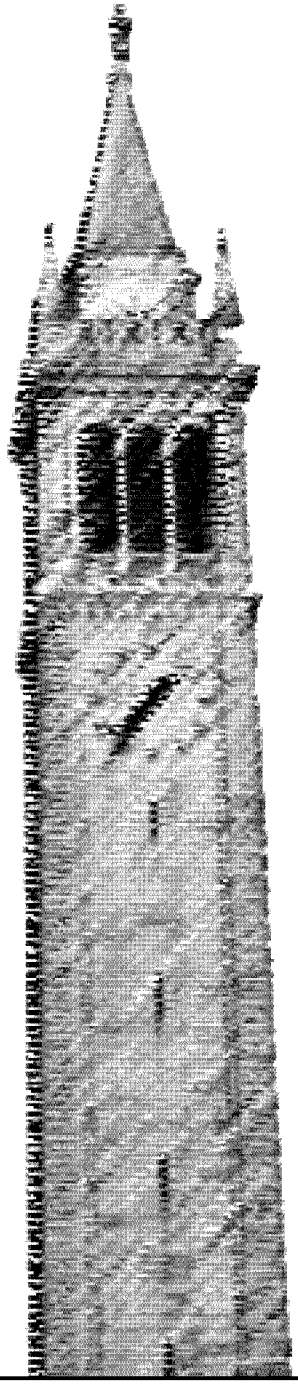
A Languge-Based Approach to Smart Contract Engineering



John Kolb

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-220
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-220.html

December 18, 2020

A Language-Based Approach to Smart Contract Engineering

by

John Kolb

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David E. Culler, Co-chair
Professor Randy H. Katz, Co-chair
Professor Christine Parlour

Fall 2020

A Language-Based Approach to Smart Contract Engineering

Abstract

A Language-Based Approach to Smart Contract Engineering

by

John Kolb

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David E. Culler, Co-chair

Professor Randy H. Katz, Co-chair

Blockchain-based smart contracts have emerged as a popular means of enforcing agreements among a collection of parties without a prior assumption of trust. However, it has proven difficult to write correct contracts that are robust when operating in the adversarial environment of public blockchains. This thesis evaluates the ability of a domain-specific contract programming language to support the expression and systematic testing of practical smart contracts. We present the design, implementation, and evaluation of *Quartz*, a contract language based on the state machine model of execution.

The design and evaluation of *Quartz* is grounded in a suite of case study smart contracts. These are intended to span a wide range of application scenarios and design patterns encountered in practice by contract developers. The language's implementation is organized around the translation of a contract to two targets: a formal specification expressed in TLA$^+$ and an implementation expressed in Solidity. Through its support for model checking contract specifications, *Quartz* enables the discovery of implementation flaws identical to those that have compromised real-world smart contracts. Moreover, its generated Solidity code imposes at most minor execution overhead compared to equivalent handwritten code. Finally, we discuss *Quartz*'s future potential to validate contracts against economic notions of correctness, which are often central concerns in contract design yet are not addressed by current verification techniques.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis, and my journey as a graduate student more broadly, would not have been possible without the help and support of many others. First and foremost, I am very grateful to my advisors, Randy Katz and David Culler. Randy was a steady and supportive presence throughout my PhD process. His guidance and consistent encouragement, especially when I was struggling, were invaluable. I would often leave my meetings with Randy feeling reinvigorated and ready to press on with my work. When Randy was promoted to Vice Chancellor, David graciously and generously agreed to take on a role as my coadvisor. I can't overstate how much I learned from David about what it means to be a scientist and a scholar: forming ideas and questions precisely, developing a proper experimental methodology to answer these questions, and situating one's work within the larger research landscape. David can also convey more about a nuanced technical topic in a single sentence than most people can get across in a paragraph, and seeing this has pushed me to try to be more precise and direct in my own thinking and writing. Finally, both Randy and David have been extremely supportive of my interests in teaching. When I wanted to teach a summer course, they advocated on my behalf to make this happen, and I can't thank them enough for helping me pursue a goal that is very important to me.

Several other faculty members at Berkeley have also supported me along the way. Koushik Sen and Brett Green both served on my qualifying exam committee and gave me valuable feedback and insights as I was in the early stages of my dissertation research. Christine Parlour kindly agreed to serve on my thesis committee on short notice, which I greatly appreciate. Dan Garcia has been extremely supportive of my teaching interests, and I have learned a lot from him about how to lead a class and how to support students in their learning. Zach Pardos collaborated with me on a project on educational analytics and was always supportive as I was learning my way around a new area of research. Finally, I want to thank John Kubiatowicz for helping me get my start at Berkeley and encouraging me as I was still figuring things out in my first year.

I have benefited from the company and assistance of many talented and generous colleagues as a graduate student. In the SDB/BETS research group, I can fondly recall many conversations with Moustafa AbdelBaky, Michael Andersen, Kaifei Chen, Gabe Fierro, Hyung-Sin Kim, Sam Kumar, and Kalyanaraman Shankari. Whether we were talking about technical matters within computer science, navigating graduate school, the craft of research, or current events, I always learned something new or saw a new perspective that helped improve my own thinking. I also enjoyed working with Nitesh Mor, Ben Zhang, Eric Allman, and Ken Lutz in Berkeley's Swarm Lab, where I started in my first year. They immediately welcomed me and included me in a lot of fun discussions around systems design and implementation. Finally, Mike Gardner and I collaborated on a research project applying parallel computing techniques to geological data sets, and I learned a lot from him about both engineering and geology and about how to stay positive as a graduate student and researcher.

Two major groups have supported my dissertation research. First, collaborators on an SRC-funded grant at Intel helped me identify real-world applications for smart contracts and gave me an initial use case to work from. Through my conversations and presentations with Mark Wilkinson, Mani Janakiram, Gustavo Lujan Moreno, and many others, I was able to build my skills in conveying the fundamental ideas behind blockchains and my work. Second, the Haas Blockchain Initiative, with support from Ripple, has also helped to support my work, both through funding and in offering a network on campus with whom I have been able to share ideas. In particular, I'd like to thank Karin Bauer for organizing group events and making sure I always had what I needed from Haas and Ripple.

The staff in Berkeley's EECS department have always been professional, helpful, and friendly on the numerous occasions where I have needed their assistance. Shirley Salanio and Jean Nguyen quickly answered any questions I had about department policies or ensuring that I was satisfying my degree requirements. Cindy Connors, Thomas Silva, and Michael-David Sasson helped immensely with my teaching efforts, whether I needed help with enrollment, moving my lectures to a new room, or accessing old teaching evaluations. Kattt Atchley, Shane Knapp, Jon Kuroda, and Boban Zarkovich helped me with anything I needed in the Rise Lab and always put a lot of work into making our retreats fun and productive. Finally, Albert Goto has not only helped me with all sorts of work-related issues like travel reimbursements and acquiring research equipment, but has also been a tremendous supporter and friend.

Most importantly, I would like to thank my friends and family, who have supported and encouraged me during the entire PhD journey, no matter what difficulties I encountered. I am very thankful that I could always rely on them to listen to me, offer advice, and help me maintain perspective. You all inspire me to keep pushing to be the best person I can be.

Chapter 1

Introduction

Contracts and agreements play a crucial role in our society. These assume a wide variety of forms such as the issue of a loan or a security, an agreement to complete payment for a good or service adhering to certain requirements, or simply the use of a mutually valued currency as a means of exchange. Historically, arrangements like these are backed by human institutions. A network of governments, banks, and other financial institutions facilitate the exchange and track ownership of financial assets. When the parties to a contract are in disagreement, they may pursue a judicial process to render an impartial interpretation of the contract's terms and their application to the situation at hand.

Blockchains have emerged as a new type of computer system that supports the specification and enforcement of the kinds of multi-party agreements described above. Under this vision, human institutions, which can present issues of overhead, corruption, prejudice, or susceptibility to manipulation, are replaced with autonomous computer software. Fundamentally, a blockchain serves as a reliable, tamper-resistant, and globally accessible ledger that is operated by an open network of participants, none of whom can unilaterally modify the ledger's contents or deviate from the agreed upon procedure for adding new ledger entries. Its most natural use is to implement a virtual currency: transfers of currency from one party to another are recorded as ledger entries, meaning anyone may use the ledger's contents to establish their ownership of virtual funds.

The first blockchains were solely concerned with currency, but newer blockchain systems have emerged that support a rich set of applications through the primitive of a *smart contract*. Where traditional contracts articulate a set of terms and contingencies in natural language, smart contracts express these conditions as the logic of a computer program. The program tracks any data relevant to the execution of the agreement and defines a set of operations that end users may invoke to modify this data. A contract's definition is recorded in an entry on the ledger when it is initially created. When a user invokes one of the contract's operations, it is executed exactly as defined and similarly recorded as a ledger entry. Anyone is then free to inspect these ledger entries to determine the contract's current status or to audit its history. Thus, a contract defines the parameters of an agreement while the underlying blockchain secures its proceedings.

Smart contracts have attracted a great deal of interest, but their use in real, mission-critical applications remains challenging. The logic for contract operations is executed exactly as written and cannot be changed once the contract is initialized on the ledger. This can lead to unanticipated scenarios where an unnoticed flaw in the contract's intended logic is exploited to circumvent its intended terms. When this occurs, the parties who suffer losses as a result have little recourse — a blockchain offers no means of reverting prior actions, even when they are clearly in violation of a contract's intended purpose.

In this chapter, we begin by summarizing the concepts of blockchains and smart contracts before discussing the challenges in building correct contracts. We present the central thesis question and then introduce our methodology behind a language and workflows for testing and deploying practical, robust smart contracts in the context of *Quartz*, our prototype language and accompanying software. Finally, we present a road map for the rest of the thesis.

1.1 Blockchains and Smart Contracts

Here, we summarize the main concepts around blockchains and smart contracts. This sets up a discussion of the challenges in writing correct smart contracts, which motivates the research agenda and approach behind this thesis, each presented in subsequent sections.

The Emergence of Blockchains

A blockchain is best viewed as a relatively new element in a long-studied domain of computer science known as distributed computing. In a distributed system, multiple independent computers work together to accomplish a shared goal. These systems have become increasingly important as the demand for computing has exploded, fed by applications like analysis of scientific data and the operation of Internet-hosted services that must accommodate millions or even billions of users and their associated data. By enlisting multiple computers, a distributed system can store more data, perform more computations, and service more end users than a single machine acting alone.

Distributed systems offer larger scale, but they also involve greater complexity. When more computers are integrated into a system, the likelihood increases that one or more of them will experience a failure. In severe cases, the larger system may fail as well, disrupting its users and forcing human operators to manually intervene. As a result, there has been a long line of work in computer science focused on making distributed systems more resilient by developing techniques that allow them to continue operating, without any need for intervention, in the face of component failures.

One particular concern for a distributed system is in achieving consensus, meaning there is agreement about the system's state among its members, and in continuing to reach agreement about changes to system state even as failures occur. Consensus is required to definitively resolve issues such as the responsibilities delegated to each computer in the system, the

contents of data stored on behalf of end users, or even which computers belong to the system. Several different methods have been devised to enable the members in a distributed system to establish consensus, each specifying a sequence of steps to follow whenever a change is made to the system's state. These methods can be characterized by the types of failures they are able to accommodate. The simpler methods can tolerate situations like one or several members losing their ability to communicate with the rest of the system. The most advanced of the traditional methods for achieving consensus tolerate "Byzantine" faults in which one member reports false information to others.

Blockchains are a class of distributed system in which members work together to maintain a global ledger, i.e., an ordered sequence of transactions, each corresponding to an entry in the ledger. They are distinguished by a mechanism for reaching consensus (specifically, an agreement on the order and contents of every ledger entry) that makes two main improvements to prior systems. First, even the traditional systems that can tolerate Byzantine faults assume there is a fixed set of members participating in the system, each uniquely identified and generally given an equal share of influence in reaching consensus. Blockchains, on the other hand, allow anyone in the world to run the software required to participate, and these participants may freely join or leave at will.

Second, where traditional distributed systems require some majority of their participants to faithfully execute the steps necessary to achieve consensus, presumably out of a vested interest in the system's success, blockchains instead assume that each participant acts only in its own interest. Blockchains are engineered to create incentive structures where it is always most profitable to follow the proper process for consensus. This is accomplished by rewarding members for participating in this process with sums of virtual currency, whose possession is tracked in the blockchain's ledger. This currency may also be explicitly transferred from one party to another by recording the transfer as an entry in the ledger. These transfers are accompanied by transaction fees, paid by the initiating party, that are distributed as rewards to the participants in the consensus process, creating the incentive described earlier.

Generalizing Blockchain's Applications with Smart Contracts

The first blockchains focused entirely on implementing virtual currencies, as described above. However, a blockchain's core ability to provide a globally maintained, secure ledger generalizes far beyond this use case. Ledger entries can record the creation of and subsequent modifications to arbitrary collections of data. In this way, blockchains do not act merely as an accounting system for virtual currency but as a general-purpose platform for data storage offering a powerful set of features: everyone in the system agrees on the same history of transactions, no user can tamper with this history once it is established, and anyone with an Internet connection is free to add new entries to this history.

Smart contracts are the mechanism through which programmers use a blockchain as a data storage system in support of their applications. A contract is a piece of software whose code is recorded on the blockchain as a new ledger entry when it is first deployed. This code defines a collection of data items and a set of transformations to this data, each also

recorded as a ledger entry, that may be invoked by any blockchain user. Contracts contain custom-written code and can therefore represent arbitrary data, such as ownership of virtual tokens, the proceedings of a game, ownership of real-world assets, and so on.

It is important to note that smart contracts operate in the same regime as the underlying blockchain's consensus process. That is, anyone with access to the blockchain may attempt to invoke one of the contract's defined transformations. Additionally, there is no expectation of trust among a contract's users who, like the entities working to operate the blockchain, assume that each participant works to maximize his or her own interests. It is the contract's code that specifies the rules of engagement, which are expected to ensure that the contract's data is properly recorded and modified only in the proper ways. When a user invokes a transformation, the blockchain's consensus protocol ensures that the relevant contract code is executed exactly as it is written.

As a result of this situation, correctness of a contract's code is a major concern for its creator and its users. If there is a loophole or vulnerability in the contract's definition, then anyone is free to exploit it by invoking transactions against the contract to be added to the global ledger. As far as the blockchain's consensus mechanism is concerned, this is a routine invocation of contract code, which must be executed to the letter. Additionally, as the ledger is openly available by nature, it is relatively straightforward for malicious users to comb it for vulnerable contract definitions, and smart contract code cannot be modified once it is recorded on the ledger. Although this is by design, to protect users from surreptitious changes, it means that a contract cannot be fixed if it is ill formed. The programmers who write smart contracts must therefore attempt to write error-free code that anticipates any possible series of events.

By far the dominant platform for smart contracts today is Ethereum, a publicly available blockchain run by a global network of participants. Ethereum has served as a valuable proving ground for the smart contract idea, but it has also revealed the difficulty in offering the right set of tools to contract developers. Ethereum-hosted contracts are predominantly written in the Solidity programming language, which was intentionally designed to resemble traditional programming languages. Contracts have a different set of concerns than traditional software, as they are publicly visible and are expected to enforce rules in the face of potentially malicious users, as described above. Moreover, code executes differently on a blockchain than it does in a traditional environment. However, Solidity's design can encourage contract programmers to think in the same terms they do when writing other software, causing them to introduce bugs they will be unable to fix after deployment. This situation, combined with the high stakes when writing a contract as described above, has led to multiple cases in which Ethereum contracts have been exploited by malicious users, allowing the theft or destruction of virtual assets valued at millions of dollars.

1.2 Thesis Question

Blockchains and smart contracts offer a unique set of capabilities, but it requires significant domain knowledge and meticulous software development practices to leverage these capabilities without risk. This creates a barrier to more widespread adoption and use of contracts and naturally motivates work to facilitate contract development. This thesis confronts this challenge by investigating the efficacy of a language-focused approach to smart contract engineering: Can we construct a language that helps authors implement smart contracts and systematically test them for correctness before they are deployed for use?

1.3 Approach and Prototype System

To answer this question, we propose a development methodology for smart contracts. This involves a simplified programming language for defining a contract and the conversion of this definition into a formal specification suitable for verification as well as an implementation suitable for production use. To evaluate our approach, we implement the language and its translators as a complete prototype system, *Quartz*.

Contract Development Workflow

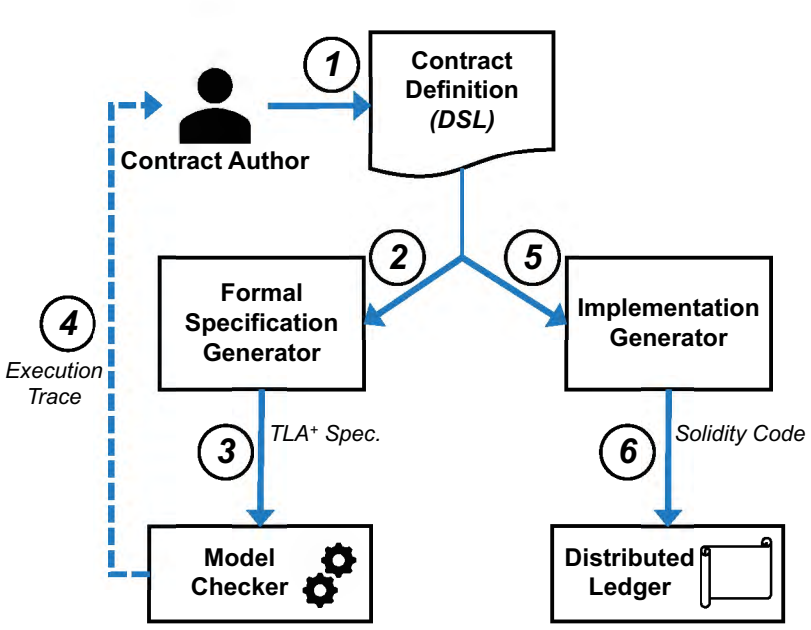


Figure 1.1: *Quartz* Architecture

Our proposed workflow for contract engineering is shown in Figure 1.1. Arguably the most important element of the approach, and the *Quartz* prototype, is its simplified programming language designed specifically for smart contracts. A developer uses this language to formally define her contract as a finite state machine, described in more detail below. Additionally, the developer states the properties she expects her contract to adhere to throughout its lifetime on the blockchain. This initial description is then converted to two target forms. The first is a specification that formally defines the contract's logic and the rules for its execution on the underlying blockchain. This specification is required to apply model checking, a technique in which we can search through possible behaviors of the contract to identify occasions where it violates the properties stipulated by its author. The second target is a complete implementation of the contract suitable for deployment to a blockchain.

Writing Contracts as Finite State Machines

A *Quartz* contract definition adheres to a widely used structure for describing and analyzing computer programs known as a finite state machine. Under this organization, a contract is in one of a fixed number of explicitly listed possible states, including a designated starting state. Each state possesses a set of transitions that may be invoked to move the contract from one state to another. A contract is still associated with a set of data values, as described earlier, and a state transition may involve a set of modifications to these values. Each state transition is effected as a contract transformation — it is invoked by an end user and recorded on the underlying blockchain. However, a *Quartz* contract naturally imposes structure to these transformations, in that only state transitions originating at the contract's current state are permitted.

This ability to restrict when certain operations can be performed is important for contract development. It allows a contract to enforce different modes of operation in which different sets of actions are available to its users. For example, this can be used to make certain data values immutable at different points in the contract's lifecycle. *Quartz*'s language offers two additional means of constraining when state transitions are permitted. First, a contract can require that its data values satisfy certain preconditions before a transition is permitted. Second, a *Quartz* contract can declare which users must approve of a transition before it may proceed. A contract author might stipulate that only a particular user may trigger a transition, one or all members of a group of users must approve of a transition, or any combination thereof. *Quartz* emphasizes these kinds of restrictions in the design of its contract definition language because they often play a crucial role in enforcing the desired terms of real applications. By supporting easy expression of transition restrictions in its language, *Quartz* relieves the contract author of the error-prone task of implementing such restrictions manually, as she would have to do in a language like Solidity.

