing 4.20: Conversion of the special call `CheckRentExempt`

C – Remmaped Code As mentioned before, remaps are also a relevant part of the extracted code. The execution model uses the type `Address` to identify the contracts. When extracting a contract this type must be remapped to the address type in the target programming language used by the target chain.

Also, in the Execution Model, the contract's `Process` function receives a `SliceAccountInformation` type, which in Coq is represented as a list of accounts. Since Coq does not have a slice type like Rust, this additional type is required to be able to remap it to an actual slice of accounts in Rust. Listing 4.21 presents the remap of these two types.

```
Definition remap_blockchain_consts : list (kername * string) := [
  remap <! ·Address !> "type **name**<'a> = Pubkey;";
  remap <! ·SliceAccountInformation !> "type **name**<'a> =
                           &'a[AccountInfo<'a>];" ].
```

Listing 4.21: `Address` and `SliceAccountInformation` remap to Rust code

Both helpers classes' functions presented in Section 4.2.1, `ChainHelpers`, and `AccountGetters`, must be remapped to their equivalent code in Rust. Listing 4.22 displays some of the remapped `ChainHelper` functions. The `next_account` receives a slice of accounts and an integer to obtain an account from the slice. Solana contracts usually use an iterator to get the accounts from this slice, however, due to lifetime and closures issues this simplified slice access is used. The serialization and deserialization remaps are all similar, using the Borsh library as their basis.

```
// remap <! @next_account !>
fn next_account<'t>(&'t self) -> impl (Fn(SliceAccountInformation<'a>)
    -> &'t dyn Fn(i64) -> Result<&'a AccountInfo<'a>, ProgramError>)
    + 't where 'a: 't, {
        move |slc: SliceAccountInformation| {
            self.alloc(move |index| {
                slc.get(index as usize)
                    .ok_or(ProgramError::AccountBorrowFailed)
            })
        }
}
// remap <! @deser_data_account !>
fn deser_data_account(&'a self) -> impl Fn(()) ->
    &'a dyn Fn(&'a AccountInfo) -> Result<State<'a>, ProgramError> {
        move |_| {
            self.alloc(move |v| match
                            State::try_from_slice(&v.data.borrow()) {
                Ok(val) => Ok(val),
                Err(_) => Err(ProgramError::BorshIoError(String::from(
                    "Deserialization error",
                ))),
            })
        }
}
```

Listing 4.22: `ChainHelper` functions remaps to Rust code

The functions from the class `AccountGetters` are used to get information from an account (struct `AccountInfo`). Some fields of this structure need to be enclosed in other structures in order to ensure its correct function. For instance, the field that contains the program balance has type `Rc<RefCell<&'`

a `mut u64`>>. However, when a function needs to use the contract balance it is expecting an integer and as such, the value obtained from a straightforward getter will not be enough. Thus, making remaps necessary for most of the account getters to destruct and expose the values that are needed for the contract functions. Without account getters, it would not be possible for the contract to, for example, check if it is rent exempt or who the owner of the contract is, making it impossible for a contract to function properly and execute actions. The exception to these getters can be seen in the account's fields that have a boolean type which can be obtained even without using a remapped getter.

In order to use the correct actions in Rust it is necessary to remap the Coq inductive type `WrappedActionBody` into the Rust enum `ActionBody`. This is done using by remapping each Coq constructor to a Rust constructor (Listing 4.23).

```
Definition remap_WrappedActionBody : remapped_inductive :=
  {| re_ind_name := "ActionBody<'a>";
     re_ind_ctors := ["ActionBody::Transfer"; "ActionBody::Call";
                "__Deploy__Is__Not__Supported"; "ActionBody::SpecialCall"];
     re_ind_match := None |}.
```

Listing 4.23: ActionBody remapped to Rust

When extracting a Coq contract, the extraction combines the three parts of the extracted code, function remaps and inductive remaps, directly printed code, and the generated code. This code is then printed onto a file, where after setting it up with `cargo` it can be compiled and deployed onto the Solana blockchain.

5 Evaluation

Contents

In this section, the developed extension is evaluated. First, we discuss the model expressiveness, i.e., what can and cannot be written using the extended framework. Afterward, we present which properties of the execution model have been proved. Then we will present two case studies, `Counter` contract, and `Escrow` contract, where both implementation and extraction will be analyzed. And, in the `Counter` contract we will look at simple properties that have been proven. Finally, in the last section, we will compare the proposed extension and the original framework.

5.1 Model Expressiveness

The Execution Model allows the user to develop a multitude of contracts. These contracts can contain common actions like transferring tokens between accounts, calling other contracts, and creating new accounts (despite some of these actions cannot be extracted). Furthermore, the execution model allows for more Solana specific actions like verifying if an account is rent exempt and transferring ownership of an account with the help of the token program.

The developed model has most of Solana's core concepts identified in Section 4.2.1. Rent is one of the missing concepts since it is usually only used to verify if a contract is rent exempt or not. To do so we use a special call action that does this check. However, this check is most meaningful when the contract is extracted and the code is converted to the actual action of verifying the contract's balance.

Accounts were fully implemented having all fields except its rent related field. It is possible to access all the fields and modify them accordingly. Moreover, there are additional functions to help with the serialization and deserialization of an account's state. In the model an auxiliary type, `SliceAccountInformation`, represents approximately a slice of accounts in Rust. The aforementioned type with the help of a helper function, `next_account`, makes it possible to iterate over the accounts inside the contract methods.

Transactions are implemented and can be used to analyze the chain, despite not being used when writing contracts. Each transaction has exactly one instruction, unlike Solana, as explained in Section 4.2. This differs from the Solana documentation but facilitates reasoning when analyzing a chain.

In this model calling between programs is highly abstracted and program derived accounts are not implemented explicitly. For instance, the special action used to transfer ownership would use program derived accounts if they were explicit (in fact, they are used but only in the extracted program).

In a contract, it is possible to iterate over the list of accounts and check each account's owner and address. For each account, it is also possible to check if it is writable, executable and if it is a signer of that call. Also, it is possible to deserialize and modify an account state and serialize it back to the account. Each contract can have many types of messages, depending on the received message the contract can have different behaviors. All of these features allow for a variety of contracts to be written and reasoned with. For instance, it is possible to implement a `Counter` contract (illustrated in Section 5.3) and an `Escrow` contract (illustrated in Section 5.4).

On the other hand, it is not possible to implement contracts that require making complex calls to other programs and Solana's existing programs. For instance, contracts that require heavy interactions with the Solana ecosystem or with external programs that are not represented in the framework.

5.2 Model Properties Proved

The developed tool can be split into two parts: extraction and execution. As aforementioned, the new extraction builds upon the existing one by adding a new print printer that targets the Solana blockchain. Due to this, there are no new proofs or modified proofs in the extraction process.

On the other hand, with the new Execution Model being an extension of ConCert's Execution model, there is the need to update the existing proofs and, in a few cases, add new ones. However, in a few cases, it was not possible to finish the proofs or add new ones for the new concepts due to time constraints.

A – Helper Classes Proven Properties Since there was a need to have auxiliary functions such as the ones in the `AccountGetters` and `ChainHelpers` classes there are some lemmas and theorems that prove their behavior. For instance, for each account getter, there is a small lemma that ensures that the auxiliary getter returns the same value as using the record projection from the record `AccountInformation` (Listing 5.1). The `ChainHelpers` class, on the other hand, is missing lemmas for every function to ensure their correct behavior when used in contract implementation due to time constraints.

```
Lemma get_account_owner_address_correct `{AccountGetters} acc :
  account_owner_address acc = get_account_owner_address acc.
```
Listing 5.1: Owner address getter correctness Lemma in Coq

B – Core Elements Proven Properties Regarding proofs of core elements of the Execution Model, all of them were kept and if needed were updated/completed. We proved that for any chain trace (i.e. list of chain states) the ending state will not have any actions from undeployed contracts (Listing 5.2). Furthermore, it was proven that undeployed contracts do not have both outgoing and incoming transactions or calls.

```
Lemma undeployed_contract_no_out_queue contract state :
  reachable state →
  address_is_contract contract = true →
  env_contracts state contract = None →
  Forall (fun act ⇒ (act_from act =? contract) = false) (chain_state_queue state).
```
Listing 5.2: Undeployed contracts do not have actions Lemma in Coq

We also proved that given a chain trace and account, the account's balance will be equal to the amounts received from incoming transactions plus the block rewards minus the amount sent in the outgoing transactions (Listing 5.3).

```
Lemma account_balance_trace state (trace : ChainTrace empty_state state) addr :
  env_account_balances state addr =
  sumZ tx_amount (incoming_txs trace addr) +
  sumZ block_reward (created_blocks trace addr) -
  sumZ tx_amount (outgoing_txs trace addr).
```
Listing 5.3: Account balance trace lemma in Coq

We also demonstrated two properties related to contracts states and addresses: if a state is reachable and a contract state is stored on an address then that address must also have some contract deployed to it; and if a state is reachable and a contract state is stored on an address then that address must be a contract address. Furthermore, it was proven that if a state has a contract state on some address then any other state reachable through the first state must also have some contract state on the same address Listing 5.4.

```
Lemma reachable_through_contract_state : forall from to addr cstate,
  reachable_through from to → env_contract_states from addr = Some cstate →
    exists new_cstate, env_contract_states to addr = Some new_cstate.
```
Listing 5.4: Reachable through contract state lemma in Coq

Related to the available contract actions, we also proved the liveness property that there is a future chain state where a transfer action has been evaluated.