Analyzing Contracts with Model Checking

State machines not only offer a helpful structure for defining smart contracts, they also facilitate the analysis of their potential behavior once deployed to a blockchain. Under this model, the life of a smart contract is constructed from a sequence of state transitions. *Quartz* uses an approach called model checking to search through possible transition sequences in order to identify situations where a contract fails. More concretely, a user writes down a collection of properties that she expects the contract to adhere to throughout its life on the blockchain. *Quartz* uses model checking to identify any sequence of transitions, starting with the contract's initial creation, that lead to a violation of one or more of these properties. If such a sequence is found, it is presented to the end user. This is more valuable than a simple indication that the contract is flawed, as it indicates precisely how a failure could occur in practice. Model checking is also a valuable approach because it explores all possible sequences of events, even those the author did not imagine and account for when designing her contract, helping to reveal "unknown unknowns" that are often the source of bugs.

While model checking is powerful and flexible, in that it can check whatever properties the user defines, it can be difficult to apply because it requires a formal specification as input. That is, the way a contract executes on the blockchain must be precisely defined before the contract can be analyzed for correctness. Therefore, *Quartz* must generate such a specification from a contract's state machine-based definition. This involves two challenges. The first is translating the structure of the state machine itself from the relatively simple initial description to the more detailed and precise form expected by a model checker. The second is specifying how the state machine executes on the blockchain — where any user may attempt to invoke any transition at any time. Thus, while *Quartz* requires the user to write down the properties to check, she does not have to complete the more difficult task of formally specifying her state machine or the blockchain's inner workings. The *Quartz* prototype system uses the TLC model checker, which is specifically designed to analyze specifications written in the TLA$^+$ language for correctness.

1.4 Thesis Road Map

A description language centered around a state machine abstraction allows developers to express their contracts concisely, to thoroughly test them and surface meaningful flaws, and to benefit from seamless implementation generation without significantly sacrificing execution efficiency.

In Chapter 2, we go into more detail on blockchains, smart contracts, relevant approaches for software testing and verification, and prior work on protocol validation using state machines as the primary abstraction. We then revisit and refine our central thesis question of language support for contract engineering.

Chapter 3 presents a collection of case study smart contract applications drawn from standardization efforts, contracts featured in the relevant literature, and real failure cases

on the Ethereum blockchain. This set of use cases serves two purposes. First, it informs the design of *Quartz*'s contract description language. Second, it serves as the basis for our evaluation, as it offers a set of benchmarks reflecting realistic scenarios that are most likely to be relevant to contract developers.

Chapter 4 presents *Quartz*, a language and associated tools that form a complete implementation of our approach and its focus on state machines and model checking. We describe the language's structures and semantics and offer two illustrative examples: an auction and a multi-signature wallet. We then evaluate the extent to which *Quartz* facilitates the concise expression of contracts by comparing the relative sizes of four different contract representations: a contract written by hand in *Quartz*, an equivalent contract written by hand in Solidity, a Solidity contract generated from *Quartz* code, and a TLA$^+$ specification of the contract generated from *Quartz* code.

Chapter 5 covers formal definitions of the *Quartz* programming language. These precisely describe the execution behavior of a contract state machine and therefore specify the semantics that must be preserved under the two respective translations to a formal specification and to an implementation.

Chapter 6 covers the translation of a contract to a formal specification, particularly *Quartz*'s translation from a state machine-based DSL to TLA$^+$. This includes how *Quartz* data types and primitives are represented in TLA$^+$ and how the blockchain's execution semantics are formally specified. We also present results from model checking contracts drawn from our set of case studies to illustrate how *Quartz* is able to surface flaws that can be exploited to compromise contract integrity.

Chapter 7 discusses the translation of *Quartz* state machines to Solidity, the de facto standard implementation language for smart contracts. This translation path involves similar issues of preserving the original state machine's behavior, but the Solidity translation process is significantly more complex when it comes to expressing and enforcing state transition authorization restrictions. We finish this chapter by quantifying the overhead of using *Quartz*, rather than writing Solidity directly, in terms of the execution efficiency of generated contracts.

Chapter 8 offers an initial exploration of an important area of future work: accounting for user incentives when analyzing potential contract behaviors. In particular, we provide a simple set of definitions and propose an augmentation to the normal model checking process that can help contract developers more precisely determine whether or not certain sequences of events are economically feasible. We finish this chapter with preliminary results to demonstrate the potential value of this approach.

Finally, Chapter 9 concludes the thesis by summarizing its ideas and results, describes recent changes in the blockchain world, and offers potential areas of future work.

Chapter 2

Background

This chapter begins by introducing blockchains and describing their operation. It introduces smart contracts, the main primitive used to construct applications on top of blockchains, and describes some of the difficulties around smart contract implementation. Next, we discuss approaches to software validation, including traditional testing approaches as well as formal proofs of correctness. We then discuss finite state machines as a useful model for robust software design, particularly for protocol development. Finally, we summarize these ideas and revisit our central question of language-centered contract engineering.

2.1 Blockchain as a Distributed Ledger

A blockchain is a peer-to-peer network of software agents participating in a shared protocol. The fundamental capability offered by this protocol is the maintenance of a global *ledger* — an append-only log for storing data. A new item can only be added at the end of the ledger, and no previous items can be modified. Members of the network exchange messages to propose new entries for inclusion in the ledger and to keep their local replicas synchronized with one another. New entries are not added to the ledger immediately. Instead, they are batched together into *blocks* which are periodically appended to the ledger as a unit. On the Bitcoin blockchain, for example, a new block is added roughly every 10 minutes. Thus, the ledger is formed from a sequence of blocks, each containing an ordered list of entries.

Cryptography plays an important role in blockchains. Each ledger entry must be signed by its creator's private key, preventing forgery or future repudiation. Additionally, as shown in Figure 2.1, each block comprising the ledger includes a header with several fields, including its position within the global sequence and a timestamp. A block's header also includes a cryptographic hash of the contents of its predecessor within the sequence. A cryptographic hash of block i is included in block $i + 1$, a hash of the contents of block $i + 1$ (including its hash of block i) is included in block $i + 2$, and so on, meaning the contents of each block are reflected via a chain of hashes found within all following blocks in the sequence. This is what gives the blocks a definitive order and makes their contents immutable — any modification

to a preceding block would break the hash chain. Hence, the sequence forms a *blockchain*, inspired by prior work in cryptography on linked timestamping of documents [10, 45, 46].

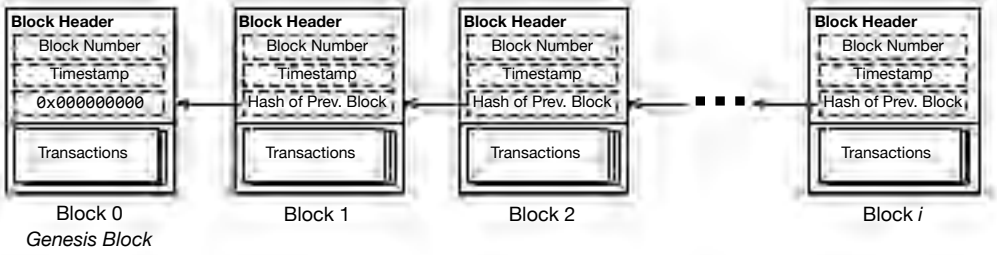


Figure 2.1: A Simple Blockchain

Much of the enthusiasm surrounding blockchains is due to their *decentralized* nature. No single entity runs a blockchain or controls its operation. Instead, correct operation of a blockchain is assured through cryptography and carefully designed incentive structures. Each member of the network is assumed to be self-interested and acting to maximize its own gains. This is in contrast to many other distributed systems, where participants are extended some measure of trust to adhere to the system's protocol. Instead, blockchain users trust the design and security of the system itself, which hinges on its underlying consensus algorithm.

Many blockchain protocols have been devised to maintain consensus among the network's participants in this decentralized regime. All have one fundamental purpose: to reach network-wide agreement on which block to next append to the ledger, enforcing a globally recognized ordering of all ledger entries. Each participant maintains a full and independent copy of the blockchain and stays synchronized with the rest of the network by updating its copy as each new block is agreed upon. We discuss only proof-of-work as a well-known example here; there are many other, more thorough discussions of blockchain consensus for the interested reader [20, 60].

The proof-of-work algorithm was introduced with the launch of the Bitcoin blockchain [70]. Its primary innovation is that it requires network members to commit computational resources to their participation in the protocol. A participant's degree of influence over the blockchain's operation is then tied to the computing power under her control. The blockchain's security rests on the assumption that no single party controls a majority of the computational resources incorporated into its network.

This design gives proof-of-work consensus resilience to *Sybil attacks* — a technique in which a single adversary masquerades as many synthetic users of a system in order to gain control of that system.[1] Many distributed systems restrict participation to a fixed set of authenticated parties to avoid this issue. Using proof of work, however, user identities are

[1] One could imagine using this strategy to hijack a system that updates its state based on a simple majority vote, for example.

made freely available. Members may join and leave the blockchain's network at will. This is possible because an entity that controls multiple user identities must then choose how to apportion their pool of computing resources among those identities. Their influence over the blockchain when acting under multiple identities is no different than if they were acting under a single identity.

Adding Ledger Entries

An end user interacts with a blockchain by submitting a request to add a new entry to the ledger. The process of fulfilling this request is known as a *transaction*. A user does not need to be a member of the blockchain's underlying network, i.e., a participant in its consensus protocol, to initiate a transaction. Instead, she may submit her request to one of these participants, typically via an RPC protocol, who then propagates it to the rest of the network. The transaction is completed once the associated ledger entry is successfully integrated into a new block. Transaction latency is therefore determined by the rate at which new blocks are appended to the chain and is typically on the order of several seconds to several minutes.

In most blockchains, a user must include a *transaction fee* alongside each of her requests to the system. This is used both to incentivize network members to process the transaction and to discourage users from overwhelming the blockchain with spurious requests. When a member proposes a new block that is successfully appended to the chain according to the network's consensus protocol, they are compensated with the fees for all transactions contained in that block.

Blockchain users compete on an open market to have their transactions fulfilled, and network members work to maximize their proceeds by prioritizing transactions with higher associated fees. This has two results. First, a transaction may take longer to complete if its creator pays a smaller fee. Second, the fee required to have a transaction fulfilled within some expected time bound will fluctuate with the current demand on the blockchain.

This naturally raises the question of the currency to use for these system fees. Bitcoin introduced an eponymous virtual currency to serve as the means of exchange for transaction fees. In fact, the Bitcoin blockchain's central purpose is to track the balance of this currency associated with each user identity (public key) in the system. Ledger entries record a transfer of Bitcoins from one party to another, and users submit transactions to the system to effect such a transfer — meaning both the Bitcoins transferred and the Bitcoins used to pay the transactions fee are deducted from her balance. The immutability and integrity of these transfers is enforced through the cryptographic elements of the system's consensus protocol and structure of its ledger, making Bitcoins a *cryptocurrency*. While there has been intense investment and speculation around Bitcoins as a financial asset, their innate value stems from a single source: they are the only currency that can be used as payment for adding a new entry to Bitcoin's blockchain.

2.2 Smart Contracts and Ethereum

Although the concept of a blockchain first emerged in the context of Bitcoin and cryptocurrency, securing the exchange of digital tokens is just one application of a blockchain's ledger. In this setting, ledger entries are interpreted only as records of currency transfers. However, the ledger can be used more generally to record changes to application-specific state. Here, ledger entries record the execution of fully programmable logic. The process of adding a new entry to the ledger becomes a transaction in the more traditional sense of the term — an invocation of an atomic sequence of modifications to shared state, visible to all blockchain users.

Ethereum [19, 98] was the first blockchain to implement this more general scheme. Its organizing abstraction is the *smart contract* [90], an object that is stored and maintained on the blockchain's ledger. Like in traditional object-oriented programming, a smart contract consists of both state (similar to fields) and logic to transform this state (similar to methods). A user creates a smart contract by submitting a transaction to add a special entry to the ledger. This ledger entry contains the contract's code, which specifies all possible transformations to its state, and records the initialization of the contract. This is analogous to executing an object's constructor. Any user may then submit a transaction to invoke one of the contract's pre-defined transformations. The current state of a smart contract is the cumulative result of sequential execution of these transactions, as ordered by Ethereum's ledger. Participants in an Ethereum blockchain maintain two data structures: a record of all prior blocks and the current state of all known contracts on the ledger, represented as a key/value store.

Smart contracts must confront the same set of challenges as the consensus algorithm of the underlying blockchain. They each implement a protocol concerning the creation and maintenance of shared state and are open to interaction with an arbitrary pool of self-interested users. These users do not trust each other, but rather trust the blockchain to effect the contract's transformations exactly as written and as ordered by Ethereum's ledger. Therefore, these users also expect the contract's code to properly implement and enforce the expected protocol in the blockchain's open and incentive-driven environment.

Contract Execution

The logic for smart contract transformations is expressed as bytecode that targets the Ethereum Virtual Machine (EVM), a stack-based runtime designed specifically for distributed execution on blockchains. Each member of the network maintains a local EVM implementation in order to execute the transactions contained in new blocks as they are created and to update the local replica of contract state. The EVM offers access only to information stored in the ledger itself, and its instructions all have deterministic results. This ensures that all participants in a blockchain independently converge to an identical state when they execute the same sequence of transactions. This approach is computationally inefficient, as each transaction is replayed on every node in the network, but it is precisely

what makes the blockchain decentralized. No single authority can stipulate the outcome of a transaction.

Ethereum transactions, like Bitcoin's, are associated with a fee denominated in a cryptocurrency, Ether, and serve to incentivize transaction processing while also creating a cost for using the blockchain. The fee for an Ethereum transaction is a function of the computational costs of the associated contract transformation. The EVM features a metered execution model in which each instruction is associated with a *gas cost*. When a blockchain user initiates a transaction, she specifies an Ether-per-gas multiplier that determines how much she will be charged for the transaction's execution. The same market forces explained above in the context of Bitcoin fees are at play here — practical transaction fees are dictated by the level of demand on the network, and higher fees induce faster service. Additionally, Ethereum enforces an upper limit on the amount of gas consumed by a single transaction. Transactions are aborted if they exceed this limit. This guarantees that the execution of any contract logic terminates, even if this logic contains an infinite loop, and that network members continue to make forward progress when processing new blocks.

Ethereum is therefore much like Bitcoin in that it includes a native cryptocurrency and it tracks balances and exchanges of this currency with its ledger. Ethereum offers user accounts (referred to as *external accounts*) that are bound to a unique public/private key pair and associated with a balance of Ether. However, an Ethereum user account is in fact a smart contract in its simplest form: it includes a single field, a balance of Ether, and a single transformation in which it accepts a deposit of Ether. That is, an external account is a special-case, degenerate smart contract. All other smart contracts implicitly contain these attributes and may extend this model with additional fields or transformations. The sole fundamental difference between the two is that only external accounts may initiate a new transaction on the blockchain. Smart contract code is executed only in the course of processing a transaction requested by an end user, never independently.

A transaction on the Ethereum blockchain may include transfer of Ether from one entity to another, invocation of smart contract logic, or both. The transfer of Ether and the execution of contract code are unified under a single execution mechanism. A request to perform a transaction is represented as a message composed of the following elements:

- A *destination*, specified as a public key
- An optional *value* of Ether
- An optional byte string *payload*
- A *signature* produced using the sender's private key

The EVM adheres to a standard ABI in which the transaction's payload is used to stipulate which contract transformation to execute as well as any arguments to use for the transformation. A contract is free to define a *fallback function* that is executed if the payload is empty or cannot be successfully resolved to any of the contract's declared transformations.

When a blockchain node processes a transaction, it performs the following steps:

1. Verify the transaction's signature.
2. Look up the entity associated with the destination public key. If no such entity is found, abort.
3. Parse the transaction's payload, if present, which indicates which transformation to invoke as well as the values for any parameters expected by the transformation.
4. If the payload is empty or refers to a non-existent transformation, execute the target contract's *fallback function*.
    - For external accounts, the fallback function is the contract's only (implicitly defined) transformation and simply accepts any Ether included with the transaction.
    - Other smart contracts may define their own, customized fallback functions to execute in this situation.
5. Otherwise, execute the transformation with the specified parameters.

One contract may invoke a transformation defined in an external contract in the course of its execution. Such an invocation is known as an *internal transaction*. It still requires the construction of a message with destination, value, and payload elements but is not recorded explicitly on Ethereum's ledger. This situation shares many similarities with a function call in traditional runtimes. The flow of control transfers to the target contract, continues in the environment of the callee (i.e., with access to its fields), and returns to the caller contract. The callee has no means of distinguishing between manual invocation of its code by an end user versus invocation by another smart contract. That is, it cannot discern if it is the root or an internal node of the call graph.

Continuing this analogy, each (external) transaction corresponds to a call graph. By design, the root of this graph always corresponds to manual invocation of a smart contract by an end user, identified by their external account, that is recorded on the ledger. All additional elements in the graph represent internal transactions initiated by smart contract code. Cycles in this graph represent a particularly important case: a reentrant invocation. Here, the assumption of atomicity no longer holds. Say a contract has two transformations f and g and that f is a predecessor of g on the call graph. Then, g will execute against the contract when it is in an intermediate state — it reflects only a partial execution of f.

Programming Smart Contracts

While contracts are ultimately represented and executed as EVM bytecode, most Ethereum contracts are implemented in higher-level, contract-specific programming languages. A number of languages that compile to EVM bytecode have been proposed or developed [33,37,52]. Solidity [36] is the most mature and best-supported language. It has therefore become the de facto standard implementation language for Ethereum contracts. Solidity is a statically-typed and imperative language that strives to offer syntax and features familiar to most

programmers such as conditionals, loops, functions, and exceptions. It is augmented with blockchain-specific keywords and constructs that allow access to information like the number and timestamp of the current block and the public key of the contract or account that invoked the current transaction.

Solidity contracts are structured similarly to classes in object-oriented programming languages. They each include a declaration of typed instance variables that persist for the lifetime of the contract, a constructor to initialize the contract when it is first deployed, and a set of methods that access and modify these fields when invoked. In Solidity terms, submitting a transaction to the blockchain means invoking a particular method of a smart contract. Solidity code may freely call functions defined in external contracts, creating the internal transactions described above. Thus, while Solidity has the advantage of closely resembling the languages that most programmers are accustomed to from traditional domains, we will also see that it obscures the inner workings of the blockchain and certain subtleties in its execution semantics.

Example: Blockchain-Hosted Auction

Imagine that we wish to use Ethereum to record and secure the proceedings of an auction. Running an auction in this fashion has a number of benefits. The rules of the auction are declared at the outset on the ledger and cannot be manipulated afterwards. For example, the seller cannot extend the deadline in an effort to solicit higher bids, nor can bids be ignored to exclude certain parties from participating in the auction. Because each bid corresponds to an entry in the ledger, the full proceedings of the auction are preserved on the blockchain and therefore open for inspection.

If we wanted to replicate the proceedings of a traditional auction, we would simply add fields to the contract to track the highest bid seen thus far, the party behind this bid, and the bidding deadline. A Solidity function allows any party to declare a new bid, provided it exceeds the current winning bid. At this point, we have an auction recorded on the blockchain in which participants must expend Ether to cover the transaction fees required to submit bids. The winner is determined by the contract's logic and recorded on the ledger. As in a traditional auction, this party settles with the item's seller afterwards, independent of the bidding process.

We can extend this design to implement a blockchain-native auction, which handles both bidding and compensation of the seller. Now, payment from the winning bidder to the seller is run through the blockchain itself. In principal, this payment may be denominated in any virtual token that is tracked on Ethereum's ledger. For simplicity, we consider the case where the same currency, Ether, is used to pay transaction fees and to compensate the auction's seller. To prevent a participant from later reneging on her bid, we force her to commit to possible future payment by depositing the sum of her bid to be held by the auction contract in escrow. If her bid is later surpassed, she is free to recover her deposit.

Figure 2.2 shows an excerpt of Solidity source code to implement part of the auction contract described above. Its fields `highestBid` and `highestBidder` respectively keep track

of the highest bid seen thus far and its sender. The `bid` function specifies the procedure that is executed upon submission of a new bid. Line 6 demonstrates two important Solidity features. The `require` keyword throws an exception if a certain condition is not satisfied, in this case rejecting the transaction if the new bid does not exceed the highest bid seen thus far in the auction. `msg.value` expresses the balance of Ether furnished by the bidder (represented by `msg.sender`) and associated with the transaction. On line 8, Solidity's `transfer` keyword is used to deduct Ether from the acution's balance and credit it to the former highest bidder. While the transfer operation appears simple, it involves a potential interaction with untrusted code defined by the recipient, a security concern that we will discuss next. Finally, the code updates the contract's internal state to reflect receipt of the new bid.

```
1  contract Auction {
2    address payable public highestBidder;
3    uint256 public highestBid;
4    ⋮
5    function bid() public payable {
6      require(msg.value > highestBid);
7      if (highestBid > 0) {
8        highestBidder.transfer(highestBid);
9      }
10     highestBid = msg.value;
11     highestBidder = msg.sender;
12   }
13   ⋮
14 }
```

Figure 2.2: Part of an Auction Contract Written in Solidity

Contract Vulnerabilities

Solidity's syntax and basic execution semantics are, by design, similar to those of traditional imperative programming languages, but writing a correct and secure smart contract can be challenging. Ethereum's contract execution model introduces subtleties in Solidity's behavior, many of which have no analogues in other domains. Contract developers rely on their prior experiences and intuitions regarding the execution of imperative code, and Solidity's efforts to present familiar syntax can obscure the underlying blockchain's true execution semantics. Writing a correct and secure smart contract is particularly important given that its

bytecode representation is immutable once added to the ledger. A contract author cannot patch the implementation to address bugs that are found after deployment.

We will focus this discussion on two Solidity programming pitfalls that stem from interactions between contracts and serve as illustrative examples. Broader discussions of Ethereum contract security are readily available in the literature [7, 65, 84]. As explained above, a transaction targeting a Solidity contract corresponds to invocation of one of its functions. While a contract author may assume a Solidity function without any explicit calls to external contract functions executes cleanly from start to finish (i.e., is atomic), transfers of control to external code can occur in subtle ways that are difficult to restrict and can compromise the contract's operation. As others have pointed out [50,84], Solidity contracts are therefore best viewed as objects with mutable state that must be carefully safeguarded against issues of concurrent execution and interaction with malicious code.

As an example, consider line 8 of the auction contract in Figure 2.2, which uses Solidity's `transfer` keyword to transfer Ether to the address (public key) designated by the contract's `highestBidder` field. Recall that a transfer of Ether from one address to another is carried out as its own (in this case, internal) transaction. A message is emitted with `highestBidder` as its destination, `highestBid` as its value, and an empty payload. This means the destination's fallback function is executed in response. If the destination is an external user account, it simply accepts the deposit of Ether, and control returns to the auction contract. However, if the destination is a smart contract, the fallback function may contain custom logic. By attempting to repay the newly supplanted bidder, the auction has invokved arbitrary external code outside of its control.

A malicious callee contract can then derail execution of the parent transaction by inducing a failure, such as throwing an exception, which reverts any changes to contract state made thus far. In this way, the target of a `transfer` operation can block the sender from making any forward progress. For example, in the auction contract of Figure 2.2, a malicious bidder can trigger an exception on line 8, preventing any progress past this point in the code and effectively seizing control of the auction by preventing any new bidder from replacing her as the acknowledged highest bidder, an update that would normally occur on lines 10 and 11.

The Solidity development community has established an idiom to address this in which funds are only returned through a specific withdrawal function [34]. A contract user invokes this function to retrieve their funds, e.g., if they are no longer the winner of an auction. Now, if the payee throws an exception to disrupt this transaction, they are only working against their own self-interest as they will be unable to recover their Ether held in escrow.

However, there is a downside to using `transfer` to exchange Ether. It delegates only a small amount of gas to the destination contract, preventing the recipient from doing any useful work in its fallback function. One could easily imagine situations where a contract may want to take some response to a deposit of Ether. In an auction, for example, a bidder may run their participation through a smart contract that automatically submits a new bid when its previous bid is exceeded, implementing an automated bidding strategy. Solidity offers an alternative `call` primitive that transfers Ether and allows the callee to freely consume gas (up to the blockchain's per-transaction limit).

An excerpt from an improved Solidity implementation of a blockchain-native auction is shown in Figure 2.3. It features two changes to the previous version. First, the `bid` function no longer refunds the previous winning bidder. Instead, the contract records the fact that it owes Ether to this entity in a new contract field. A separate `withdraw` function is invoked by participants in the auction to reclaim their deposits.

```
1  contract Auction {
2    address public highestBidder;
3    uint256 public highestBid;
4    mapping(address => uint256) pendingWithdrawals;
5    ⋮
6    function bid() public payable {
7      require(msg.value > highestBid);
8      if (highestBid > 0) {
9        pendingWithdrawals[highestBidder] += highestBid;
10     }
11     highestBid = msg.value;
12     highestBidder = msg.sender;
13   }
14
15   function withdraw() public {
16     if (pendingWithdrawals[msg.sender] > 0) {
17       msg.sender.call.value(pendingWithdrawals[msg.sender])();
18       pendingWithdrawals[msg.sender] = 0;
19     }
20   }
21   ⋮
22 }
```

Figure 2.3: Part of an Improved Auction Contract

The improved contract of Figure 2.3 is still not secure. When a Solidity contract invokes external code, control may eventually return to that contract in a reentrant function call, as described above. The second function's code then executes while the contract is in an intermediate state reflecting partial completion of the previously invoked function. In the case of our auction implementation, a malicious contract is able to steal funds from the auction through the following steps:

1. The attacker submits a legitimate bid to the auction contract.

2. When the attacker's bid is supplanted by a new and higher bid, the auction contract records its debt to the attacker in the `pendingWithdrawals` field.

3. The attacker sends a transaction invoking the auction contract's `withdraw` function. Because of the attacker's previous bid, the conditional check on line 16 of Figure 2.3 passes.

4. The auction contract initiates a `call` targeting the attacker's contract, invoking its fallback function.

5. The attack contract is credited with new Ether from the `call`. It then uses its fallback function to invoke, and reenter, the auction contract's `withdraw` function.

6. In `withdraw`, `pendingWithdrawals` does not reflect the completion of the `call` from step 4. The conditional on line 16 passes again, and another `call` to the attacker occurs.

7. The attacker repeats this process, receiving more Ether each time a `call` occurs, and continuing the attack until the balance of the auction contract is exhausted.

There are two ways to prevent this attack. One is to again use `transfer` to send funds. This prevents the recipient contract from executing malicious code in its fallback function, but also means the fallback function cannot do any useful work in response to the deposit. In a second approach, widely used among contract developers [34], the `withdraw` function updates the user's balance in the `pendingWithdrawals` field *before* initiating the `call`, as shown in Figure 2.4. While this fix is straightforward to implement, it is not immediately obvious, particularly to an inexperienced Solidity developer.

```
1 function withdraw() public {
2   uint256 amount = pendingWithdrawals[msg.sender];
3   if (amount > 0) {
4     pendingWithdrawals[msg.sender] = 0;
5     msg.sender.call.value(amount)(): // Can also use transfer
6   }
7 }
```

Figure 2.4: A Safe Token Refund Implementation

This reentrancy bug is essentially the same as the vulnerability behind the now-infamous attack on `TheDAO`, a smart contract that accumulated approximately $150 million worth of Ether. The attacker was able to seize Ether valued at the time at nearly $60 million by exploiting reentrancy [26], although this caused the market value of Ether, and thus the value of the attacker's stolen tokens, to plummet. In response, a majority of the Ethereum blockchain's participants agreed to a controversial update to rewrite transaction history to revert the attack [51].

2.3 Software Testing and Verification

Given the mission critical nature of smart contracts and the subtleties in their execution described above, there have been many efforts to develop tools to systematically test or analyze smart contracts in order to lend confidence to their correctness before they are deployed in production. These tools draw on a rich history of work in software testing and verification. While they share common goals, the approaches described below vary widely in the expected degree of user (contract author) intervention and in the amount and form of information the user must furnish upfront before the relevant technique can be applied.

Explicit Testing

Handwritten unit and integration tests are a cornerstone of modern software engineering and are strongly encouraged in the smart contract development community. A number of contract development environments and frameworks offer tools to facilitate such testing [25, 30, 75]. Typically, these frameworks allow a developer to provision a local mock blockchain as a test environment, deploy contracts to this test environment, run a reproducible sequence of transactions against these contracts, and validate the results at each point in this sequence. For example, Truffle [95] — arguably the most popular contract development platform — allows the developer to script the deployment of a set of contracts to the local environment and to define unit tests that exercise these contracts. Truffle test suites are written in Solidity, JavaScript, or a combination of the two.

Explicit testing does, however, have its limitations. These are no different for smart contracts than for traditional software. First, a developer must manually write each test case. This means she must write a sizeable body of code that is often larger than the contract's actual implementation to thoroughly exercise the various aspects of a smart contract's behavior. This is a particular challenge when the contract author wishes to test behavior when a contract interacts with potentially malicious external contracts, as in the attack scenarios described above. Secondly, handwritten tests cannot identify the "unknown unknowns" in potential contract behaviors, i.e., contract failures that occur in scenarios that the developer does not imagine and therefore does not guard against in her implementation.

Model Checking

Model checking is a software validation technique that addresses some of the issues in explicit testing identified above, although it features weaknesses of its own. We only summarize the approach and its relevance to smart contracts here; interested readers may wish to refer to surveys in the literature for more information [1, 11]. Model checking is fundamentally based on search. A model checker searches all possible execution traces of a program. The user submits properties about her program for the model checker to validate. These typically fall into two categories: *safety* properties that are expected to hold at all times and for all possible execution traces and *liveness* properties that are expected to be satisfied at some

point in some or all possible execution traces. A model checker explores these traces, checks if user-specified properties are satisfied at each point, and returns an error if it finds a violation, e.g., a safety property that is not satisfied. A model checker is able to not only inform the user of the violation, but also to provide her with a concrete counterexample: the execution trace in which the violation occurs.

Model checkers are not particularly sophisticated in the search algorithms they use. Most rely on a traditional breadth-first or depth-first traversal of the search space of possible program execution traces. However, such a search space can be extremely large even for a seemingly simple computer program. For example, if the search arrives at a point in the program where a 32-bit integer variable is assigned a user-specified value, there are roughly four billion possibilities to explore. Sequences of operations with multiple possible outcomes further compound this expansion factor. This "state space explosion" problem is arguably the main challenge in the practical application of model checking. A naive attempt at model checking a program may take many hours to terminate or may exhaust the memory resources of the host machine as the data structures representing the space explored thus far and the search frontier become intractably large.

There are two main ways in which model checkers remedy this. First, they may offer the user the ability to impose bounds on the search space that is explored. Often, these bounds are restrictions on the domain of possible values for certain data types. For example, integers may be allowed to assume values from -100 to 100. The tightness of these bounds represents a tradeoff: a search of a more restricted domain will require less time and less memory, but it may not find an important failure case that would be revealed with a wider search. Users often follow a workflow in which they first model check their program using a tightly bounded domain to ensure fast completion and quick identification of any problems, and then repeat the process, widening the search space as long as no problems are found. While bounded model checking is widely used and practical, it is important to note that it does not carry the same weight as a formal proof of correctness. When this approach fails to identify any violations of user-specified properties, it is not an absolute guarantee that these properties hold. Instead, it is only an indication that they hold for all program states explored during the model checker's search.

Second, a model checker may try to use a coarser but more tractable representation of the search space. *Explicit state* model checkers distinguish between every possible state of a program, i.e., all possible combinations of values assigned to the variables currently in scope. This is the most precise but least efficient representation. Alternatively, model checkers may use *predicate abstraction*. A set of predicates on program state, such as an inequality on an integer variable, are used to organize the search space. Each node in the search graph now represents an equivalence class of program states that all satisfy the same combination of predicates rather than an individual program state. While this reduces the size of the search space, it introduces a number of challenges. It is not obvious how to select predicates to use alongside the user-provided properties, and once a violation is found, it is no longer straightforward to construct an execution trace that induces the violation as a concrete counterexample.

The main advantage of model checking over traditional testing is that it does not require the user to identify and specify the potential failure scenarios. Instead, the model checker explores scenarios automatically, meaning it is capable of finding failure cases that a developer would not have imagined and tested for herself. This has proven particularly valuable when designing and validating concurrent software in which the actions of multiple interacting processes or threads may be interleaved arbitrarily by the underlying execution environment.

However, the use of a model checker can still be a labor-intensive process. This is because any model checker requires a representation of the program to validate that it can reason about mechanically and formally. Model checkers that directly accept an implementation of the program often impose limitations or can only reason about a restricted class of properties. For example, the Blast [48] model checker will accept a C program as long as it uses only an approved subset of the language's features. Seahorn [44] can reason about LLVM bytecode, but only in terms of low-level properties such as memory safety. More flexible model checkers such as Spin [53] and TLA$^+$ [63] instead require a formal specification of the program's structure and behavior that is separate from its implementation. These specifications are written in specialized languages that tend to focus more on mathematical precision and convenience rather than similarity to the features of common implementation languages. Thus, a software developer may have to learn a new language and translate their existing implementation to this language in order to make effective use of a fully-featured model checker.

2.4 Protocol Design and Validation

One domain in which software design and validation has been particularly well developed is the engineering of protocols to govern interactions between multiple computational agents, such as CPUs in a multiprocessor system or nodes in a distributed, networked system. Such a protocol establishes the structure and rules for communication among these agents and allows each agent to form conclusions regarding the collective state of the system based on the communications it has exchanged thus far. Protocols also allow users and designers to reason precisely about a system's behavior, for example to establish safety and liveness properties.

These protocols can be challenging to design. They must be exhaustive in the sense that they address all potential failure cases and all possible sequences of events. A well-designed protocol never induces a situation in which the current state of the system isn't covered by the protocol's rules. As a result, there is significant motivation both to construct clear protocol representations to make them easier to reason about and to systematically vet protocols before they are used in production. Here, we can see many similarities to smart contracts on a blockchain, which one can view as a type of protocol definition where communication corresponds to invocation of contract transformations and the system's state is the values of the contract's fields.

State Machines

State machines have emerged as a common way to represent, specify, and analyze protocols in distributed systems including cornerstones of modern computing such as TCP [54] and two-phase commit. A state machine is a model of computation in which the system is in one of a fixed number of possible *states*. The system undergoes a *transition* to move from one state to another. All possible transitions are explicitly enumerated as part of a state machine's definition, each specifying a start and end state. Each transition may also include a *guard*: a predicate on the machine's current status and any information about the event that triggered the transition. A transition is *enabled* when the machine is in its specified starting state and the transition's guard, if present, is satisfied. The status of the machine may simply be its current state, or it may include the values of auxiliary *fields* that persist for the lifetime of the machine and are modified during the execution of transitions. Machines that include such fields are called *extended finite state machines*.

In the case of a protocol, transitions are triggered by the receipt of a message. The message recipient will then execute a state transition based on the type and contents of that message. This formulation simplifies both the representation and analysis of distributed protocols. Rather than reading a body of code with message parsing and branching logic to determine the appropriate response to each potential message, engineers can instead use a state machine, possibly represented pictorially, to record and communicate a protocol's design. Reasoning about the behavior of the protocol becomes a matter of enumerating possible transition sequences and tracing the state machine's actions taken in response. Model checking is therefore a natural technique to apply in this setting, as it provides an automated means of enumerating these transition sequences and thus validating a protocol.

2.5 Summary

A blockchain represents a new kind of distributed system that emphasizes open access and participation as well as collective governance and operation rather than control by a central institution or set of known entities. Smart contracts are the standard abstraction through which users leverage a blockchain's capabilities to support some larger application. They allow a developer to define a collection of data and a set of transformations on this data. Both this initial definition and any transformations are recorded and secured by the blockchain and its network of participants.

The features that characterize blockchains — openness, decentralization, immutability, and rigid adherence to protocol (in terms of both consensus and contract execution) — lend smart contracts both strengths and weaknesses. Anyone is free to interact with a contract, no one may exert unilateral control over the contract's operation, contract history cannot be repudiated, and a contract cannot be manipulated outside of the means offered by its definition. Conversely, anyone may inspect a contract for flaws in its logic, no one (including contract authors) may play a supervisory role and step in if a contract's proceedings go

awry, a contract definition cannot be fixed after deployment, and even flawed contract code is respected as the last word in a contract's operation. Additionally, we have seen how subtleties in contract execution semantics can lead to unanticipated behavior.

2.6 Revisiting the Thesis Question

With this background in place, we refine our formulation of the thesis question originally posed in Chapter 1. Our approach leverages state machines as the organizing abstraction for programming smart contracts. Additionally, we are interested in identifying contract flaws that may stem not only from incorrect logic but also from subtleties in blockchain-based execution such as the exception handling and reentrancy cases described above. Finally, as Ethereum-based blockchains feature an innate notion of execution cost, quantified as gas, we are also interested in the gas overhead incurred by writing and reasoning about contracts as state machines rather than in terms of native code. A more precise formulation of the thesis question is: Can a programming model based on state machines enable the expression of practical smart contracts, surface flaws related to both contract logic and subtleties in execution on the blockchain, and support the generation of working implementations that feature minimal overhead in terms of execution gas costs?

The remainder of the thesis presents our approach and progress in addressing this question. This begins in the next chapter, where we present a collection of case study applications, which play two important roles in this work. First, they represent a survey of contract design patterns and the features that contract developers commonly seek to implement, which informs the design of our contract description language. Second, they also serve as a set of benchmarks against which we can evaluate our solution in terms of expressiveness, conciseness, and overhead.

Chapter 3

Case Studies

The task of implementing a language-focused workflow for contract engineering immediately raises two questions. First, what features and primitives should be emphasized in the design of a contract description language? Second, under which conditions and situations do we evaluate this workflow? Answering the first question requires identifying a set of recurring design patterns and challenges in the smart contract domain. We approach this task by gleaning a set of recurring design patterns and themes from a set of well-chosen case study contracts. Then, we can reuse these case studies as evaluation benchmarks.

In this chapter, we discuss our methodology for choosing these case study smart contracts. We then present the set of case studies we have assembled. For each, we summarize the problem and some of the challenges in creating a smart contract to address it. Next, we extract and present a set of themes that frequently appear in smart contract design and show the themes raised by each case study contract. These themes then inform the design of the *Quartz* contract description language.

The evaluations presented in subsequent chapters are all grounded in this body of case study contracts. Chapter 4 offers statistics on the relative sizes of the different representations (*Quartz* code, Solidity code, and TLA$^+$ specification) of each case study. Chapter 6 discusses experiences in validating contracts for two sample applications. Finally, the case studies presented in this chapter are used as a benchmark suite for the measurement of execution overhead of *Quartz*-generated contracts against handwritten Solidity equivalents in Chapter 7.

3.1 Selecting Case Studies

The main goal in selecting a body of case study contracts is that it serves as a properly representative sample. The selected contracts must span a sufficiently diverse array of application domains, and each application should be something that the smart contract community is likely to care about, ideally stemming from a proven deployment. We have therefore drawn all case studies from external sources, rather than positing our own. These sources are:

- *Contract programming documentation and tutorials*: These present example contracts to showcase the most fundamental or interesting features of various contract programming languages, or they highlight pitfalls where these languages have subtleties.
- *Related papers on contract verification*: Like *Quartz*, other smart contract research efforts justify and evaluate their work through benchmarks. We measure *Quartz* against some of the same benchmarks to serve as a basis for comparison.
- *Contracts involved in high-profile security breaches*: There have been occasions where even experienced contract developers failed to identify security holes. We need to demonstrate that *Quartz* can flag these kinds of vulnerabilities through its validation process.
- *Contract standardization efforts*: Ethereum, in particular, has been working to standardize contracts for certain applications its community has deemed important, such as virtual tokens and financial instruments, through its ERC process. We have selected a subset of these standards to include as case studies.

Case Study List

Table 3.1 lists the case studies we chose for further study, with a brief description and, when available, references to the original source for each. We also offer a brief summary of each case study contract below. Later, in table 3.2, we list the key design patterns encountered in each use case as evidence that this sample of case studies is sufficiently broad.