That being said, there are many proofs that were not finished in time, i.e., the lemma is defined but the proof was not finished. For instance, the lemma that shows that there is a future chain state where a contract call is evaluated was not proven. Furthermore, some ConCert proofs were not affected by the proposed extension to the framework and, as such, it is not mentioned in this section together with unfinished proofs. That being said, the extension presented in this thesis overall removes some of the formal verification done by the original framework, since a moderate amount of proofs were not finished in time.

5.3 Case Study A—Counter Contract

As an example of our extension, we consider the implementation of a counter contract. This type of contract allows arbitrary users to have a counter which they can increment or decrement. Contracts like this one are standard uses of smart contracts and appear in many blockchains. This particular implementation is an adaptation of the existing one in ConCert's repository.

Before implementing the contract functions it is necessary to add to the context the helper classes (Listing 5.5) to allow the users to use the auxiliary functions previously mentioned.

```
Context {BaseTypes : ChainBase}.
Context {AccountGetters : AccountGetters}.
Context {HelperTypes : ChainHelpers}.
```

Listing 5.5: Contract implementation context in Coq

Each contract must define its state structure and the permitted messages. The `Counter` contract is a simple contract, it has a state with three fields (Listing 5.6) and three possible instructions (Listing 5.7). The state fields contain an integer that holds the counter value, a boolean that tells if the count is active and the address of the owner of the program. The three possible instructions consist of an initializing function, a function to increment the counter, and a function to decrement it.

```
Record State :=
    build_state {
      count : Z ;
      active : bool ;
      owner : Address
}.
```

Listing 5.6: Counter program state

```
Inductive ContractInstruction :=
  | Init (i : Z)
  | Inc (i : Z)
  | Dec (i : Z).
```

Listing 5.7: Counter program instructions

Each program instruction results in a new state (Listing 5.8) that is then returned and serialized into the counter account.

```
Definition counter_init (owner_address : Address) (init_value : Z) : State :=
    {| count := init_value ; active := false ; owner := owner_address |}.

Definition increment (n : Z) (st : State) : State :=
    {| count := st.(count) + n ; active := true ; owner := st.(owner) |}.
```

Listing 5.8: Counter program state init and increment

The contract is defined by an entry point function that receives a `Chain`, a list of accounts, and an instruction. The entry point function checks if there is an instruction, if not it will throw an error, otherwise, it executes the main functionality of the program. This main function iterates over the accounts and obtains the counter program state and, according to it and the received instruction will modify the state accordingly (Listing 5.9).

```
Definition counter (accounts : SliceAccountInformation)
  (inst : ContractInstruction) : result unit ProgramError :=
    let index := 0 in
    do counter_account <- next_account accounts index;
    let index := 1 in
    do counter_owner_account <- next_account accounts index;

    match deser_data_account State counter_account with
    | Ok state => let new_state := match inst with
        | Init i => Some (counter_init
            (get_account_owner_address counter_owner_account) i)
        | Inc i  => if (0 <? i) then Some (increment i state) else None
        | Dec i  => if (0 <? i) then Some (decrement i state) else None
      end in
      match new_state with
        | Some st =>
        do ser_data_account st counter_account;
        Ok tt
      | None => Err InvalidAccountData
      end
    | Err e => Err e
   end.
```

Listing 5.9: Counter contract main functionality

When the contract is fully defined in Coq it is possible to verify simple properties regarding its behaviour. For instance, in Listing 5.10 we verify the correctness of the increment function (seen in Listing 5.8).

```
Lemma counter_increment_correct {prev_state next_state i} :
   increment i prev_state = next_state →
   0 < i →
   prev_state.(count) < next_state.(count)
      ∧ next_state.(count) = prev_state.(count) + i.
```

Listing 5.10: Counter program state init and increment

At this stage, the Counter contract is ready to be extracted. Once the extraction is complete, a Rust program is generated as it was explained in Section 4.3. The records used to represent both state and instructions are enums in the Rust program, as can be seen in Listing 5.11 and Listing 5.12 respectively.

```
pub enum State<'a> {
  build_state(
    PhantomData<&'a ()>,
      i64, bool, Address<'a>)
}
```

Listing 5.11: Rust Counter program state enum

```
pub enum ContractInstruction<'a> {
  Init(PhantomData<&'a ()>, i64),
  Inc(PhantomData<&'a ()>, i64),
  Dec(PhantomData<&'a ()>, i64)
}
```

Listing 5.12: Rust Counter program instruction enum

All the previous contract functions are extracted and have both a curried and an uncurried version. This is done to close the gap between Coq and Rust partial applications disparity [3]. For instance, the curried version of the counter entry point can be seen in Listing 5.13.

```
fn counter_process__curried(&'a self) -> &'a dyn Fn(&'a Chain<'a>)
    -> &'a dyn Fn(SliceAccountInformation<'a>)
    -> &'a dyn Fn(Option<&'a ContractInstruction<'a>>)
    -> Result<(), ProgramError> {
        self.closure(move |chain| { self.closure(move |accounts| {
```

```
                    self.closure(move |inst|
                    self.counter_process(chain, accounts, inst))})})
}
```

Listing 5.13: Counter process curried function

Finally, the whole program comes together and can be accessed and interacted with through the Solana contract entrypoint statically defined in the extraction. The program definition[1] and extracted code[2] can be seen in the repository.