Auction Auctions, more precisely English auctions with open participation, serve as a very common example use case in tutorials and introductions to smart contract programming [38, 60]. A blockchain-adjudicated auction runs differently than a traditional auction. Each participant who submits a bid to the auction's backing smart contract must provide an immediate deposit covering their bid denominated in a virtual currency. This currency may or may not be the same currency used to pay transaction fees (in Ethereum, gas costs). For our purposes and without loss of generality, we consider the case where the same virtual currency (Ether) plays both roles. If this participant is later supplanted as the winner, she must be able to recover her deposit, either receiving a refund from the contract or proactively initiating a recovery transaction against the smart contract.

Auction contracts often include a deadline that is declared at the outset and enforced by its code, leaving it immune to manipulation by the auction's selling party or any participants. Bids are only accepted before this deadline, and the auction's winner is determined by the contract once the deadline has passed. At this point, the seller is free to claim the auction's proceeds.

Name	Description
`Auction` [38]	Simple auction with open participation and open bids
`Crowdfunding` [79, 87]	Crowdfunding campaign with deadline and token refund logic
`Logistics`	Shipment tracking contract
`SimpleMultiSig`	Two-participant multi-signature wallet
`StaticMultiSig` [77, 78]	Multi-signature wallet with fixed set of signers
`DynamicMultiSig` [77, 78]	Multi-signature wallet with dynamic set of signers
`ERC-20` [77, 97]	Ethereum standard token implementation
`ERC-721` [31, 76, 77]	Ethereum standard non-fungible token implementation
`ERC-1202-Simple` [99]	Ethereum standard voting implementation
`ERC-1202-Weighted` [99]	Voting implementation with voter-specific weights
`ERC-1540` [47, 57]	Ethereum standard asset management implementation
`ERC-1630` [16, 49]	Ethereum standard time-based fund distribution implementation
`ERC-1850` [15]	Ethereum standard token loan implementation
`ERC-780` [94]	Ethereum standard metadata registry
`RockPaperScissors` [27]	Simple rock-paper-scissors game implementation
`DAO` [88]	Decentralized autonomous organization

Table 3.1: Contract Case Studies

Crowdfunding Crowdfunding campaigns have served as another popular example contract application, particularly as a benchmark in related academic work on alternative contract programming languages [87] and verification tools [79]. A crowdfunding contract is initialized by the campaign's beneficiary, who wishes to solicit funding from the general public (or at least the users of a particular blockchain) for the development of some specific product or service that they will later release. The beneficiary sets a target amount for total funding and a deadline that marks the end of the campaign. If the target is met by the deadline, enforced by the contract, then the beneficiary may claim the campaign's proceeds. Otherwise, all donors are free to reclaim their staked funds from the contract and the beneficiary receives nothing.

Logistics This represents a scenario in which two companies wish to jointly store and manage business records on a blockchain. The blockchain then serves as the authoritative source for these records, rather than a database system maintained by, and therefore subject to manipulation by, either company individually. In this work, we consider a specific case in which a smart contract represents a high-value asset sold by one company to the other. The contract tracks the item's shipment, delivery, and quality certification dates and adjudicates any warranty claims made by the buyer related to the item. This was motivated by our discussions with supply chain professionals at Intel, who engages in precisely these kinds of

business arrangements with suppliers of the equipment for its fabrication facilities.

SimpleMultiSig In the standard wallet concept for virtual currency, a single user controls a balance of funds and controls any payments made to external parties against this balance. Multi-signature wallets, in which multiple users jointly control a balance of funds, have been a popular smart contract application, with several standard implementations available [77]. One of these standard, and widely used, implementations was the victim of a severe security vulnerability [78].

The simplest multi-signature wallet involves two users, or "signers," – the contract's creator and an additional party declared when the contract is first initialized — who both must approve of any payment of funds. The contract may freely receive a deposit of virtual currency, but neither of its signers can unilaterally complete a payment. These signers trust the contract's implementation to hold each other accountable for this mutual approval process.

StaticMultiSig This represents a more advanced version of the previous multi-signature wallet. This contract allows an arbitrary collection of blockchain users to be designated as signers. However, signers may only be added to the contract in a designated initialization phase in which no payments may be issued. Once all signers agree, the contract begins normal operation, at which point no new signers may be added.

DynamicMultiSig This represents a more advanced version of the static multi-signature wallet in which new signers may be freely added at any time, rather than during a specific initialization phase. Payments must still be approved by all signers. If a new signer is added to the wallet while a payment is pending, i.e. it has approval from a subset of the signers, then the new signer must also approve of the payment before it can proceed.

ERC-20 This is arguably the most popular contract standard on Ethereum. It defines a relatively simple contract that represents and administers a virtual currency independent of Ether, i.e. it cannot be used to cover transaction fees on an Ethereum-based blockchain. The contract is initialized with a finite supply of tokens. Users may run transactions against the contract to transfer a sum of tokens they own to a designated recipient. There are countless ERC-20 instances running on Ethereum-backed blockchains today.

ERC-721 ERC-721 represents an exchange platform for non-fungible virtual tokens, i.e. virtual assets that are not freely interchangeable but instead represent unique items. The value of an ERC-721 token instance does not have a market-defined value, instead it has different values for different principals, much like collectibles in the physical world. ERC-721 contracts permit users to register ownership of new non-fungible assets and to exchange those assets for a sum of virtual currency (e.g, an ERC-20 token or Ether). There are in fact online marketplaces that are backed by an ERC-721 contract, such as `opensea.io` [76].

ERC-1202-Simple ERC-1202 defines a standard voting contract. A fixed set of options are declared when the contract is initialized, and any user may submit a vote for any of these options. Much like an auction, a voting contract has a deadline that is set at initialization. All votes must be submitted before this deadline, after which the contract is programmed to reject any new votes. In the simplest version of ERC-1202, each user counts equally, namely for one vote. The option with the highest number of votes is declared the winner after the deadline has expired.

ERC-1202-Weighted This is an advanced version of the previous ERC-1202 contract in which each user has a weight ascribed to her. When she votes for an option, her weight is credited to that option. The option with the highest total assigned weight at the time of the deadline is declared the winner. Weights could be determined by a variety of factors. Here, the contract also implements a simple token (making its functionality a rough superset of that of an ERC-20 contract) that may be freely exchanged. A user's weight in the voting process is equal to her holdings of the contract's integrated token. This allows for a rich set of capabilities such as delegating some or all of one's influence to another user, or of ascribing influence proportional to one's role in an organization.

ERC-1540 ERC-1540 is a contract that tracks ownership of an asset. The asset's owner creates the contract and initially has full control. She may transfer or sell the asset to a new owner by running a transaction against the contract. Or, she may issue a fixed number of shares in the asset. The contract then tracks ownership of all shares and facilitates their exchange. The asset cannot be sold or transferred when its shares are in circulation. If some individual acquires all outstanding shares, she may declare herself as the new owner and then is free to transfer or sell the asset or to issue a fresh batch of shares. This is all enforced by the contract's code, which tracks the outstanding shares owned by each user, supervises the exchange of these shares, and only allows a user to declare herself owner if she truly owns all shares.

ERC-1630 ERC-1630 standardizes what is known as a hashed time-lock contract. The contract's creator, say Alice, initializes the contract with a sum of virtual currency to hold in escrow and that is designated for some specific recipient, Bob. She also provides the output h of some standard hash function H and a deadline t. If Bob submits a value s to the contract such that $H(s) = h$, i.e., the preimage to Alice's original value, then he is compensated with the contract's deposit. Otherwise, if t arrives before Bob has made a successful claim, Alice is free to withdraw her original deposit. This contract enables what are known as atomic cross-chain swaps, in which multiple parties participate in a protocol involving contracts hosted on multiple blockchains to exchange assets similarly hosted on multiple blockchains. They have therefore been a subject of interest both in industry and academia [49].

ERC-1850 This standard builds on ERC-1630 to implement a loan, with an asset maintained on an external blockchain used as the borrower's collateral. This arrangement is called a cross-chain atomic loan. An ERC-1850 contract goes through a specific lifecycle. First, the lender deposits the sum of the loan in escrow with the contract and specifies a hash value h and deadline t. The borrower (out of band) puts their collateral in escrow on a separate blockchain. Once the lender sees evidence of this, she provides a secret s to the borrower (also out of band), such that $H(s) = h$. This allows the borrower to present evidence of the lender's approval to the contract, which then releases the loan to the borrower. If the borrower does not repay the loan to the contract by time t, the contract includes logic for the lender to run an auction against the borrower's collateral. Thus, ERC-1850 includes elements of ERC-1630 and of the auction use case described above.

ERC-780 This is one of the simplest contract standards. An ERC-780 contract simply acts as a registry for "claims" — arbitrary pieces of information about a *subject* submitted by some *issuer*. For example, the issuer could assert that the subject has a certain privilege or that they meet some qualification. One example might be a government attesting that a certain person is a citizen, or a company attesting that a customer is entitled to some discount or reward. By hosting these claims on a blockchain, the idea is that they are authoritative, tamper-resistant, and open for inspection by anyone. The standard itself is very simple, requiring little more than an index on the issuing identity and subject identity that maps to a collection of claims involving the two parties. It is, however, very similar to more advanced use cases such as using a blockchain-backed repository of security permissions as the backend for an authorization platform [3].

RockPaperScissors Games and puzzles have been popular use cases for smart contracts. The goal in all of these instances is to execute the game's logic, and thus enforce the game's rules, through a smart contract rather than a trusted mediating party. We chose rock paper scissors as a prototypical example of this class of smart contracts because it is simple yet still brings up relevant concerns and because its implementation as a contract was previously discussed in the literature [27]. Games have to deal with participants each acting in their own self interest, yet with a very clear incentive model - to reap the benefits of winning the game while avoiding the penalties of losing, even if that means achieving this result by circumventing the contract's intended protocol rather than winning the game outright. Therefore, a game contract's code must be engineered to account for this and to ensure that it is never possible, or at least not rational, for a participant to deviate from the expected sequence of steps.

One of the main issues in a rock paper scissors implementation is that each player must submit their choice of move, and be held committed to that move, without revealing it. Otherwise, the first player to submit their move to the contract would inevitably lose, as this would be visible to all participants on the blockchain, including the game's other player, who would simply submit the winning countermove. Instead, each player submits a hash of

their move and a privately chosen nonce. Once both moves are submitted, players reveal their move and nonce, which are validated against the previously recorded hash. The contract may include a timeout mechanism so that, if one of the players refuses to reveal her move before a deadline, the other player is automatically recorded as the winner.

DAO A decentralized autonomous organization (DAO) contract is roughly a combination of ERC-20 and ERC-1202. It implements a virtual token by tracking ownership and facilitating exchanges, per usual. A user invests in a DAO by purchasing some of these tokens. The DAO then executes voting procedures in which its investors decide how to use the DAO's holdings. For example, a user may propose that the DAO transfer a portion of its assets to a crowdfunding campaign contract. Thus, the DAO's tokens both reflect a stake of its total assets and give the bearer voting privileges. Typically, a DAO contract weighs each user's vote by the amount of tokens she possesses. DAO contracts have been an extremely popular subject of study, particularly for smart contract development and verification, because a DAO implementation was a victim of arguably the most severe security vulnerability in Ethereum's history.

3.2 Common Themes

While each of these case studies represents a different application scenario and features a unique set of requirements for design and functionality, there are a number of common themes that pervade multiple case studies. We identify and codify these themes here in order to inform the design of the programming interface we offer to contract authors using *Quartz*. Each theme represents a recurring design element or concern that is central to the successful operation of one or more of the contracts belonging to our body of case studies.

Event Ordering Many smart contracts are used as a means of creating and maintaining a secure log of real-world events. The contract acts as the authoritative source for information about the timing and relative ordering of these events. Alternatively, a contract may contain logic to ensure that operations against it are performed in a specific order.

Action Authorization Contracts very often need to restrict who may invoke a certain operation. There are actually a variety of circumstances in which this is required. A contract may have a notion of an owner or administrator who is allowed to take actions that other users cannot, or perhaps a group of such users. A contract may require that multiple parties independently approve an operation before it executes.

Token or Asset Ownership Contracts of this type act as evidence that a specific entity owns or has some stake in a physical or, more typically, digital asset. The contract must be structured carefully so that no malicious user is able to falsely claim they are the owner.

Time-Based Logic A contract may need to enforce restrictions not only on who can invoke a particular operation, but on *when* it can be invoked. Such a restriction can assume a variety of forms, such as enforcing a voting or bidding deadline, preventing a user from invoking operations too frequently, or enforcing a timeout mechanism to prevent a user from impeding forward progress.

Token or Asset Exchange Some contracts track the current owner of an asset and allow it to be transferred or sold for cryptocurrency. A common variant of this pattern is to issue shares in an asset, which may entitle their owners to some kind of dividend payment or to vote on decisions about the asset. In this case, the contract must track possession of these shares and allow shareholders to transfer or sell their holdings.

Joint Decision Making Contracts may be used to record and execute some process for reaching collective agreement among a group of users. This could be a simple requirement of unanimous consent (similar to one form of authorization described above) or it could be a more complex voting scheme.

Contingency Triggers Some contracts, like an auction, go through a linear sequence of steps in their execution. Others encounter branches, or contingencies, that lead to different outcomes. Examples include a crowdfunding contract, where the funding goal may or may not be met, or a voting contract where a motion may or may not pass.

Static Participant Set Some contracts place restrictions on the set of users who may interact with it. Any users outside of the set, who have access to the blockchain hosting the contract, may still submit a transaction against the contract, but the contract contains logic to ignore such requests. We say that this set of users is *static* when it is determined at initialization time or before any other operations may occur against the contract.

Dynamic Participant Set A contract's user set is *dynamic* if it may freely change over the lifetime of the contract, but remains explicitly enumerated. That is, the contract is still not open to any user with access to the blockchain network, although such these entities may still be able to join the collection of privileged users after the contract is initialized.

Explicit Incentives While all smart contracts must account for the fact that they are open to interaction with self-interested participants, some applications confront this more directly than others. ERC-1630 and ERC-1850, for example, include mechanisms that penalize users for failing to take timely action when they are expected to participate in the contract's protocol and prevent a single user's inaction (calculated or otherwise) from blocking forward progress. Game contracts, such as our rock paper scissors case study, offer a second example, where a contract must offer a mechanism for the game's players to commit to a move without immediately revealing the move.

	Event Ordering	Action Authorization	Token or Asset Ownership	Time-Based Logic	Token or Asset Exchange	Group Decision Making	Multiple Contingencies	Static Participant Set	Dynamic Participant Set	Explicit Incentives
Auction	✓	✓		✓						
Crowdfunding	✓	✓		✓			✓		✓	
Logistics	✓	✓		✓			✓	✓		
SimpleMultiSig		✓	✓		✓			✓		
StaticMultiSig		✓			✓	✓		✓		
DynamicMultiSig		✓	✓		✓	✓			✓	
ERC-20			✓		✓					
ERC-721			✓		✓					
ERC-1202-Simple		✓		✓		✓	✓	✓		
ERC-1202-Weighted		✓		✓		✓	✓	✓		
ERC-1540		✓	✓		✓		✓		✓	
ERC-1630		✓	✓	✓	✓		✓	✓		✓
ERC-1850	✓	✓	✓	✓	✓		✓			✓
ERC-780		✓								
RockPaperScissors	✓	✓		✓			✓	✓		✓
DAO		✓	✓	✓	✓	✓	✓		✓	

Table 3.2: Themes in Smart Contract Design Exhibited by Each Case Study

3.3 Insights on Language Design

The case studies and the contract design themes identified above bring forward a number of concerns to address in the design of a simplified contract programming language. In particular, they should inform such a language's main abstractions, primitives, and type system. Below, we discuss the most significant of these concerns and how they can be addressed through language design.

Contract Lifecycles Nearly all of the contracts above go through a sequence of phases in their operation. Different operations against the contract are permitted in different phases.

For example, an auction contract goes through a phase in which it is open to new bids and a settlement phase in which the seller redeems her proceeds. A contract facilitating the sale and service of a physical asset, as in our `Logistics` use case, may go through a sequence of phases reflecting the item's status as ordered, shipped, delivered, certified, or under service. In each phase, users may only manipulate the contract's state in specific ways, such as indicating that shipped item has been delivered. A contract language must offer contract authors a means of clearly enumerating these phases, the operations that are allowed in each phase, and under what conditions the contract moves from one phase to another.

Finite Virtual Resources Many contracts are used to track ownership, exchange, and consumption of finite virtual resources. These might be fungible tokens, shares in an asset, or the privilege of voting in an election. The underlying logic for such contracts tends to make use of a specific set of language features including unsigned arithmetic, struct data types to represent resource instances or to represent offers to buy or sell such instances, and often the ability to send and receive the blockchain's native currency (in this work, Ether).

Cryptographic Hashing Several of the contracts studied above, such as ERC-1630 and rock paper scissors, require cryptographic hashing functions. These are used to implement primitives like a basic commitment scheme or to allow a user to present knowledge of some secret as a credential to complete an operation such as withdrawing virtual currency. Note, however, that hashing has also been abused by contract developers in the past, particularly as a source pseudo-randomness that is ultimately vulnerable to manipulation by blockchain miners [8]. Therefore, we must attempt to develop a contract language that directly incorporates cryptographic hashing as a primitive but discourages inappropriate application of this primitive.

Time-Based Logic Contracts need to track and record timestamps and time durations for multiple purposes. First, the contract phases and operations described above are often subject to time-based constraints. A contract may only be permitted to stay in a particular phase for a specified time period, such as the bidding period of an auction. Also, a contract may need to record the time at which certain events occur in order to serve its role as an authoritative, tamper-resistant source of information. For example, a contract may track shipment and delivery dates to resolve potential future disputes in a purchase agreement. Therefore, a contract language should feature timestamps and durations as first-class data types with associated operations for time comparisons and arithmetic.

Authorization Operations against contracts are frequently only permitted by authorized users. It is up to the contract's code to properly enforce these permissions. Additionally, authorization constraints can come in a variety of forms. The simplest is when an operation may only be performed by a single, specific party such as the owner of an asset or the seller in an auction. However, as noted above, contracts often enforce the notion of user groups,

e.g., where any individual in a privileged set of users may complete an operation. Or, a contract operation may require agreement among multiple users, as in a multi-signature wallet. Here, the contract must track all prior approvals of the operation in question, which remains pending until it has received approval from a sufficient set of users. Thus, a contract language should allow programmers to express a wide range of authorization constraints involving checks against the invoking party's identity or membership in a group and should not require the programmer to handle the underlying bookkeeping involved in deferring operations that require approval from multiple parties.

3.4 Summary

This chapter has presented our methodology for selecting a set of representative and complete sample of case study contract applications, detailed each element of this sample, and identified a set of fundamental contract design patterns and concerns. This serves two important roles, introduced at the beginning of this chapter, that are both borne out in subsequent chapters. First, there is the issue of designing the contract description language at the heart of our approach, which we will introduce in Chapter 4 and precisely formalize in Chapter 5. Design patterns like authorization, time-based logic, and the definition of contract lifecycles to enforce an ordering of events or address multiple contingencies all figure prominently into language design.

These case studies also form foundation of our evaluation methodology — each one serves as a benchmark and, collectively, they cover a range of contract use cases that are of concern to blockchain developers and users. In Chapter 4, we argue for the expressiveness of *Quartz*, our contract description language, by demonstrating that it can be used to implement every case study in our sample. In the same chapter, we argue for *Quartz*'s conciseness by comparing the length of each case study's *Quartz* implementation to an equivalent Solidity implementation and to an equivalent specification written in TLA$^+$. In Chapter 6 we show the ability to identify flaws in our approach by examining particular case study contracts and vulnerabilities surface by *Quartz* in detail. Finally, the last element of our evaluation comes in Chapter 7, where we measure gas costs of *Quartz*-generated Solidity against those of handwritten Solidity for every case study.

Chapter 4

System and Language Overview

Thus far, we have motivated a more precise and methodical approach to the design and testing of smart contracts, proposed such an approach with a domain-specific language as the focal point, and discussed the selection and use of case study applications to inform and to evaluate this approach. In this chapter, we present *Quartz*, a prototype system that instantiates our proposed approach, establishes its feasibility, and serves as the subject of several evaluations. We describe *Quartz*'s language design and system architecture, discuss two example contracts in terms of both their state machine representations and their descriptions as *Quartz* code. We then present the first of our system evaluations, which examines the relative sizes of alternative contract representations in an effort to judge *Quartz*'s conciseness. Finally, we offer a summary of prior systems that leverage state machines as the primary programming abstraction and efforts in designing smart contract programming languages.