5.4 Case Study B—Escrow Contract

For the second case study, we consider the implementation of an escrow contract. Such a program allows blockchains users to sell an item for an agreed price in a trustless environment. This implementation follows the Escrow contract existing in ConCert's repository which itself follows a Solidity contract example.

This contract works as follows, first, the seller initializes the program and commits two times the price of the item, then the buyer commits two times the price of the item before the deadline. When the two initial steps are finished, the seller (outside of the contract) sends the item over to the buyer which then confirms the item's arrival. Once confirmed the buyer can withdraw from the program the price of the item and the seller can withdraw three times the price. If the buyer does not commit the funds before the deadline, the seller gets his money back from the contract.

As mentioned in the previous case study, most contract functions require some of the helper functions defined in the execution model, as such those classes must be brought into context to allow their functions to be usable. The `Escrow` program state (Listing 5.14) consists of a value that marks the slot in which the contract has been started, a field that stores the next expected action, the seller's and the buyer's addresses, the price of the item being sold and, the amount seller's and buyer's withdrawable amount. To achieve the contract behavior previously described there are four instructions (Listing 5.15): an initialization instruction, a commit funds instruction used by the buyer, an instruction to confirm the arrival of the item (also used by the buyer), and an instruction to withdraw funds from the contract.

```
Record State := build_state {
     last_action: nat; next_step: NextStep; seller: Address; buyer: Address;
     seller_withdrawable: Amount; buyer_withdrawable: Amount; expected_amount: Amount;
}.
```

Listing 5.14: Escrow program state

[1] https://github.com/siimplex/ConCert/blob/master/execution/examples/CounterSolana.v
[2] https://github.com/siimplex/ConCert/blob/master/extraction/examples/extracted-code/solana-extract/counter-extracted/src/lib.rs

```
Inductive ContractInstruction :=
   | init_escrow (buyer: Address) (item_price: Amount)
   | commit_money
   | confirm_item_received
   | withdraw.
```

Listing 5.15: Escrow program instructions

In this program, each program instruction results in a call to a method that returns a success or an error and follows the same structure. For instance, the contract's init function (Listing 5.16) receives the chain, a list of accounts, the buyer address, and the price of the item. When called it iterates over the accounts using a helper function to obtain the seller account and the escrow account that will keep the program's data and it does some sanity and safety checks on them. Then (line 11), there is a special call to check if the escrow account is rent exempt, this call is done using the `exec_act` helper function which receives an action and sends it to another function to convert it into code. Once these checks are done the state is initialized with the necessary information and serialized into the escrow account (line 14). Finally, the seller transfers two times the item price to the escrow account and if nothing went wrong the function returns successfully.

```
1  Definition init (chain : Chain) (accounts : SliceAccountInformation)
2    (buyer : Address) (item_price : Amount) : result unit ProgramError :=
3    let index := 0 in
4    do seller_account <- next_account accounts index;
5    let index := 1 in
6    do escrow_account <- next_account accounts index;
7
8    do if get_account_is_signer seller_account then Ok tt
9          else Err MissingRequiredSignature;
10
11   do (exec_act (wact_special_call escrow_account
12             (check_rent_exempt escrow_account)));
13
14   let init_state :=  (build_state  (current_slot chain)
15       buyer_commit  (get_account_address seller_account)
16       buyer 0 0 item_price) in
17
18   do ser_data_account init_state escrow_account;
19
20   let expected := item_price * 2 in
21   do if (get_account_balance seller_account >? expected) then Ok tt
22             else Err InsufficientFunds;
23
24   do (exec_act (wact_transfer seller_account escrow_account expected));
25
26   Ok tt.
```

Listing 5.16: `Escrow` init function

The `Escrow` program entry point, `escrow_process`, receives a `Chain`, a list of accounts, and an instruction (Listing 5.17). This function matches the instruction and depending on it a different function is executed.

```
Definition escrow_process (chain : Chain)
  (accounts : SliceAccountInformation) (inst : option ContractInstruction)
    : result unit ProgramError :=
  match inst with
    | Some (init_escrow buyer_address item_price) => init chain accounts
             buyer_address item_price
    | Some commit_money => buyer_commits chain accounts
    | Some confirm_item_received => confirm_received chain accounts
    | Some withdraw => withdraw_money chain accounts
    | _ => Err InvalidInstructionData
  end.
```

Listing 5.17: Escrow process function

With the state, instructions and functions defined the `Escrow` contract is ready to be extracted. During extraction, as previously explained, curried and uncurried versions of the contract functions are generated and the records used to define the state and instructions are extracted into enums. Once extracted, the program needs to be manually modified by the user to compile, and when ready it can be deployed onto the Solana blockchain where it can be interacted with. The program definition[3] and extracted code[4] can be seen in the repository.