4.1 System Architecture

Quartz's software architecture is shown in Figure 4.1. It features three primary components, each with additional internal elements. The first component is a shared front end that implements the steps common to both translation paths. Depending on which translation the user has requested, either to TLA$^+$ or to Solidity, one of the two target-specific backends is then invoked and provided with the output of the common front end. More specifically, the front end involves lexing and parsing a state machine description written in the *Quartz* DSL, type checking, and validating state machine structure. This produces an abstract syntax tree annotated with type information, which serves as an intermediate form and is offered to the appropriate back end. We present *Quartz* syntax later on in this chapter and formalize the DSL's semantics and typing in Chapter 5.

Quartz's TLA$^+$ back end enables contract validation through model checking. It consumes the annotated AST produced by the front end and produces a formal specification of the user's original contract, written in TLA$^+$, and an independent configuration file for model checking which allows the user to specify bounds on the search space. This involves

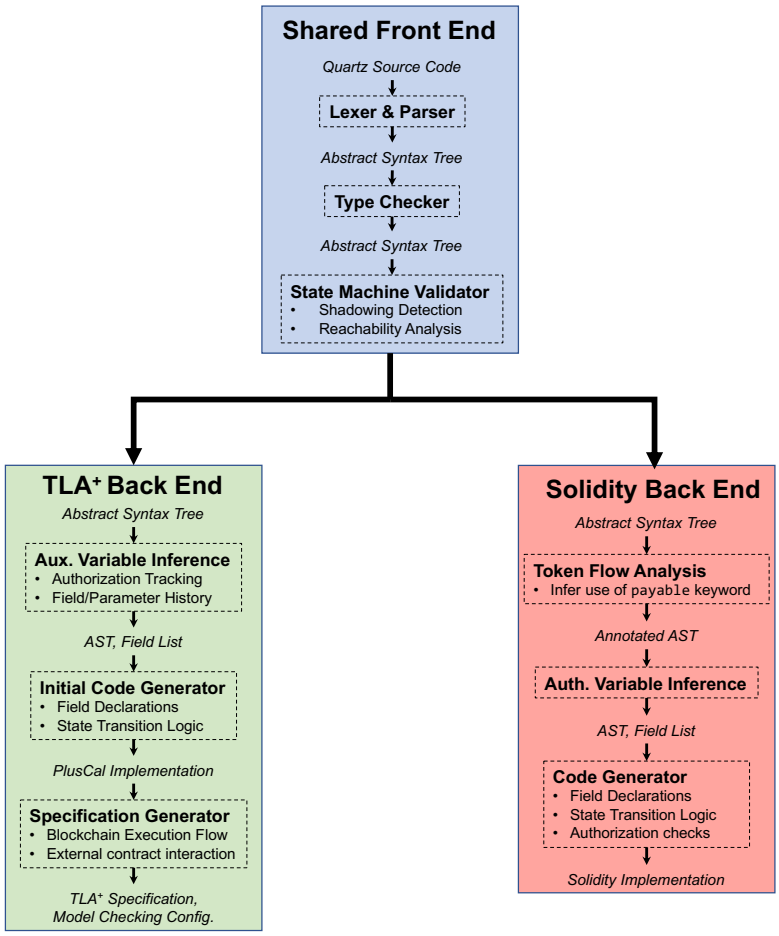


Figure 4.1: *Quartz* Architecture

generating PlusCal code implementing the logic of each state transition and then building a specification that encompasses this logic and captures the blockchain's execution model. This backend also infers any auxiliary state that needs to be tracked during model checking yet is not explicitly declared in the original contract description, such as authorization information or historical values of contract fields. The TLA$^+$ generation process is described in more detail in Chapter 6.

Finally, the Solidity back end allows an author to deploy an implementation of her contract to an active Ethereum-based ledger. It shares some similarities with the TLA$^+$ back end, such as inferring additional authorization state to track and, obviously, recreating state transition logic in the target language. Solidity requires that any external contract address that is the recipient of a token send must be explicitly annotated as a safety precaution. The Solidity backend traverses the abstract syntax tree to identify token flows in the contract's

state transitions and thus infers all locations in the generated code where such an annotation is required. *Quartz*'s Solidity tranlation is described in detail in Chapter 7.

System Implementation

We implemented the *Quartz* core in roughly 2800 lines of Scala code. We used Scala's parser-combinator library[1] to generate a combined lexer and parser for *Quartz* contract descriptions. Type checking and translation operate directly on the abstract syntax trees generated by the parser. The case studies detailed in Chapter 3 were handwritten in Solidity as a baseline for the evaluation described in Section 7.5, totalling about 1000 lines. We built reproducible workloads to exercise these contracts using Python and its Web3 library,[2] which consisted of about 1700 lines in total.

4.2 Contracts as State Machines

Language Structures

A contract definition in *Quartz* consists of an extended finite-state machine and an optional set of properties to verify about the machine's behavior. This structure allows developers to cleanly express a contract's different operations, constraints on their use, and the different phases of contract operation. A state machine consists of three pieces. The first is an optional sequence of definitions of any `Struct` types to be used within the contract. The second piece is a set of *fields*, each given a unique name and annotated with a type. *Quartz* supports simple types such as `Int` and `Uint`, parameterized types such as `Map`s and `Sequence`s, and `Struct` types. *Quartz* also includes types specifically useful for contract development such as `Identity` (a unique identifier for a ledger participant) and a `Timespan` type. A `HashValue` type is parameterized by a sequence of types indicating the structure of its preimage. It only supports equality checks with instances of the same type. This encourages the use of hashing for purposes like commitment schemes and capability-based access control while discouraging the use of hashing as a pseudo-random number generator, a practice that has introduced vulnerabilities in past contracts [8].

The last component of a state machine is a set of state *transitions*. Each transition consists of the following elements:

- A unique *name*, used to invoke the transition
- A *source state* and *destination state*
- A set of *parameters*, each given a name and type
- A *guard*, written as a predicate over the machine's fields and the transition's parameters

[1] `https://github.com/scala/scala-parser-combinators`
[2] `https://web3py.readthedocs.io`

- An *authorization predicate* restricts which parties may trigger a transition
- A *body*, written as a sequence of statements executed for their side effects

Quartz state machines are event triggered, and a transition is only eligible for execution if its guard is satisfied. The statements within a transition body are kept simple to facilitate model checking, with no branching constructs. They may either modify a field or transfer tokens to an external contract. An authorization predicate determines who may initiate execution of the associated transition, a particularly important concern for smart contract applications. *Quartz*'s authorization clauses allow contract authors to express rich semantics that are cumbersome to express using guards alone. They are built from three terms of the form i, satisfied when `Identity` i approves the transition, and the forms `any(`I`)` and `all(`I`)`, where I is of type `Sequence[Identity]`. These are satisfied when one or all members of the referenced group approve, respectively. These terms may be arbitrarily combined with Boolean `&&` and `||` operators.

The second, optional element of a contract description is a set of *invariants* regarding the state machine's possible execution traces. These are written as predicates over the state machine's fields, with some additional primitives. Predicates may refer to transition parameter values or to an aggregate sum over a `Sequence` or `Mapping` type. Additionally, a predicate can use `min` or `max` to refer to the minimum or maximum value that a variable assumes over the lifetime of the state machine. For example, given a state machine containing transition t with parameter p, `max(`t`.p)` refers to the maximum value of p ever used in an execution of t. This allows *Quartz* to check rudimentary temporal properties [73, 86].

Language Syntax

Figure 4.2 provides a formal definition of the syntax for *Quartz*'s domain-specific language. *Quartz* supports the standard arithmetic and Boolean operators, comparisons, and both `in` and `not in` operators to check for membership in an object of `Sequence` type. Literals of type `Bool`, `Int`, `Uint`, and `Timespan` are written as expected, with the possible exception of a `Timespan` instance, written as an integer followed by a unit such as `minutes` or `hours`.

Quartz transitions begin with a header of the form *source* `->` (*Parameters*) *destination*. This is followed by an optional `requires` block to express a guard and an optional `authorized` block to express an authorization predicate. Finally, the body of the transition is enclosed within braces and consists of a sequence of simple statements, like assignment to a field.

The *Quartz* language contains several contract-specific features. Many distributed ledgers, most notably Ethereum, have first-class support for virtual currency that may be bound to contracts and exchanged among them. State machines in *Quartz* use keywords to check their balance or disburse tokens to an external contract. If we wish to produce contracts for a ledger without first-class tokens, we can emulate this functionality by adding an extra field and the necessary operations to the generated implementations. Transition authorization is

treated as a first-class primitive in *Quartz*, unlike in Solidity and other contract languages. *Quartz* allows contract authors to express rich authorization constraints such as restricting an operation to any member of a particular group or requiring approval from all members of a group before it is executed. Finally, *Quartz* restricts communication between state machines. A state machine may send tokens to another state machine, but it cannot invoke another machine's transitions directly. This simplifies the expression and verification of contract logic. Note that *Quartz* makes no assumptions about the behavior of the recipient, which may or may not be another *Quartz* state machine, for model checking.

⟨*specification*⟩ ::= **contract** name '{' ⟨*structDecl*⟩* ⟨*fields*⟩ ⟨*transition*⟩* '}' ⟨*property-spec*⟩

⟨*fields*⟩ ::= **data** '{' ⟨*field*⟩* '}'

⟨*field*⟩ ::= name ':' ⟨*type*⟩

⟨*structDecl*⟩ ::= **struct** structName '{' ⟨*field*⟩* '}'

⟨*transition*⟩ ::= name ':' ⟨*sourceSt*⟩ '->' ⟨*params*⟩ dest ⟨*authPred*⟩ ⟨*guard*⟩ ⟨*transBody*⟩

⟨*sourceSt*⟩ ::= ε | source

⟨*params*⟩ ::= ε | '(' ⟨*param-list*⟩ ')'

⟨*paramList*⟩ ::= ⟨*param*⟩ ',' ⟨*paramList*⟩ | ⟨*param*⟩

⟨*param*⟩ ::= name ':' ⟨*type*⟩

⟨*guard*⟩ ::= ε | **requires** '[' ⟨*expr*⟩ ']'

⟨*authPred*⟩ ::= ε | **authorized** '[' ⟨*authExpr*⟩ ']'

⟨*transBody*⟩ ::= '{' ⟨*stmt*⟩* '}'

⟨*type*⟩ ::= **Int Uint** | **Timestamp** | **Timespan** | **Bool**
| **Map** '[' ⟨*type*⟩ ']'
| **Sequence** '[' ⟨*type*⟩ ']'
| **HashValue** '[' ⟨*typeList*⟩ ']'

⟨*typeList*⟩ ::= ⟨*type*⟩ ',' ⟨*typeList*⟩ | ⟨*type*⟩

⟨*lValue*⟩ ::= x | ⟨*mapRef*⟩ | ⟨*structRef*⟩

Figure 4.2: EBNF Definition of *Quartz*'s DSL (Continued Below)

⟨*mapRef*⟩ ::= ⟨*lValue*⟩ '`[`' ⟨*expr*⟩ '`]`'

⟨*structRef*⟩ ::= ⟨*lValue*⟩ '`.`' ⟨*expr*⟩

⟨*expr*⟩ ::= **balance** | **sender** | **now** | *b* | *i* | *u* | *t* | ⟨*lValue*⟩
| **min** '`(`' ⟨*expr*⟩ '`)`' | **max** '`(`' ⟨*expr*⟩ '`)`'
| **size** '`(`' ⟨*expr*⟩ '`)`' | **hash** '`(`' ⟨*expr*⟩ '`)`'
| ⟨*expr*⟩ ⟨*binOp*⟩ ⟨*expr*⟩

⟨*authExpr*⟩ ::= *x* | '`any`' *x* | '`all`' *x*
| ⟨*authExpr*⟩ '`||`' ⟨*authExpr*⟩
| ⟨*authExpr*⟩ '`&&`' ⟨*authExpr*⟩

⟨*stmt*⟩ ::= ⟨*lValue*⟩ '`=`' ⟨*expr*⟩
| **send** ⟨*expr*⟩ '`to`' ⟨*expr*⟩
| **sendAndConsume** ⟨*expr*⟩ '`to`' ⟨*expr*⟩
| **append** ⟨*expr*⟩ '`to`' ⟨*expr*⟩
| **clear** ⟨*expr*⟩
| **if** '`(`' ⟨*expr*⟩ '`)`' '`{`' ⟨*stmt*⟩* '`}`'
| **if** '`(`' ⟨*expr*⟩ '`)`' '`{`' ⟨*stmt*⟩* '`}`' **else** '`{`' ⟨*stmt*⟩* '`}`'

⟨*propertySpec*⟩ ::= **properties** '`{`' ⟨*expr*⟩* '`}`'

b ∈ **bool** *i* ∈ **Int** *u* ∈ **Uint**
s ∈ **String** *t* ∈ **Timespan**
x, name, structName, source, dest ∈ ⟨*identifier*⟩

Figure 4.2: EBNF Definition of *Quartz*'s DSL

4.3 Example Contracts

We will consider two contract use cases here and in future chapters. They serve to motivate the features of *Quartz*'s domain-specific language, demonstrate how these features are used, and illustrate generation of contract specifications in TLA$^+$, generation of contract implementations in Solidity, and the contract validation process. The first is the auction contract defined in Chapter 3, which raises issues around subtleties in a contract's control flow and related vulnerabilities that can be caught with *Quartz*'s validation. The second, Chapter 3's static multi-signature wallet case study, is primarily concerned with enforcing authorization constraints, which can be expressed concisely in *Quartz* but involve several tradeoffs in their translation to Solidity.

Auction

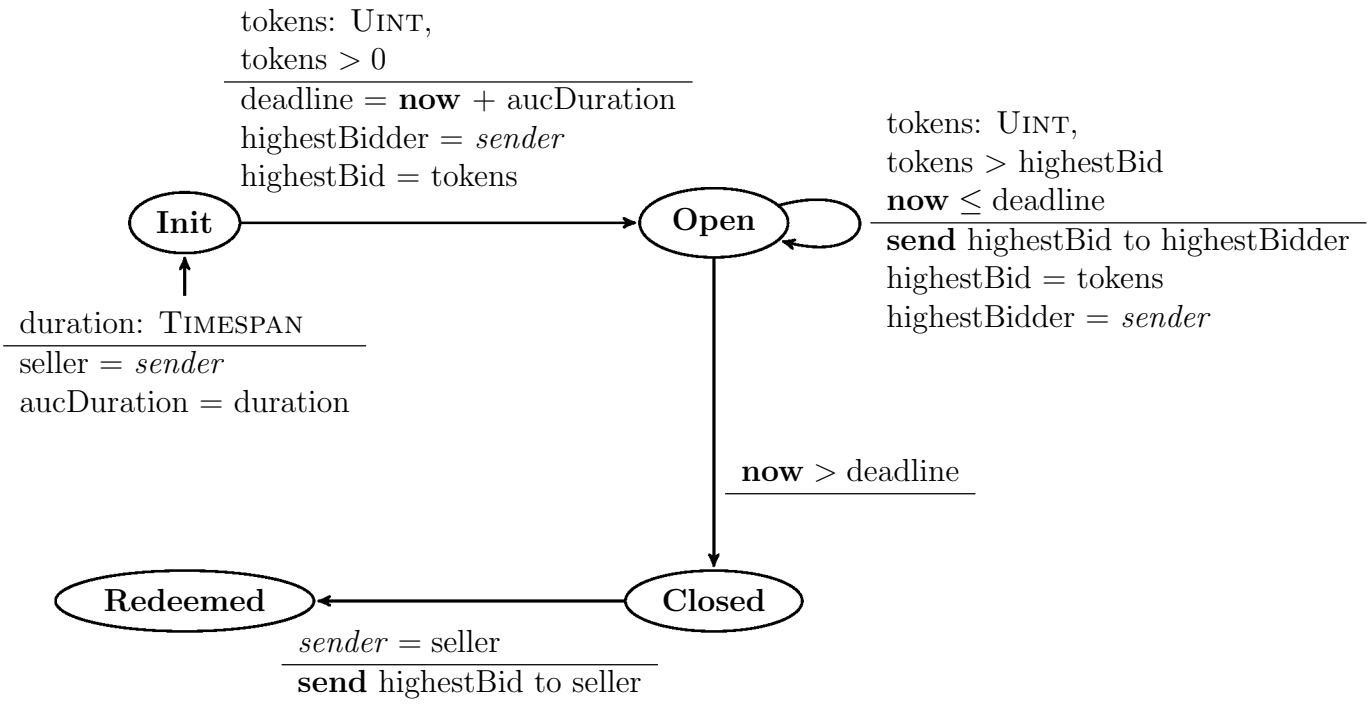


Figure 4.3: A State Machine for an Auction Contract

The purpose of our auction contract is to accept a sequence of ascending bids, each backed by a token deposit, from any potential party for an item put up by a specific seller, who is responsible for deploying the contract. It is up to the contract's implementation code, which is executed exactly as written by the underlying distributed ledger, to properly enforce the auction's terms. First, the issuer of the highest bid, regardless of their identity, is duly recorded as the winner. Bids are accepted up until a publicly declared deadline, which cannot be adjusted after the start of the auction by any party. Any principal who issued a losing bid is able to recover their tokens, while the seller may only claim the auction's proceeds once it is closed.

Our auction features four phases of operation. Its representation as a state machine is shown in Figure 4.3. Each transition is annotated with a guard, written above the horizontal line, and actions shown below the horizontal line. When the contract is deployed, its creator is recorded as the seller. The contract begins its life in the **Init** phase, awaiting the first bid. As soon as a bid arrives, the bid and its sender are recorded, and the contract transitions to the **Open** phase. Here, an arbitrary number of subsequent bids may be received and recorded, as long as each exceeds the previous highest bid. Unlike in the previous phase, the contract now must also refund the newly-supplanted highest bidder. Finally, once the auction's deadline

has passed, any party may move to close the auction. Only once the **Closed** phase is reached can the seller claim their earnings, with a transition into the **Redeemed** phase.

The equivalent *Quartz* code is shown in Figure 4.4. It begins with a list of *fields* annotated with their types before a sequence of *transition* definitions. The contract's states are not explicitly enumerated; rather, they are inferred as the union of all states referenced in transitions. The `requires` keyword is used to define a predicate over contract state that must be satisfied for the transition to fire. *Quartz* defines several globally available values, including the transition's invoking party (`sender`), the current time (`now`), the contract's balance of tokens (`balance`), and the tokens sent with the current transition (`tokens`). *Quartz* is not sensitive to whitespace. For example, a transition's states and guard may be written on the same line, as in line 29, or separately, as in lines 22 and 23.

```
contract Auction {
    data {
        Seller: Identity
        HighestBid: Uint
        HighestBidder: Identity
        Duration: Timespan
        Deadline: Timestamp
    }

    initialize: ->(duration: Timespan) init {
        Seller = sender
        Duration = duration
        HighestBid = 0
    }

    initialBid: init ->(tokens: Uint) open {
        Deadline = now + Duration
        HighestBid = tokens
        HighestBidder = sender
    }

    submitBid: open ->(tokens: Uint) open
    requires [ tokens > HighestBid && now <= Deadline ] {
        send HighestBid to HigestBidder
        HighestBid = tokens
        HighestBidder = sender
    }

    close: open -> closed requires [ now > Deadline ]

    redeem: closed -> redeemed authorized [ Seller ] {
        send HighestBid to Seller
    }
}
```

Figure 4.4: An Auction Contract Written in *Quartz*

Multi-Signature Wallet

Our multi-signature wallet features three states. When the contract is deployed, its creator is added as the first authorized signer who may approve payments. The contract begins its life in a **Config** state where new signers may be added by any previously declared signer. Once all signers agree, the contract enters an **Open** state where any signer may propose a payment. If that payment is approved by all signers, funds are sent to the intended recipient. Finally, all signers may agree to transition to a **Closed** state, where any remaining funds may be returned specifically to the contract's creator, again with the approval of all signers. In any phase, the contract accepts external deposits from any source.

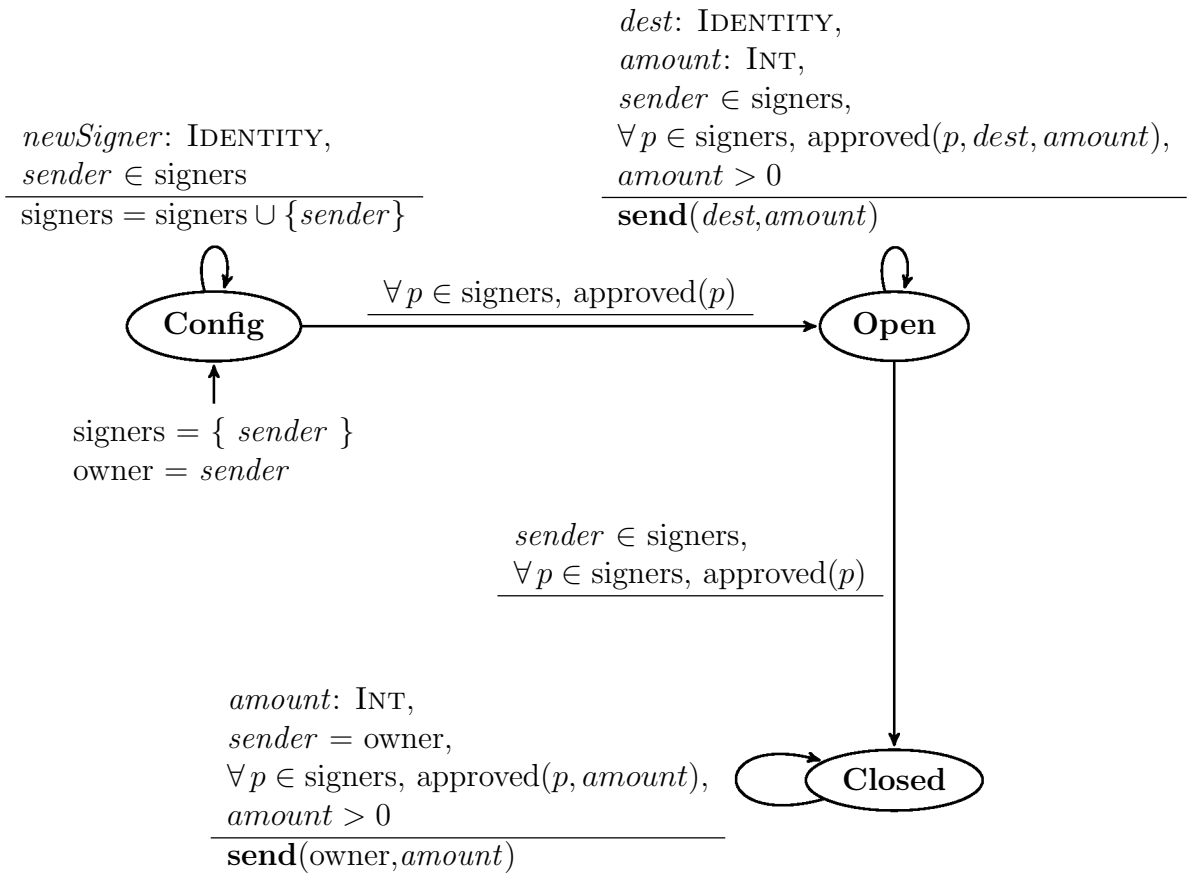


Figure 4.5: Multi-Signature Wallet State Machine

The logic for this contract is specified as a state machine in Figure 4.5, with notation identical to that of Figure 4.3. Transitions for deposits to the account are omitted for brevity. In guards, *approved* represents a Boolean-valued function that determines if principle p authorizes execution of the relevant transition with the specified input parameters. The *Quartz* definition of this state machine is given in Figure 4.6. Here, the `authorized` construct

```
contract MultiSig {
  data {
    Owner: Identity
    Signers: Sequence[Identity]
  }

  initialize: -> config {
    Owner = sender
    add sender to Signers
  }

  addSigner: config ->(newSigner: Identity) config
  authorized [ any Signers ] {
    add newSigner to Signers
  }

  declareOpen: config -> open
  authorized [ all Signers ]

  pay: open ->(recipient: Identity, amount: Int) open
  authorized [ all Signers ]
  requires [ amount > 0 ] {
    send amount to recipient
  }

  close: open -> closed authorized [ all Signers ]

  refund: closed ->(amount: Int) closed
  authorized [ all Signers ]
  requires [ amount > 0 ] {
    send amount to Owner
  }
}
```

Figure 4.6: *Quartz* Multi-Signature Wallet

is more prominent than in the auction implementation. It is used to defer execution of a transition until the requisite parties have approved. When using the `all` primitive, or when an authorization predicate contains a conjunction of multiple clauses, this requires maintaining state across transition invocations, namely the parties that have invoked (and thus approved) the transition in the past. We will see how this figures into TLA$^+$ and Solidity code generation in the following chapters.

4.4 Results: Contract Size

We use lines of code as a proxy for code complexity and required developer effort. Here, we are interested in the extent to which *Quartz* enables concise expression of rich contract logic, particularly when compared to Solidity, the de facto standard programming language for smart contracts. We also seek to determine if *Quartz* introduces overhead by generating significantly more verbose Solidity code than would be produced by a Solidity programmer. This is particularly relevant in the blockchain setting, where the size of a contract's compiled bytecode directly impacts the gas costs of deploying the contract to the ledger. Finally, we quantify developer effort saved when using *Quartz* for model checking by measuring the size of a contract's TLA$^+$ representation.

Case Study	**Quartz**	**Handwritten Solidity**	**Generated Solidity**	**TLA$^+$**
`Auction`	33	40	53	205
`Crowdfunding`	31	39	50	193
`RockPaperScissors`	39	76	75	261
`Logistics`	20	35	46	172
`SimpleMultiSig`	12	34	30	137
`StaticMultiSig`	13	38	44	125
`DynamicMultiSig`	15	39	50	141
`ERC-20`	43	49	50	209
`ERC-721`	48	59	86	227
`ERC-1202-Simple`	24	50	49	197
`ERC-1202-Weighted`	42	57	73	261
`ERC-1540`	114	143	181	464
`ERC-1630`	22	23	37	161
`ERC-1850`	110	182	242	588
`ERC-780`	16	17	24	140
`DAO`	126	126	150	390

Table 4.1: Lines of Code to Express Case Studies

Table 4.1 shows the lines of code needed to express all of our case studies in *Quartz* and in Solidity. It also shows the lines of Solidity and TLA$^+$ generated from the *Quartz* implementation of each contract. Each row in the table corresponds to one of the case studies presented in Table 3.1. When writing Solidity for each case study, we modified an existing code base (cited in Table 3.1) whenever possible rather than starting from scratch to ensure we produced idiomatic Solidity code. We were also careful to remove any comments or extraneous lines of code, like redundant getter functions that have no *Quartz* equivalent, to make the comparison as fair as possible.

When comparing the length of a *Quartz* contract to its handwritten Solidity equivalent, the case studies fall into two categories. First, there are the instances where the *Quartz* representation is significantly shorter. This includes `Logistics`, `ERC-1202-Simple`, `ERC-1850`, `RockPaperScissors`, and the `MultiSig` variants. These use cases tend emphasize two design elements. First, they may involve authorization restrictions on contract operations. Second, they may involve the enforcement of a contract lifecycle featuring distinct phases of operation in which different sets of operations are allowed. For example, in a multisignature wallet, only authorized parties may enact a payment of funds. In `Logistics` or `RockPaperScissors`, the contract goes through a sequence of phases in which different parties must act to move the contract's proceedings forward. *Quartz*'s DSL was designed for concise and clear expression of authorization constraints, and the state machine structure naturally captures the notion of a contract lifecycle, so it follows that *Quartz* contracts with these features are more concise.

Quartz's conciseness compared to Solidity becomes more marginal for the other case studies. Benchmarks like `Auction`, `Crowdfunding`, `ERC-20` are only somewhat shorter when written in the DSL. Others like the `DAO` and `ERC-780` are roughly the same. This is because certain elements, such as logical checks, arithmetic operations, and updating contract fields, are no more concise in *Quartz* than they are in Solidity. Applications that more heavily emphasize these elements lead to similarly sized representations in either *Quartz* or in Solidity. A good example of this is a comparison of `ERC-1202-Simple` against `ERC-1202-Weighted`. Both are implementations of a standardized voting scheme, but the latter allows different entities to be assigned a higher or lower weight to their vote. Once this weighting scheme is introduced, which involves additional data structures to track weights and additional arithmetic to enforce them, the advantage of *Quartz* over Solidity, at least in terms of brevity, diminishes.

Quartz typically, but not always, generates Solidity code that is more verbose than the handwritten equivalent. There are cases where the generated code is shorter than the handwritten code. This is often the case when the handwritten Solidity leverages domain knowledge to use more verbose but arguably more efficient data structures, particularly to track authorization, than the general code produced by *Quartz*.

Finally, significantly more TLA$^+$ code is needed to express each contract than Solidity or *Quartz* code. This is mainly because a contract's TLA$^+$ specification expresses both the contract's logic and its execution semantics. In particular, the TLA$^+$ representation of any contract must describe the main invocation loop in which any user may invoke any of the

contract's transitions with arbitrary parameter values. It must also express the potential for reentrant execution after a `send` and exception handling. Therefore, even a short contract generates a relatively lengthy TLA$^+$ specification.

4.5 Related Work: State Machine-Based Development and Testing

A number of prior works explore the use of state machines as a fundamental programming abstraction and organizing principle. *Quartz* is particularly inspired by frameworks that exploit the natural coupling of state machines with model checking for validation as described in Chapter 2. A classical example of this approach is the Teapot [21] platform for development and validation of cache-coherence protocols in distributed systems. Intended applications include software such as shared-memory execution platforms and distributed file systems, where the constituent nodes may elect to store local copies of global state and need to coordinate how this state is shared and maintained.

Much like in *Quartz*, a Teapot user writes her protocol as a state machine in a domain-specific language. This protocol description enumerates all states and, within each state, how a node should respond to receipt of each possible message type. Also like *Quartz*, Teapot translates the state machine description into two targets: a formal specification suitable for analysis by the Murphi model checker [29], and an implementation in C. Model checking allows Teapot to check cache coherence protocols for deadlocks and for violations of user-specified invariants. Teapot's programming model is more powerful than *Quartz*'s. It permits constructs like loops and also features continuations. The latter are intended to simplify implementation of situations where a participant in the protocol expects a specific sequence of messages and must wait to perform some action until the next element in the sequence is received.

Teapot only offers built-in support for a particular set of data types that are cleanly translated to both C and Murphi. If a developer wishes to include a user-defined data type, e.g., a struct, in her protocol, she must manually specify how the type is represented in Murphi. This is unlike *Quartz*, where the user is never expected to write any TLA$^+$ code. Moreover, while *Quartz* assumes execution on an Ethereum-based blockchain, Teapot is not tied to a particular execution environment. This is an advantage in the sense that it makes Teapot flexible. The user can manually specify which sequences of events (i.e., messages received) are possible in the protocol's environment or try to reuse an existing specification. However, this situation is a disadvantage in that it increases the burden on the developer by expecting her to write such a specification. For example, the xFS developers reported a need to explicitly model blocking system calls to the local kernel before they could validate their protocol using Teapot. Although *Quartz* is restrictive in its focus on blockchains, it fully captures the semantics of this environment without the involvement of the user.

P [28] is a second, more recent instance of programming based on state machines coupled

with validation based on model checking. Its creators describe its target domain as asynchronous, event-driven programming. Example applications include the message handling for distributed protocols described above and device driver software, which must coordinate interactions between the operating system and peripheral hardware. As in *Quartz* and Teapot, P expects developers to write a state machine description in a domain-specific language. This description is then checked for deadlocks or violations of safety properties (written explicitly by the user with `assert` statements) using the Zing model checker [5].

P is similar to Teapot in its advantages and disadvantages relative to *Quartz*. Its domain-specific language offers more powerful primitives than *Quartz*, including a form of procedure call. A P state machine may also choose to defer an incoming message, placing it on a persistent queue where it may be popped by a future transition execution. This diverges from the semantics of *Quartz*, which features an implicit queue of transition invocations, formalized in Chapter 5 that may not be freely manipulated.

A P user must specify the semantics of the environment in which her state machine executes by writing an additional "ghost state machine" to interact with the original state machine. This ghost state machine is responsible for thoroughly exercising the developer's implementation by emitting events. P therefore validates the user's implementation by exploring traces of repeated interactions in which the ghost state machine emits an event and the original state machine response. The P compiler elides the ghost state machine from the C implementation it generates. As with Teapot, this design choice presents an additional burden on the developer that is not present in *Quartz* because of its more specific focus on blockchains.

State machine-based software design has also been applied to smart contracts. One of the first such efforts is FSolidM [66], a tool for structuring Solidity contracts as finite state machines. FSolidM offers users a graphical editor for building smart contracts in which they draw a pictorial representation enumerating all of the machine's states and transitions. Like in *Quartz*, transitions may have guards to restrict their execution. FSolidM also offers a rudimentary authorization mechanism. A contract author may declare a dynamic list of contract administrators and restrict execution of certain transitions to members of this list. This is not as powerful as *Quartz*'s authorization logic, which allows arbitrary composition of authorization restrictions referencing multiple individuals or groups and allows execution of a transition to be deferred until multiple parties have expressed approval.

FSolidM offers tools to deal with concurrency that are not available in *Quartz*. First, FSolidM features an explicit locking mechanism, implemented using a simple boolean flag, to prevent reentrant execution of transitions. Second, FSolidM offers a form of optimistic concurrency control in which a contract is augmented with a sequence number field. Any transaction against the contract must provide the sequence number last seen by the caller and is rejected if this does not match the current value of the field. In this way, transactions, i.e., state transitions, may only occur when the state of the contract has not been modified without the knowledge of the caller. Although neither of these mechanisms are particularly complex and could be replicated with *Quartz* code, FSolidM conveniently generates their implementations on behalf of the contract author.

FSolidM diverges from *Quartz* in that it does not actually produce a complete contract implementation from the developer's state machine description. Rather, it generates a partially-filled Solidity template that the user must complete by hand. More specifically, the template contains all of the contract's functions, corresponding to the state machine's transitions, and it contains the fields needed for the authorization and concurrency control strategies described above. However, the body of each transition, i.e., the actions taken when the transition fires, must be written directly in Solidity by the user.

VeriSolid [67] extends FSolidM to offer validation, combining contracts as state machines with model checking validation much like *Quartz*. The user still builds a state machine using FSolidM and completes the resulting Solidity template. She can then augment this Solidity implementation with a set of contract properties written in a DSL. The Solidity code is converted into a specification for the nuXmv model checker [18], while the properties are converted to a formal representation in Computational Tree Logic (CTL). Much like in *Quartz*, this gives rise to an iterative development model in which a contract author builds her contract, runs it through the validation process, and refines her implementation based on problems identified by the model checker. One advantage of VeriSolid over *Quartz* is its full support for liveness properties, i.e., asserting that some desirable condition will eventually occur. However, VeriSolid also imposes restrictions that *Quartz* does not. First, it does not model reentrant execution and therefore cannot identify some of the contract vulnerabilities discussed in Chapter 2. Second, the properties it enables are of a very specific form: they concern only the order in which code statements are executed. A user may not assert more flexible variants such as constraints on the values assumed by a contract field, or absence of an unbounded token withdrawal as they can in *Quartz*.

4.6 Related Work: Contract Programming Languages

Smart contracts have become a very popular domain for new programming languages, particularly as shortcomings to Solidity [36] have emerged. Many of these languages therefore cast themselves as contrasts to Solidity, striving for greater safety, predictability, and robust operation in the face of potentially adversarial users. There are a variety of approaches to achieving these properties including contract-specific language primitives, an emphasis on functional programming, advanced type systems, and language design focused on suitability for verification. Like traditional, general-purpose programming languages, any contract language represents a choice in the tradeoffs between ease of use, safety, expressiveness, and amenability to verification.

One class of contract programming languages are the simplified, domain-specific languages that target a specific application domain. One of the early works in this area [40] offers a tool in which developers express the operations and rules for a contract in a DSL meant to resemble natural language. For example, the author of a ballot contract may write "People vote only once and people must not vote after the deadline." These statements are then translated into a Solidity template that the author completes by hand. On

one hand, the use of natural language allows users without programming expertise to build smart contracts, but the generation of only a Solidity template, rather than a complete implementation, offsets this benefit by requiring knowledge of Solidity programming to create a finished product. *Quartz* also offers a domain-specific language, but it is meant to resemble a simplified programming language rather than natural language. It therefore requires some programming expertise, although no expertise in Solidity specifically, and generates a complete implementation, not a template, from source.

There are other domain-specific languages that, more like *Quartz*, feature grammars structured around traditional programming constructs rather than natural language. One example is Findel [14], a domain-specific language specifically for the implementation of derivative contracts. It follows on previous work [80] that proposes DSLs for derivatives before blockchains existed as an enforcement mechanism. These derivatives are bilateral agreements between an *owner* and *issuer* about the exchange of an asset, which may involve contingencies and time-based constraints. The code for a Findel contract explicitly defines the structure of an abstract syntax tree, much like the S-expressions familiar to Lisp programmers. Nodes in this AST may feature blockchain-specific primitives such as the exchange of virtual currency or the enforcement of a time-based restriction on an action. A contract deployed on the Ethereum blockchain acts as a unified marketplace for Findel contracts and interprets their ASTs. Findel contract transactions and execution therefore proceed through this Ethereum contract. Findel, like *Quartz*, therefore assumes no knowledge of Solidity from the user even when it leverages an Ethereum-based blockchain. However, it is a DSL that aims to fill a specific niche in the blockchain ecosystem, where *Quartz* seeks to support a broad range of application domains.

The second class of contract programming languages set aside the simplicity of domain-specific languages for the expressiveness of mainstream, general-purpose languages [2,37,83]. Several of these languages [55, 56, 81] adopt a functional paradigm with the rationale that functional code is more easily analyzed and verified through static analysis tools. For example, the Pact and Plutus contract languages feature syntax and primitives very similar to Lisp and Haskell, respectively. They are instead distinguished by the addition of blockchain-specific features. Pact, for example, offers a globally persistent tabular data store (maintained by the underlying ledger) as a first-class primitive. The Tezos blockchain takes another approach, offering a high-level programming language [92] that compiles to a low-level stack-based language [91] designed to be amenable to formal analysis. In all of these cases, it may arguably be easier to reason about contract behavior when reading source code than it is for Solidity. However, these languages do not incorporate validation as a first-class element of their workflows as is the case for *Quartz*. For example, the Tezos developers advocate manual use of a theorem prover like Coq to analyze and prove properties of contracts expressed in their stack-based language but do not provide any tools or features to assist in this process.

A second line of work has explored the use of sophisticated type systems to ensure contract safety at compile time. This trend is already underway in mainstream software development with growing adoption of the Rust programming language [82], which features a compiler

that is capable of verifying memory safety. The Libra blockchain has introduced the Move language [17]. A key primitive in Move is the notion of a *resource*: a virtual asset with a finite supply enforced by the type system itself. For example, once contract code uses a reference to an asset instance to transfer that instance to a new owner, that reference becomes invalid, preventing invalid, duplicate transfers. This functionality makes Move's type system more powerful than Rust's — it is a linear type system rather than an affine type system. Obsidian [23, 24] also adopts a linear type system, but combines it with state machines as an organizing programming model, much like *Quartz*. Both Move and Obsidian are more expressive than *Quartz*, and they feature more powerful compile-time checks than *Quartz*. However, Move and Obsidian are only capable of ensuring the absence of a fixed class of contract flaws. They cannot offer the same flexibility as *Quartz*'s verification of contract-specific properties through model checking.

Scilla [87] is a functional contract programming language that also focuses on state machines. It is a more expressive, but also more complex, language than *Quartz* with syntax resembling OCaml and Standard ML. It offers a wide array of features such as lists, options, maps, dependently-typed byte strings, and pattern matching. A Scilla contract is specified as a "communicating automaton" — a state machine that fires a transition when it receives a message and may emit messages to specific external automata. An author writes each transition and specifies the logic to execute when it fires. To avoid the reentrancy issues that have plagued Solidity, a Scilla state machine may only emit a message at the conclusion of a transition, avoiding any transfer of control while the transition is in progress.