Both Counter and Escrow contracts once extracted generate extensive programs. These programs are not optimal for the Solana environment and, as such, their performance is not as good as other implementations directly written in Rust.

5.5 Extension Comparison

The tool proposed in this thesis is an extension of the smart contract verification framework ConCert. In this section, we compare our extension to the original framework. More specifically, it is compared the number of contract examples and variety, i.e., what is possible to write. Moreover, the counter and escrow implementation and extraction will be compared, and the proofs accomplished by each model will be compared.

The implemented extension has two contract examples, a `Counter` contract and an `Escrow` contract which represents what is possible to implement using the extended Execution Model. It is, however, possible to implement additional contracts but they are limited to the existing actions. Since many Solana programs require calls to other programs (e.g. system program, token program, token swap program) it would be needed to develop the Execution model further. The ConCert framework, on the

[3]https://github.com/siimplex/ConCert/blob/master/execution/examples/EscrowSolana.v
[4]https://github.com/siimplex/ConCert/blob/master/extraction/examples/extracted-code/solana-extract/escrow-extracted/src/lib.rs

other hand, has a wide variety of contracts in their repository[5] implemented using their execution model. Furthermore, many of these contracts are partly or fully verified, whereas in our extension there is not a fully verified contract.

In Table 5.1 we present a comparison between the two implemented contracts using the extension and the same contracts implemented in the original framework. The contracts prefixed with ”Solana” are the contracts implemented with the extension and extracted to Rust (Solana) the ones without a prefix are implemented using ConCert and extracted to Rust (Concordium). The column ”Coq Lines” contains roughly the amount of lines needed to implement the contract in Coq. The column ”Extraction Time” contains the average time (in seconds) needed to extract each contract to Rust using the corresponding extractions (i.e. for new contracts the new Rust extraction was used). The column ”Rust Extracted Lines” contains the number of lines of the Rust code generated by the extraction procedure.

Contract	Coq Lines	Extraction Time (s)	Rust Extracted Lines
ConCert's Counter	82	4.368	501
Solana Counter	97	7.602	478
ConCert's Escrow	125	41.048	1496
Solana Escrow	195	52.571	1319

Table 5.1: Contract implementation and extraction comparison.

Both `Counter` contracts are similar in the three aspects, the main difference is the number of lines needed to implement the contract and the additional time to extract it. The `Escrow` contracts vary slightly in every aspect. However, this is to be expected since in every contract function there is the need to obtain the accounts and run checks on them. Furthermore, more lines, checks, and functions increase the time it takes to complete the extraction procedure. This shows that our Execution Model introduces a negligible amount of overhead to the contracts and, as expected, the Extraction time increases slightly due to new function remaps and increased contract size in Coq.

Regarding proofs achieved, since our model builds upon ConCert, the proofs in each model are, for the most part, the same. With the exception of the helper classes implemented which proofs were implemented. However, modifying the execution model invalidates some of the existing proofs, and as such those proofs need to be updated accordingly but, as previously mentioned in Section 5.2, it was not possible to complete some of the existing proofs within the time frame of this work. This means that whilst the ConCert model is verified our model is only partly verified since some of the proofs are still to be completed.

[5] `https://github.com/AU-COBRA/ConCert`

6 Conclusion

Contents

Blockchain technology and smart contracts are two quickly advancing research topics that have resulted in many platforms with a wide array of applications. Smart Contracts transactions can have huge amounts of cryptocurrency, and consequently, if a vulnerability or bug is found it could easily lead to a significant financial loss. There are previous works that allow smart contracts to be developed, verified, and extracted for blockchains like Ethereum, Concordium, Tezos, and others but, to the best of our knowledge, there is not any work that does this for Solana programs.

This thesis presents an extension to the existing smart contract verification framework in Coq, ConCert. This extension aims at reducing the overall amount of bugs and vulnerabilities present in Solana smart contracts by allowing users to develop and verify contracts in Coq and then extract them to Rust to later be deployed onto the Solana Blockchain.

This extension is split into two parts, Execution and Extraction. The Execution allows users to develop and implement existing Solana contracts in Coq with basic contract actions (e.g. contract call, transfer, and deployment) and Solana specific actions (e.g. check account rent and transfer ownership). Furthermore, once a contract is implemented, it is possible to verify its behavior and other properties. The Extraction grants the user the ability to implement contracts using the Execution Model and extract them to Rust so they can be deployed onto the Solana Blockchain.

6.1 Limitations and Future Work

This dissertation proposes an extension to the existing smart contract verification framework ConCert to allow for Solana programs to be implemented using the Execution model and extracted using the Extraction. Although the current extension proves to be useful to address the problem, it can be further improved. This section will discuss the existing limitations together with suggestions to solve them, and suggestions for future work.