Scilla has several strengths that *Quartz* does not share. It is more expressive than *Quartz* as it takes after fully-featured functional programming languages rather than a DSL. Scilla allows communication, although restricted, between separate contracts. Scilla users can thus deploy applications built out of multiple, cooperating contracts where *Quartz* currently supports only monolithic contracts. Moreover, Scilla detects integer overflows at runtime and raises an error, protecting contract users against an area of potential vulnerability that *Quartz* does not address.

However, *Quartz* surpasses Scilla in its support for flexible contract verification. Scilla offers limited verification in the form of static analysis tools. Users can write their own analysis logic, essentially specifying any bookkeeping and reasoning that occurs over an AST traversal, but the logic must generalize to any potential contract and therefore is only suitable for identifying general code patterns that may be problematic rather than contract-specific properties as in *Quartz*. For example, the Scilla authors describe static analyzers to predict gas costs or to identify incorrect accounting when tracking asset ownership (similar to the linear type systems of Move and Obsidian), but these analyzers cannot reason about potential execution traces and invariants like a model checker does.

4.7 Summary

In this chapter, we put our proposed approach into practice through the design and implementation of *Quartz*. Its domain-specific language is designed specifically to address the recurring themes in contract applications raised in Chapter 3, while its two translation paths enable verification through model checking and seamless deployment to Ethereum-based blockchain systems. We laid out the software components backing these translation paths and how they fit together to form a larger system. Through multiple examples, we saw how a contract is framed as a state machine and how this can be expressed in the *Quartz* language. Our evaluation of contract sizes, in terms of lines of code, establishes two conclusions. First, *Quartz*, and more broadly an approach focused on finite state machines as the fundamental abstraction, is expressive enough to support the implementation of a broad range of applications. Second, state machine-based contracts written in *Quartz* are no more verbose than their Solidity counterparts, and are often more concise when issues like authorization arise.

With a working system implementing of our proposed methodology for contract engineering in hand (described more formally in Chapter 5), what remains is to validate the anticipated benefits as well as the potential costs of using this system. Chapters 6 and 7 address this task. In Chapter 6, we discuss the translation to TLA$^+$ for model checking in more detail and establish *Quartz*'s ability to surface contract vulnerabilities. Next, in Chapter 7, we discuss translation to Solidity and quantify the cost of using *Quartz* against writing Solidity directly in terms of contract execution efficiency.

Chapter 5

Language Formalisms

This chapter formally describes the structures and semantics of the *Quartz* contract description language. We begin with the definition of a *Quartz* state machine and its components. A *Quartz* contract is a written specification of these components, which forms a particular instance of a state machine. Next, we describe the characteristics of a well-formed state machine. The *Quartz* compiler ensures that a contract possesses these characteristics before it proceeds with code generation. We then present the operational semantics of a *Quartz* contract. These are particularly important as they dictate both the search space for model checking and the implementation of *Quartz* contracts in Solidity. Finally, we provide the rules used for *Quartz*'s type checking.

5.1 State Machine Structure

A *Quartz* contract is an addressable, extended finite state-machine formally defined as a 4-tuple $\langle Q, q_0, T, F \rangle$ where:

- Q is a set of states
- q_0 is the initial state
- T is a set of *transitions*, defined below
- F is a set of fields, each with a specific name and type

Each state is uniquely identified by a symbolic name. The state machine is always in one particular state at any point in time, and the machine may undergo a *transition*, moving from one state to another, when that transition fires. Note that a state is distinct from the current *status* of the machine, which is a product of its current state, the values of its fields, and so on. This will be defined more formally below.

A *transition* $t \in T$ is defined as a 7-tuple $\langle$name, src, dst, g, a, P, $B\rangle$ where:

- *name* is the transition's unique name

- *src* is the transition's starting state
- *dst* is the transition's ending state
- g is a Boolean-valued *guard* expression over the fields of the state machine and transition parameters that must be satisfied for the transition to fire
- a is an *authorization predicate*, defined below, that must also be satisfied for the transition to fire
- P is a set of transition-specific parameters, each with a name and type
- B is the transition's *body* — a sequence of statements executed in order when the transition fires

A *Quartz* state machine is event-triggered and addressed by a globally unique identifier, provisioned by the underlying distributed ledger that hosts the contract. *Quartz* treats these addresses as first-class values. They may be assigned to fields or transition parameters, and certain statements allow a state machine to emit messages with a specific target address.

This allows external, addressable entities (either an end user or another state machine) to direct messages to the machine to invoke transitions. Messages are of the form $m = \langle i, t, V \rangle$ specifying the identity i of the sender, a transition t to execute, and a (possibly empty) set V of value assignments for t's parameters.

The current status S of a state machine is defined as a 7-tuple $\langle q, \sigma, \alpha, M, s, \gamma, C \rangle$ where:

- q is the machine's current state
- σ is a mapping from names to values representing the current *context*. It always includes the state machine's fields as well as built-in values **sender**, **balance**, and **now**.
- α is a mapping from transitions and transition parameter values to the set of identities that have authorized invocation of that transition with that particular set of parameter assignments.
- M is a queue of transition invocation messages. The machine may only read the head of the queue or remove the head of the queue. It may not inspect any other queue elements or the queue's length.
- s indicates which statement within a transition to execute. The special statement form $\top$ indicates that the machine has not yet started execution of any transition, while $\bot$ indicates that a transition has just completed. Thus when executing some transition with body $B = [b_1, \ldots, b_n]$, s will assume the sequence of values $[\top, b_1, \ldots, b_n, \bot]$.
- γ is an authorization clause that must be evaluated to determine if the invoked transition may proceed, or $\square$ if no such determination is in progress.

- C is a stack containing records of in-progress transitions. More specifically, element i of the stack is the statement to evaluate on resuming the i^{th} ancestor of the currently executing transition.

A machine's initial status, before it executes its initial transition, is therefore $\langle q_0, \{f.\text{name} \mapsto 0_{f.\text{type}} : f \in F\}, \emptyset, \varepsilon, \top, \square, \varepsilon\rangle$, where ε denotes an empty sequence and 0_T denotes the zero element of type T.

Static Contract Validation

Before any translation takes place, *Quartz* performs a variety of checks on a contract specification. A well-formed state machine has a unique initial state with no guard or authorization predicate. Any state used as the source or destination of a transition must be reachable from the starting state by some sequence of naive transitions (neglecting their guards and authorization). *Quartz* ensures that each transition is labeled with a unique name and that no transition parameter shadows the machine's fields due to a name collision. All guards and authorization predicates must refer to well-defined fields or transition parameters, and all assignments within a transition body must have some effect, i.e., no self-assignments are allowed.

5.2 Operational Semantics

Next, we give a formal presentation of the operational semantics of a *Quartz* contract. In all evaluation rules shown below, $\rightarrow$ indicates small-step evaluation and $\Downarrow$ indicates big-step evaluation. $\Downarrow_\sigma$ more specifically denotes big-step evaluation with σ as the initial environment. The $\circ$ symbol denotes concatenation, e.g., $m \circ M$ signifies the element m prepended to the sequence M.

Quartz centers around two main forms of evaluation, each tracked by a different element of a state machine's status. First, there is evaluation of a transition's authorization predicate, tracked by $S.\alpha$. Second, there is evaluation of the statements within a transition's body for their side effects, tracked by $S.s$. Only one of these two forms of evaluation may be in progress at a time. When $S.\alpha \neq \square$, meaning an authorization predicate is being evaluated, we must have $S.s = \top$, and when $S.s \neq \top$ we must have $S.\alpha = \square$.

Transition Authorization

When a new message m is dequeued, the machine's status is updated to indicate that the message's sender, i, approves of the transition's execution. *Quartz* then begins evaluation of the relevant transition's authorization predicate, $t.a$, as shown below in the UpdateAuth evaluation rule.

$$\frac{\begin{array}{c} S = \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \square,\, C\rangle \qquad m = \langle i,\, t,\, V\rangle \\ q = t.\text{src} \qquad \sigma' = \sigma \cup V \\ t.g \Downarrow_{\sigma'} \texttt{true} \qquad \alpha_{t,V} = \alpha[\langle t,\, V\rangle] \end{array}}{S \to \langle q,\, \sigma',\, [\langle t,\, V\rangle \mapsto \alpha_{t,V} \cup \{i\}]\alpha,\, m \circ M,\, \top,\, t.a,\, C\rangle} \text{ (UpdateAuth)} \tag{5.1}$$

Terms in the authorization predicate come in three forms:

- A reference to an identity variable i, meaning i must approve of the transition
- An expression `all(`I`)`, where $I \in F$ is a field of type `Sequence[Identity]`, meaning all $i \in I$ must approve the transition
- An expression `any(`I`)`, where $I \in F$ is again of type `Sequence[Identity]`, meaning any $i \in I$ must approve the transition

These terms are evaluated as shown in the rules below.

$$\frac{\begin{array}{c} S = \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, i,\, C\rangle \\ m = \langle j,\, t,\, V\rangle \qquad i \in \alpha[\langle t,\, V\rangle] \end{array}}{S \to \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \texttt{true},\, C\rangle} \text{ (AuthSingleTrue)} \tag{5.2}$$

$$\frac{\begin{array}{c} S\langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \texttt{any}(I),\, C\rangle \\ m = \langle j,\, t,\, V\rangle \qquad \exists\, i \in I : i \in \alpha[\langle t,\, V\rangle] \end{array}}{S \to \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \texttt{true},\, C\rangle} \text{ (AuthAnyTrue)} \tag{5.3}$$

$$\frac{\begin{array}{c} S = \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \texttt{all}(I),\, C\rangle \\ m = \langle j,\, t,\, V\rangle \qquad \forall\, i \in I : i \in \alpha[\langle t,\, V\rangle] \end{array}}{S \to \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \texttt{true},\, C\rangle} \text{ (AuthAllTrue)} \tag{5.4}$$

Each of these rules also has a complement:

$$\frac{\begin{array}{c} S = \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, i,\, C\rangle \\ m = \langle j,\, t,\, V\rangle \qquad i \notin \alpha[\langle t,\, V\rangle] \end{array}}{S \to \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \texttt{false},\, C\rangle} \text{ (AuthSingleFalse)} \tag{5.5}$$

$$\frac{\begin{array}{c} S\langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \texttt{any}(I),\, C\rangle \\ m = \langle j,\, t,\, V\rangle \qquad \forall\, i \in I : i \notin \alpha[\langle t,\, V\rangle] \end{array}}{S \to \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \texttt{false},\, C\rangle} \text{ (AuthAnyFalse)} \tag{5.6}$$

$$\frac{\begin{array}{c} S = \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \texttt{all}(I),\, C\rangle \\ m = \langle j,\, t,\, V\rangle \qquad \exists\, i \in I : i \notin \alpha[\langle t,\, V\rangle] \end{array}}{S \to \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \texttt{false},\, C\rangle} \text{ (AuthAllFalse)} \tag{5.7}$$

Authorization predicates may also contain logical `AND` and `OR` operators applied to the term forms presented above. Their evaluation is defined by the following rules:

$$\frac{S = \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, a_1 \wedge a_2 \rangle \qquad \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, a_1 \rangle \rightarrow \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, a_1' \rangle}{S \rightarrow \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, a_1' \wedge a_2 \rangle} \text{ (AuthEvalAnd)} \tag{5.8}$$

$$\frac{S = \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, a_1 \vee a_2 \rangle \qquad \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, a_1 \rangle \rightarrow \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, a_1' \rangle}{S \rightarrow \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, a_1' \vee a_2 \rangle} \text{ (AuthEvalOr)} \tag{5.9}$$

$$\frac{S = \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, \texttt{false} \vee a_2 \rangle}{S = \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, a_2 \rangle} \text{ (AuthOrFalse)} \tag{5.10}$$

$$\frac{S = \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, \texttt{true} \vee a_2 \rangle}{S = \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, \texttt{true} \rangle} \text{ (AuthOrTrue)} \tag{5.11}$$

$$\frac{S = \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, \texttt{true} \wedge a_2 \rangle}{S \rightarrow \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, a_2 \rangle} \text{ (AuthAndTrue)} \tag{5.12}$$

$$\frac{S = \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, \texttt{false} \wedge a_2 \rangle}{S \rightarrow \langle q,\, \sigma,\, \alpha,\, M,\, \top,\, \texttt{false} \rangle} \text{ (AuthAndFalse)} \tag{5.13}$$

If $t.a$ ultimately evaluates to `true`, then evaluation proceeds to the transition's body. As shown in AuthSuccess, the current statement in the machine's status advances from $\top$ to b_1, the first element of $t.b$. If $t.a$ evaluates to `false`, *Quartz* discards message m and removes the transition's parameter assignments from the current context σ. If the machine's stack (explained in more detail below) is empty, it awaits the next incoming message, as shown in AuthFailure1. Otherwise, the parent transition is resumed, as shown in AuthFailure2. In either case, the sender's approval of the transition persists in $S.\alpha$, meaning it affects future evaluations of $t.a$. This is what allows approvals to accumulate over time.

$$\frac{S = \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \texttt{true},\, C \rangle \qquad m = \langle i,\, t,\, V \rangle \qquad t.B = [b_1,\, \ldots,\, b_n]}{S \rightarrow \langle q,\, \sigma,\, \alpha,\, m \circ M,\, b_1,\, \square,\, C \rangle} \text{ (AuthSuccess)} \tag{5.14}$$

$$\frac{S = \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \top,\, \texttt{false},\, \varepsilon \rangle \qquad m = \langle i,\, t,\, V \rangle}{S \rightarrow \langle q,\, \sigma - V,\, \alpha,\, M,\, \top,\, \square,\, \varepsilon \rangle} \text{ (AuthFailure1)} \tag{5.15}$$

$$\frac{S = \langle q,\, \sigma,\, \alpha,\, m \circ m' \circ M,\, \top,\, \texttt{false},\, c \circ C \rangle \qquad m = \langle i,\, t,\, V \rangle \qquad m' = \langle i',\, t',\, V' \rangle}{S \rightarrow \langle q,\, (\sigma - V) \cup V',\, \alpha,\, m' \circ M,\, c,\, \square,\, C \rangle} \text{ (AuthFailure2)} \tag{5.16}$$

To summarize, these evaluation rules mean that any message invoking a particular transition is interpreted as an approval of the transition's execution by the sender. An approval

is specific to a set of arguments for the transition. A *Quartz* state machine tracks previous approvals by both transitions and parameter assignments in $S.\alpha$. If the transition's authorization predicate is not satisfied by the current set of approvals, it is effectively deferred until this is the case. This scheme also relies on $S.\alpha$ being properly reset once a transition completes, preventing the transition from being repeated an arbitrary number of times when approval was only given for one execution.

Transition Execution

A transition's body consists of a sequence of statements, evaluated in order when the transition executes. For convenience in the definitions below, define the function *next* as follows:

$$\text{next}(b_i) = \begin{cases} b_{i+1} & \text{if } i < n \\ \bot & \text{otherwise} \end{cases} \tag{5.17}$$

where a transition body consists of the statements $[b_1, b_2, \ldots, b_n]$.

We then have the following evaluation rules for *Quartz*'s relatively small collection of statement forms:

$$\frac{\begin{array}{c} S = \langle q, \sigma, \alpha, M, s, \square, C\rangle \\ s = \texttt{x = v} \qquad \texttt{x} \Downarrow_\sigma x' \qquad \texttt{v} \Downarrow_\sigma v' \end{array}}{S \to \langle q, [x' \mapsto v']\sigma, \alpha, M, \text{next}(s), \square, C\rangle} \text{ (EvalAssign)} \tag{5.18}$$

$$\frac{\begin{array}{c} S = \langle q, \sigma, \alpha, M, s, \square, C\rangle \qquad s = \texttt{append v to x} \\ \texttt{v} \Downarrow_\sigma v' \qquad \texttt{x} \Downarrow_\sigma x' \qquad \sigma[x'] = [x_1, \ldots, x_n] \end{array}}{S \to \langle q, [x' \mapsto [x_1, \ldots, x_n, v']\sigma, \alpha, M, \text{next}(s), \square, C\rangle} \text{ (EvalAppend)} \tag{5.19}$$

$$\frac{\begin{array}{c} S = \langle q, \sigma, \alpha, M, s, \square, C\rangle \\ s = \texttt{clear x} \qquad \texttt{x} \Downarrow_\sigma x' \end{array}}{S \to \langle q, [x' \mapsto \varepsilon]\sigma, \alpha, M, \text{next}(s), \square, C\rangle} \text{ (EvalClear)} \tag{5.20}$$

$$\frac{\begin{array}{c} S = \langle q, \sigma, \alpha, M, s, \square, C\rangle \\ s = \texttt{if (x)} \ \{s_1, \ldots, s_n\} \qquad \texttt{x} \Downarrow_\sigma \texttt{true} \end{array}}{S \to \langle q, \sigma, \alpha, M, s_1, \square, C\rangle} \text{ (EvalIfTrue)} \tag{5.21}$$

$$\frac{\begin{array}{c} S = \langle q, \sigma, \alpha, M, s, \square, C\rangle \\ s = \texttt{if (x)} \ \{s_1, \ldots, s_n\} \qquad \texttt{x} \Downarrow_\sigma \texttt{false} \end{array}}{S \to \langle q, \sigma, \alpha, M, \text{next}(s), \square, C\rangle} \text{ (EvalIfFalse)} \tag{5.22}$$

$$\frac{\begin{array}{c} S = \langle q, \sigma, \alpha, M, s, \square, C\rangle \\ s = \texttt{if (x)} \ \{s_1, \ldots, s_n\} \ \texttt{else} \ \{s_1, \ldots, s_n\} \\ \texttt{x} \Downarrow_\sigma \texttt{true} \end{array}}{S \to \langle q, \sigma, \alpha, M, s_1, \square, C\rangle} \text{ (EvalIfElseTrue)} \tag{5.23}$$

$$\frac{\begin{array}{c} S = \langle q,\, \sigma,\, \alpha,\, M,\, s,\, \square,\, C\rangle \\ s = \texttt{if (x) } \{s_1,\, \ldots,\, s_n\} \texttt{ else } \{s'_1,\, \ldots,\, s'_n\} \\ \texttt{x} \Downarrow_\sigma \texttt{false} \end{array}}{S \to \langle q,\, \sigma,\, \alpha,\, M,\, s'_1,\, \square,\, C\rangle} \text{ (EvalIfElseFalse)} \tag{5.24}$$

Quartz's `send` primitive has more involved evaluation rules than the other statement forms. It involves communication with an external contract and manipulation of the state machine's stack C. Recall that each element $c \in C$ is a pair $\langle \sigma,\, s\rangle$, where σ is the current transition's context, i.e. its name-value bindings, and s is the statement to execute when control returns to the transition. Note that we don't need to store the identify of the transition itself, as the message invoking the transition is retained on the state machine's queue until the transition is complete.

The key issue around a `send` is that it may cause the sending state machine to cede control to the recipient. Therefore, its evaluation non-deterministically leads to one of three possible outcomes, depending on the actions of the recipient. The simplest possible evaluation for `send` occurs when the receiving contract is credited with the specified amount, takes no observable action in response, and control immediately resumes within the same transition:

$$\frac{\begin{array}{c} S = \langle q,\, \sigma,\, \alpha,\, M,\, s,\, \square,\, C\rangle \qquad s = \texttt{send a to i} \\ \texttt{a} \Downarrow_\sigma a' \qquad a' \leq \sigma[\texttt{balance}] \qquad b = \sigma[\texttt{balance}] - a' \end{array}}{S \to \langle q,\, [\texttt{balance} \mapsto b]\sigma,\, \alpha,\, M,\, \text{next}(s),\, \square,\, C\rangle} \text{ (SendSuccess)} \tag{5.25}$$

It is also possible that the target throws an exception, reverting execution of the current transition:

$$\frac{S = \langle q,\, \sigma,\, \alpha,\, M,\, s,\, \square,\, C\rangle \qquad s = \texttt{send a to i}}{S \to \textsc{Error}} \text{ (SendError)} \tag{5.26}$$

Finally, the recipient may react by sending an arbitrary message to the sender, deferring completion of the current transition and prompting re-entrant execution of a separate transition that executes immediately.

$$\frac{\begin{array}{c} S = \langle q,\, \sigma,\, \alpha,\, m \circ M,\, s,\, \square,\, C\rangle \qquad m = \langle i,\, t,\, V\rangle \\ s = \texttt{send a to j} \qquad \texttt{a} \Downarrow_\sigma a' \qquad \texttt{j} \Downarrow_\sigma j' \\ a' \leq \sigma[\texttt{balance}] \qquad \sigma' = [\texttt{balance} \mapsto \texttt{balance} - a']\sigma \\ m' = \langle j',\, t' \in T,\, V'\rangle \end{array}}{S \to \langle q,\, (\sigma' - V) \cup V',\, \alpha,\, m' \circ m \circ M,\, \top,\, \square,\, \text{next}(s) \circ C\rangle} \text{ (SendReenter)} \tag{5.27}$$

Note that this last evaluation rule is the only way a state machine's stack may grow.