The first limitation relates to the Execution Model expressiveness. The presented model allows the implementation of a variety of contracts that are not complex and do not require interacting with other deployed or native Solana programs. However, complex contracts, like DeFi protocols, or contracts that interact heavily with the Solana ecosystem cannot be implemented using the current model.

The second limitation relates to the verification of the Execution Model. Due to time constraints, we were unable to complete all the proofs of the present model, and as such, the model is not fully verified. Verifying the entire model, like it has been done in ConCert [2, 9], would increase the reliability and trustworthiness of this extension. Furthermore, completing the proofs in the execution model would improve the trustworthiness of properties verified in smart contracts.

The third limitation relates to the Extraction aspect of the extension. The current extension can extract both contracts used for case studies, however, both extractions result in a Rust program that cannot be immediately compiled. Most of the compilation issues stem from Rust lifetimes, which require updating most of the lifetimes present in the extracted contract functions. Another error preventing compilation originates from Rust borrow checker however, it can be easily solved by following the compiler hints and error descriptions.

As future work is driven by the present work and existing limitations, the next step would be to optimize the extension, i.e. the Execution Model and Extraction. The existing Execution Model is limited in terms of the complexity of contracts it can represent. Further developing the model and optimizing the way Solana specific actions are implemented would increase the model's expressiveness. Fully verifying the model would be the vehement next step.

In the Extraction procedure, the evident next step would be to generate code that can be automatically compiled by the target language which would allow the generated program to be directly deployed on the target blockchain. Thus, preventing possible bugs or vulnerabilities introduced during the generated program debug.

An interesting path for future work would be the implementation of a DeFi protocol (fully or partially), verifying correctness and safety properties, and finally extracting it to Rust and deploying it onto the Solana blockchain. This would prove difficult with the current model and extraction procedure but it would result in a reliable and useful framework that could produce safe contracts.

Finally, even if the Execution Model is fully verified and the contracts implemented using it are fully verified there is no guarantee that these verified properties will hold after the extraction procedure. Thus, as future work, it would be interesting to verify the entire extraction procedure as it would allow users to develop and verify contracts and extract them directly to the blockchain with the guarantee that the deployed contract is bug and vulnerability free.

Bibliography

[1] M. Wander, “Bitcoin blockchain structure,” Jun 2013. [Online]. Available: https://en.wikipedia.org/wiki/Blockchain#/media/File:Bitcoin_Block_Data.svg

[2] D. Annenkov, J. B. Nielsen, and B. Spitters, “Concert: a smart contract certification framework in coq,” *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Jan 2020. [Online]. Available: http://dx.doi.org/10.1145/3372885.3373829

[3] D. Annenkov, M. Milo, J. B. Nielsen, and B. Spitters, “Extracting functional programs from coq, in coq,” 2021.

[4] P. Daian, “Analysis of the dao exploit,” Jan 2016. [Online]. Available: https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[5] S. Palladino, “The parity wallet hack explained,” Jul 2017. [Online]. Available: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/

[6] C. Faife, “Nomad crypto bridge loses $200 million in ’chaotic’ hack,” Aug 2022. [Online]. Available: https://www.theverge.com/2022/8/2/23288785/nomad-bridge-200-million-chaotic-hack-smart-contract-cryptocurrency

[7] X. Li, Z. Shi, Q. Zhang, G. Wang, Y. Guan, and N. Han, “Towards verifying ethereum smart contracts at intermediate language level,” in *International Conference on Formal Engineering Methods*. Springer, 2019, pp. 121–137.

[8] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” *Decentralized Business Review*, p. 21260, 2008.

[9] J. B. Nielsen and B. Spitters, “Smart contract interactions in coq,” in *International Symposium on Formal Methods*. Springer, 2019, pp. 380–391.

[10] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, “A survey of smart contract formal specification and verification,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–38, 2021.

[11] E. Team, “Etherscan: The ethereum block explorer,” *URL: https://etherscan. io*, 2017.

[12] A. Collomb and K. Sok, “Blockchain/distributed ledger technology (dlt): What impact on the financial sector?” *Digiworld Economic Journal*, no. 103, 2016.

[13] S. Haber and W. S. Stornetta, “How to time-stamp a digital document,” in *Conference on the Theory and Application of Cryptography*. Springer, 1990, pp. 437–455.

[14] D. Bradbury, “The problem with bitcoin,” *Computer Fraud & Security*, vol. 2013, no. 11, pp. 5–8, 2013.

[15] N. Szabo, “Smart contracts. 1994,” *Virtual School*, 1994.

[16] ——, “Smart contracts: building blocks for digital markets,” *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, no. 2, 1996.

[17] V. Buterin *et al.*, “Ethereum whitepaper,” 2013. [Online]. Available: https://ethereum.org/en/whitepaper/

[18] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, “An overview on smart contracts: Challenges, advances and platforms,” *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.

[19] C. Sillaber and B. Waltl, “Life cycle of smart contracts in blockchain ecosystems,” *Datenschutz und Datensicherheit-DuD*, vol. 41, no. 8, pp. 497–500, 2017.

[20] A. Yakovenko, “Solana: A new architecture for a high performance blockchain v0. 8.13,” *Whitepaper*, 2018.

[21] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[22] W. Bugden and A. Alahmar, “Rust: The programming language for safety and performance,” *arXiv preprint arXiv:2206.05503*, 2022.

[23] T. R. C. Team, “Announcing rust 1.0: Rust blog,” May 2015. [Online]. Available: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html

[24] C. Paulin-Mohring, “Introduction to the calculus of inductive constructions,” 2015.

[25] ——, “Introduction to the coq proof-assistant for practical software verification,” in *LASER Summer School on Software Engineering*. Springer, 2011, pp. 45–95.

[26] M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter, “The MetaCoq Project,” *Journal of Automated Reasoning*, Feb. 2020. [Online]. Available: https://hal.inria.fr/hal-02167423

[27] A. Anand, S. Boulier, C. Cohen, M. Sozeau, and N. Tabareau, “Towards Certified Meta-Programming with Typed Template-Coq,” in *ITP 2018 - 9th Conference on Interactive Theorem Proving*, ser. LNCS, vol. 10895. Oxford, United Kingdom: Springer, Jul. 2018, pp. 20–39. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01809681

[28] D. Annenkov, M. Milo, J. B. Nielsen, and B. Spitters, “Extracting smart contracts tested and verified in coq,” *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Jan 2021. [Online]. Available: http://dx.doi.org/10.1145/3437992.3439934

[29] D. Annenkov, M. Milo, and B. Spitters, “Code extraction from coq to ml-like languages,” *ML Workshop*, 2021.

[30] D. Annenkov, M. Milo, J. B. Nielsen, and B. Spitters, “Extending metacoq erasure: Extraction to rust and elm,” *The Coq Workshop 2021*, 2021.

[31] P. Letouzey, “A new extraction for coq,” in *International Workshop on Types for Proofs and Programs*. Springer, 2002, pp. 200–219.

[32] ——, “Extraction in coq: An overview,” in *Conference on Computability in Europe*. Springer, 2008, pp. 359–369.

[33] M. Wenzel, L. C. Paulson, and T. Nipkow, “The isabelle framework,” in *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2008, pp. 33–38.

[34] L. Hupel and T. Nipkow, “A verified compiler from isabelle/hol to cakeml,” in *European Symposium on Programming*. Springer, 2018, pp. 999–1026.

[35] M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter, “The metacoq project,” *Journal of Automated Reasoning*, 2020.

[36] A. Šinkarovs and J. Cockx, “Choosing is losing: How to combine the benefits of shallow and deep embeddings through reflection,” *arXiv preprint arXiv:2105.10819*, 2021.

[37] E. Mullen, S. Pernsteiner, J. R. Wilcox, Z. Tatlock, and D. Grossman, “Œuf: minimizing the coq extraction tcb,” in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018, pp. 172–185.

[38] C. Pit-Claudel, P. Wang, B. Delaware, J. Gross, and A. Chlipala, “Extensible extraction of efficient imperative programs with foreign functions, manually managed memory, and proofs,” in *International Joint Conference on Automated Reasoning*. Springer, 2020, pp. 119–137.

[39] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver, “Certicoq: A verified compiler for coq,” in *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

[40] X. Leroy, “Formal certification of a compiler back-end or: programming a compiler with a proof assistant,” in *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006, pp. 42–54.

[41] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[42] A. Bove, P. Dybjer, and U. Norell, “A brief overview of agda–a functional language with dependent types,” in *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2009, pp. 73–78.

[43] A. Arusoaie, “Certifying findel derivatives for blockchain,” *Journal of Logical and Algebraic Methods in Programming*, vol. 121, p. 100665, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352220821000286

[44] B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson, “Mi-cho-coq, a framework for certifying tezos smart contracts,” in *International Symposium on Formal Methods*. Springer, 2019, pp. 368–379.

[45] J. Chapman, R. Kireev, C. Nester, and P. Wadler, “System f in agda, for fun and profit,” in *International Conference on Mathematics of Program Construction*. Springer, 2019, pp. 255–297.

[46] Y. Hirai, “Defining the ethereum virtual machine for interactive theorem provers,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 520–535.

[47] J. Kongmanee, P. Kijsanayothin, and R. Hewett, “Securing smart contracts in blockchain,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 2019, pp. 69–76.

[48] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, “Nusmv: a new symbolic model checker,” *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[49] I. Sergey, A. Kumar, and A. Hobor, “Scilla: a smart contract intermediate-level language,” *arXiv preprint arXiv:1801.00687*, 2018.

[50] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao, “Safer smart contract programming with scilla,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[51] G. A. Pierro and A. Amoordon, “A tool to check the ownership of solana’s smart contracts,” in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 1197–1202.

[52] T. N. Tavu, “Automated verification techniques for solana smart contracts,” Ph.D. dissertation, 2022.

A

Code of Project

```
Inductive SpecialCallBodyEvaluation
          (prev_env : Environment) (act : Action)
          (new_env : Environment) (new_acts : list Action): Type :=
  | eval_transfer_ownership :
    forall (origin from_addr to_addr acc_addr new_owner_addr : Address)
           (wc : WeakContract),
      env_account_owners prev_env acc_addr = Some origin ∨
         env_account_owners prev_env acc_addr = Some from_addr →
      env_contracts prev_env to_addr = Some wc →
      EnvironmentEquiv
        new_env
        (set_account_owner acc_addr new_owner_addr prev_env) →
      new_acts = [] →
      SpecialCallBodyEvaluation prev_env act new_env new_acts
  | eval_check_rent_exempt :
    forall (origin from_addr to_addr : Address)
           (wc : WeakContract),
      env_contracts prev_env to_addr = Some wc →
      EnvironmentEquiv
        new_env
        prev_env →
      new_acts = [] →
      SpecialCallBodyEvaluation prev_env act new_env new_acts
  | eval_check_token_owner : forall (origin from_addr : Address),
      EnvironmentEquiv
        new_env
        prev_env →
      new_acts = [] →
      SpecialCallBodyEvaluation prev_env act new_env new_acts.
```

Listing A.1: `SpecialCallBodyEvaluation` definition.

```
Inductive ActionEvaluation
          (prev_env : Environment) (act : Action)
          (new_env : Environment) (new_acts : list Action) : Type :=
  | eval_transfer :
    forall (origin from_addr to_addr : Address)
           (origin_acc from_acc to_acc : AccountInformation)
           (amount : Amount),
      amount >= 0 →
      amount <= env_account_balances prev_env from_addr →
      account_address origin_acc = origin →
      account_address from_acc = from_addr →
      account_address to_acc = to_addr →
      address_is_contract to_addr = false →
      act = build_act origin from_addr (act_transfer to_addr amount) →
      EnvironmentEquiv
        new_env
        (transfer_balance from_addr to_addr amount prev_env) →
      new_acts = [] →
      ActionEvaluation prev_env act new_env new_acts
  | eval_deploy :
    forall (origin from_addr to_addr : Address)
           (origin_acc from_acc to_acc : AccountInformation)
           (amount : Amount)
           (wc : WeakContract)
           (state : SerializedValue)
           (accounts : SliceAccountInformation)
           (result : unit),
    amount = 0 →
    amount <= env_account_balances prev_env from_addr →
    account_address origin_acc = origin →
    account_address from_acc = from_addr →
    account_address to_acc = to_addr →
    address_is_contract to_addr = true →
    env_contracts prev_env to_addr = None →
    act = build_act origin from_addr (act_deploy wc) →
    wc_process
      wc
      prev_env
      accounts
      None = Ok result →
    EnvironmentEquiv
      new_env
      (set_contract_state to_addr state (add_contract to_addr wc prev_env)) →
    new_acts = [] →
    ActionEvaluation prev_env act new_env new_acts
| eval_call :
    forall (origin from_addr to_addr : Address)
           (origin_acc from_acc to_acc : AccountInformation)
           (amount : Amount)
           (wc : WeakContract)
           (accounts : SliceAccountInformation)
           (msg : option SerializedValue)
           (prev_state : SerializedValue)
           (new_state : SerializedValue)
           (resp_acts : list ActionBody)
           (result : unit),
    amount >= 0 →
    amount <= env_account_balances prev_env from_addr →
    account_address origin_acc = origin →
    account_address from_acc = from_addr →
    account_address to_acc = to_addr →
    env_contracts prev_env to_addr = Some wc →
    env_contract_states prev_env to_addr = Some prev_state →
    act = build_act origin from_addr
                  (match msg with
                   | None ⇒ act_transfer to_addr amount
                   | Some msg ⇒ act_call to_addr msg
                   end) →
    wc_process
      wc
      prev_env
      accounts
      msg = Ok result →
    new_acts = map (build_act origin to_addr) resp_acts →
    EnvironmentEquiv
      new_env
      (set_contract_state to_addr new_state
          (transfer_balance from_addr to_addr amount prev_env)) →
    ActionEvaluation prev_env act new_env new_acts
| eval_special_call :
  forall (origin from_addr to_addr : Address)
         (origin_acc from_acc to_acc : AccountInformation)
         (body : SpecialCallBody)
         (amount : Amount)
         (wc : WeakContract),
    amount = 0 →
    amount <= env_account_balances prev_env from_addr →
    account_address origin_acc = origin →
    account_address from_acc = from_addr →
    account_address to_acc = to_addr →
    env_contracts prev_env to_addr = Some wc →
    act = build_act origin from_addr (act_special_call to_addr body) →
    SpecialCallBodyEvaluation prev_env act new_env new_acts →
    ActionEvaluation prev_env act new_env new_acts.
```

Listing A.2: `ActionEvaluation` definition.