When the end of the transition's body, $\bot$, is reached, *Quartz* consults the state machine's stack for a record of an in-progress parent transition, consisting of the next statement c to execute. If such a record exists as shown in FinishTransition1, control returns to the parent transition. Otherwise, the state machine moves to the implicit $\top$ statement as

shown in FinishTransition2. In both cases, the message at the head of the queue M is finally removed and the current environment σ is stripped of the transition's parameter assignments. Prior approvals of the transition's execution are removed from α to prevent repeated invocations without new approval.

$$\frac{\begin{array}{c} S = \langle q,\, \sigma,\, \alpha,\, m \circ m' \circ M,\, \bot,\, \square,\, c \circ C \rangle \\ m = \langle i,\, t,\, V \rangle \qquad m' = \langle i',\, t',\, V' \rangle \end{array}}{S \to \langle t.\text{dst},\, (\sigma - V) \cup V',\, [\langle t,\, V \rangle \mapsto \emptyset]\alpha,\, m' \circ M,\, c,\, \square,\, C \rangle} \text{ (FinishTransition1)} \tag{5.28}$$

$$\frac{S = \langle q,\, \sigma,\, \alpha,\, m \circ M,\, \bot,\, \square,\, \varepsilon \rangle \qquad m = \langle i,\, t,\, V \rangle}{S \to \langle t.\text{dst},\, \sigma,\, [\langle t,\, V \rangle \mapsto \emptyset]\alpha,\, M,\, \top,\, \square, \varepsilon \rangle} \text{ (FinishTransition2)} \tag{5.29}$$

Expressions

Quartz supports a typical set of arithmetic and logical expression forms, evaluated in the expected fashion. For example, Boolean operators follow rules analogous to those given in Equations 5.8 through 5.13 above. All arithmetic and logical operations follow the standard order of precedence (in particular, arithmetic operators bind more tightly than logical operators).

Quartz supports assignment to and reading from values of three different forms:

- A reference to a variable by name, e.g., `x`
- A reference to an element within a `Map`, e.g., `x[y]`
- A reference to a field of a `Struct`, e.g., `x.y`

References to these forms are evaluated as expected. For example:

$$\frac{a[b] \to x}{a[b][c] \to x[c]} \text{ (EvalMapRef)} \tag{5.30}$$

$$\frac{a.b \to x}{a.b.c \to x.c} \text{ (EvalStructRef)} \tag{5.31}$$

Quartz also allows computation of the size of lists and the Keccak-256 hash of any expression:

$$\frac{x \Downarrow [x_1,\, x_2,\, \ldots,\, x_n]}{\texttt{size}(x) \to n} \text{ (EvalSize)} \tag{5.32}$$

$$\frac{x \Downarrow x'}{\texttt{hash}(x) \to \text{keccak256}(x')} \text{ (EvalHash)} \tag{5.33}$$

5.3 Type System

Quartz performs simple type checking to eliminate potential errors in the later stages of the translation process. Each of the DSL's types are listed in Table 5.1. Many of these types are the same as one would expect in a traditional programming language, but a few are specifically intended for contracts. They are either specific to the distributed ledger setting (e.g., `Identity`) or are motivated by recurring concerns in smart contract implementation such as time-based actions and authorization (e.g., `Timestamp`, `Timespan`).

Type	Description
`Bool`	Boolean values
`Int`	Integer values
`Uint`	Non-negative integer values
`String`	Character sequences
`Identity`	A unique identifier for a user or external contract
`Timestamp`	An instant in time
`Timestamp`	A duration of time
`Sequence[T]`	An ordered collection of elements of type `T`
`Map[K,V]`	Associative array with keys of type `K` and values of type `V`
`Struct`	A user-defined collection of fields
`HashValue`$[\texttt{T}_1, \ldots, \texttt{T}_n]$	A hash of values of types $\texttt{T}_1$ through $\texttt{T}_n$ concatenated together

Table 5.1: Types in the *Quartz* DSL

In the typing rules given below, Γ indicates the current typing context, i.e. the assumed types of all variables currently in scope. $\Gamma \vdash x : T$ denotes that the expression x has type T under context Γ. Context construction is simplified by *Quartz*'s design — it requires all fields and transition parameters to be declared with explicit type annotations and does not allow binding new variables within transition bodies with something like a `let` form. For example, say we have a state machine with the set of fields F and set of transitions T. Then when type checking the body of some transition $t \in T$ with the set of parameters P we have:

$$\Gamma = \{f.\text{name} \mapsto f.\text{ty} : f \in F\} \cup \{p.\text{name} \mapsto p.\text{ty} : p \in P\}\,.$$

These two sets are guaranteed to be disjoint as *Quartz* validates that no transition parameters shadow fields through a name collision.

The `Bool`, `Int`, `Uint`, and `String` types each have literals of the expected forms, familiar to any user of a typical programming language. `Timespan` literals are created from a positive integer followed by a unit keyword, such as `24 hours` or `7 days`. The only literals of types `Identity` and `Timestamp` are the `sender` and `now` keywords, respectively.

A `HashValue` type is parameterized by a sequence of types — the types of the elements concatenated together to produce the hash's preimage.

$$\frac{\Gamma \vdash x_1 : T_1 \qquad \Gamma \vdash x_2 : T_2 \qquad \cdots \qquad \Gamma \vdash x_n : T_n}{\Gamma \vdash \texttt{hash}(x_1,\, x_2,\, \ldots,\, x_n) : \texttt{HashValue[}T_1,\, T_2,\, \ldots,\, T_n\texttt{]}} \text{ (T-Hash)} \tag{5.34}$$

A `HashValue` only supports equality checks with instances of the same type. This encourages the use of hashing for purposes like commitment schemes and capability-based access control while discouraging the use of hashing as a pseudo-random number generator, a practice that has introduced vulnerabilities in past contracts [8].

Maps must be accessed using a proper instance of the key type and produce an instance of the value type, as expected:

$$\frac{\Gamma \vdash m : \texttt{Map[}K,\, V\texttt{]} \qquad \Gamma \vdash x : K}{\Gamma \vdash m\texttt{[}x\texttt{]} : V} \text{ (T-Map)} \tag{5.35}$$

A `Struct` is formally a collection of fields, each with a name and type. Therefore, type checking a reference to a struct element involves extending the context Γ and then applying the usual typing rules.

$$\frac{\Gamma \cup \{f.\text{name} \mapsto f.\text{ty} : f \in s\} \vdash x : T}{\Gamma \vdash s\,.\,x : T} \text{ (T-Struct)} \tag{5.36}$$

Quartz type checks arithmetic expressions to ensure compatibility of their operands. Variables of types `Int` and `Uint` cannot be freely mixed as they are in other programming languages. There are two reasons for this. First, `Uint` variables are most frequently used to track ownership of virtual tokens or other assets, and, in every use case we surveyed, their interaction with an `Int` variable was the result of a programming error rather than a requirement motivated by the domain. *Quartz*'s type system enforces this separation. More practically, smart contract implementation languages like Solidity also encourage this distinction, e.g., by requiring an explicit conversion so that the two operands are of the same type, and adding this constraint to *Quartz* simplifies the task of correct code generation.

There are, however, many cases where an `Int` variable needs to interoperate with `Int` or `Uint` literals, e.g., to scale a value by a constant factor. *Quartz* thus breaks each of `Int` and `Uint` into two subtypes, one for constant literals and another for variables. With multiplication, for example, we then have the following typing rules.

$$\frac{\Gamma \vdash x : \texttt{IntVar} \qquad \Gamma \vdash y : \texttt{IntVar}}{\Gamma \vdash x * y : \texttt{IntVar}} \text{ (T-Mul1)} \tag{5.37}$$

$$\frac{\Gamma \vdash x : \texttt{IntConst} \qquad \Gamma \vdash y : \texttt{IntVar}}{\Gamma \vdash x * y : \texttt{IntVar}} \text{ (T-Mul2)} \tag{5.38}$$

$$\frac{\Gamma \vdash x : \texttt{UintConst} \qquad \Gamma \vdash y : \texttt{IntVar}}{\Gamma \vdash x * y : \texttt{IntVar}} \text{ (T-Mul3)} \tag{5.39}$$

$$\frac{\Gamma \vdash x : \texttt{UintVar} \qquad \Gamma \vdash y : \texttt{UintVar}}{\Gamma \vdash x * y : \texttt{UintVar}} \text{ (T-Mul4)} \tag{5.40}$$

$$\frac{\Gamma \vdash x : \texttt{UintConst} \qquad \Gamma \vdash y : \texttt{UintVar}}{\Gamma \vdash x * y : \texttt{UintVar}} \text{ (T-Mul5)} \tag{5.41}$$

This omits several rules for brevity, such as operations on `IntConst` and `Uintconst` which are typed as expected as well as the analogues of rules 5.38, 5.39, and 5.41 with the types of x and y swapped. Note that `IntVar` and `UintVar` cannot mix and that an expression involving an `IntVar` is well typed when the other operand is any integer constant, while `UintVar` may only be multiplied by `UintConst` or `UintVar`.

Quartz also supports arithmetic on `Timestamp` and `Timespan` instances. `Timespan`s may be added to or subtracted from one another:

$$\frac{\Gamma \vdash x : \texttt{Timespan} \qquad \Gamma \vdash y : \texttt{Timespan}}{\Gamma \vdash x + y : \texttt{Timespan}} \text{ (T-TimespanAdd)} \tag{5.42}$$

$$\frac{\Gamma \vdash x : \texttt{Timespan} \qquad \Gamma \vdash y : \texttt{Timespan}}{\Gamma \vdash x - y : \texttt{Timespan}} \text{ (T-TimespanAdd)} \tag{5.43}$$

A `Timespan` may be added to or subtracted from a `Timestamp`:

$$\frac{\Gamma \vdash x : \texttt{Timestamp} \qquad \Gamma \vdash y : \texttt{Timespan}}{\Gamma \vdash x + y : \texttt{Timestamp}} \text{ (T-TimestampAdd)} \tag{5.44}$$

$$\frac{\Gamma \vdash x : \texttt{Timestamp} \qquad \Gamma \vdash y : \texttt{Timespan}}{\Gamma \vdash x - y : \texttt{Timestamp}} \text{ (T-TimestampSub)} \tag{5.45}$$

Finally, `Timestamp`s may be scaled by unsigned integers:

$$\frac{\Gamma \vdash x : \texttt{Uint} \qquad \Gamma \vdash y : \texttt{Timespan}}{\Gamma \vdash x * y : \texttt{Timespan}} \text{ (T-ScaleTimespan)} \tag{5.46}$$

Finally, logical expressions involving Boolean operators are predominantly type checked as expected. Operands for `AND` and `OR` must both be of Boolean type, while operands for comparisons must be compatible, e.g., a `Timestamp` may only be compared with another `Timestamp`. Unordered types, such as `Identity` and `HashValue`, may be checked for direct equality but may not be used in comparisons.

5.4 Summary

This chapter has precisely and formally laid out the structure and execution of a *Quartz*-defined state machine. The definitions presented here are central to our approach. Chapter 4 detailed the *Quartz* language as a means of specifying a contract within a state machine structure, while the translation paths that will be described in Chapter 6 and Chapter 7 must preserve the original state machine's semantics in the alternative forms of TLA$^+$ specification and Solidity implementation, respectively. In both translation paths, subtleties arise

in replicating certain *Quartz* evaluation rules in alternative source languages. For example, Chapter 6 details the importance of faithfully capturing issues around reentrancy and exception handling, while Chapter 7 describes challenges around state transition authorization.

Chapter 6

Translation to TLA+ and Validation

In this chapter, we describe the elements of our approach concerning contract validation. *Quartz* translates a contract state machine definition to a formal specification expressed in TLA$^+$ [63]. This specification captures both the contract's logic as well as the semantics of the execution environment, namely an Ethereum-based distributed ledger. *Quartz* then provides this specification to TLC, an explicit-state model checker for TLA$^+$ that enumerates and searches possible execution traces for violations of user-provided properties, i.e., the invariants written by the contract author in her description. This approach has the advantage of being fully automated, with no intervention needed from the user. It does, however, raise the challenge of bounding the execution search space so that model checking terminates.

6.1 Why Model Checking and TLA+?

We chose bounded model checking as *Quartz*'s core verification technique because it does not require significant intervention from the end user, i.e., the contract author. Although the author must write the invariants she would like to have verified, *Quartz* fully automates the more difficult task of writing a formal specification of the contract's behavior and its execution environment. This is the required input for model checking and has historically served as the largest barrier to its effective use. Model checking also offers the benefit of providing immediately useful feedback to the user as its output — an execution trace that produces a violation of one or more of the desired properties. This feedback helps guide a contract author in making iterative refinements to her state machine to address these violations.

TLA$^+$ serves as *Quartz*'s target specification language and its verification backend. TLA$^+$ and its model checker, TLC, are relatively mature, well-documented, and have been successfully applied in developing and testing significant systems [72]. More modern model checkers have since emerged, but they tend to be inherently tied to the semantics of particular implementation languages such as C [48] or operate at the low level of bytecode [44]. The flexibility of TLA$^+$'s specification language simplifies *Quartz*'s task of generating a formal

contract specification. This becomes especially important when describing the execution semantics of Solidity, which have important differences from the semantics of traditional programming languages. Moreover, there are ongoing efforts to modernize verification in TLA$^+$, such as symbolic model checking with SMT solvers [61], that *Quartz* may be able to use in the future.

6.2 Specification Generation

Quartz's specification generator targets PlusCal, an intermediate language built on top of the original TLA$^+$ specification language. An example body of PlusCal, for an auction contract, is presented and discussed below in Section 6.4. PlusCal offers several features that make it a more natural target than TLA$^+$ itself, such as procedures to model state transitions and conditionals to model transition guards. Translating *Quartz* data types and transitions into PlusCal is straightforward, but modeling *Quartz*'s authorization semantics and the blockchain execution environment is more challenging. The PlusCal generated by *Quartz* is translated into TLA$^+$ with off-the-shelf tools.

Data Types

Every data type in *Quartz* maps to a counterpart in PlusCal. Many have direct equivalents such as `Int`s and `Map`s. *Quartz* defines the domain of the `Identity` type as a fixed set of symbolic constants, with a user-configurable size. The translation of a `HashValue` is more subtle. *Quartz* has no need to model hash functions in detail aside from the fact that they are assumed to be injective, nor does it require that `HashValue` instances are ordered. The output of `hash`$(x_1, \ldots, x_n)$ is simply modeled as a PlusCal tuple $\langle x_1, \ldots, x_n \rangle$, preserving injectivity and enabling equality checks among `HashValue` instances.

Transitions

Each transition defined in a *Quartz* contract's state machine is generated as a PlusCal procedure, with transition parameters naturally mapping to procedure parameters. An extra parameter is added to the PlusCal procedure to track the transition's invoking party. An auxiliary field is used to track the machine's current state. The procedure body begins with three conditional checks: one to ensure that the state machine is currently in the transition's designated starting state, a second to ensure that the transition's guard, if defined, is satisfied, and a third to ensure that the transition's authorization predicate, if defined, is also satisfied. Finally, the statements forming the transition body are converted to PlusCal in the expected way, as PlusCal supports a standard collection of assignment, arithmetic, and logical operators.

Authorization

Quartz adds auxiliary fields to a contract's PlusCal specification to accurately model the authorization semantics detailed in evaluation rules 5.1 through 5.16 above. Note that an entity approves of the execution of a transition for a particular set of parameter values. For example, consider a transition T with input parameters of types $t_1, \ldots, t_n$ that includes a term of the form `all(I)` in its authorization predicate. *Quartz* generates a PlusCal function (associative array) F of type $t_1 \times \cdots \times t_n \times \texttt{Identity} \rightarrow \texttt{Boolean}$. Then, `all(I)` is evaluated in PlusCal, which has native support for quantifiers, as $\forall\, i \in I : F(p_1, \ldots, p_n, i)$, where p_i is the i^{th} transition argument. `any(I)` is evaluated as $\exists\, i \in I : F(p_1, \ldots, p_n, i)$, and a term of the form i is simply $F(p_1, \ldots, p_n, i)$. *Quartz* performs a static analysis of authorization predicates to generate the minimum number of auxiliary fields, only when a transition must be authorized through multiple approvals.

Modeling the Environment

Once an Ethereum contract is deployed to a blockchain, any of its transformations may be invoked at any time, by any user. For a *Quartz* contract, this means any of the state machine's transitions may be invoked at any time. The PlusCal model generated by *Quartz* is organized around a main invocation loop that simulates this environment. Each time through the loop, a transition t is non-deterministically selected for execution. Values for its input parameters $v_1, \ldots, v_n$ are similarly selected from their respective domains, including an identity i as the invoking party.

The second major challenge in modeling the Ethereum execution environment is capturing the behavior of sending tokens from one contract to another, i.e., the semantics of evaluation rules 5.25 throuh 5.27. As explained above, there are two primary means of exchanging tokens between one Ethereum contract and another: using Solidity's `transfer` primitive or using the `call` primitive. Both yield control to the destination contract. `call` is more flexible in that it allows the recipient to execute arbitrary code, but this may include a reentrant invocation of the sending contract. `transfer` restricts execution, by limiting the amount of gas available to the receiver in its execution, but propagates any exceptions thrown by the receiver back to the sender, which may block forward progress.

In *Quartz*, the user may specify the use of `transfer` or `call` as a configuration parameter. The system is capable of modeling either primitive's behavior. The generated PlusCal model deducts from the `balance` field and then makes a non-deterministic choice to model the recipient's response. When modeling `transfer`, the recipient either does nothing, indicating a routine token transfer, or throws an exception. When modeling `call`, the recipient either does nothing, meaning any code executed by the recipient had no consequence for the sender, or it non-deterministically selects some transition t of the sender's to invoke, modeling possible re-entrant execution.

The behavior of Ethereum's exceptions cannot be expressed in PlusCal. Instead, *Quartz* generates an initial PlusCal specification, invokes the PlusCal translator to produce TLA$^+$,

and modifies this code directly. The final TLA$^+$ generated by *Quartz* for model checking formalizes unwinding of the stack upon an exception: the contract's fields are reverted to their state before the current call chain, and execution jumps to the main invocation loop to begin a fresh transition.

6.3 Bounding the Search Space

The TLA$^+$ specifications generated by *Quartz*, as described so far, induce an infinite execution space. We must apply bounds to this search space to ensure that model checking terminates. *Quartz* allows the user to specify these bounds when validating a contract. Setting constraints on the search process itself, e.g., maximum depth traversed in the state space, requires an understanding of the internals of model checking. Instead, *Quartz* exposes parameters that have direct significance to the user. All parameters convey aspects of the contract's execution or the domain of data types:

- Minimum and maximum integer values
- The number of unique entities that may interact with the contract
- The maximum call depth reached during transition execution
- The maximum number of iterations of the main transition invocation loop

The model checker is essentially exploring all possible sequences of state machine transitions. The first and second parameters above control the branching factor of the search space. They restrict the number of unique values that are tested when selecting possible transition parameter values. The second parameter restricts the number of reentrant calls that may be made against the contract stemming from a single root invocation. Finally, the third parameter restricts the total length of any sequence that is searched.

A typical workflow when using *Quartz* is to begin with a well-constrained search space. Once the system fails to identify violations of the specified invariants in the current space, these constraints are iteratively loosened to explore increasingly large search spaces. Many contract vulnerabilities, including the examples discussed below in Section 6.5, are surfaced even when searching a relatively small space, which means that model checking terminates and provides feedback to the user quickly. The contract author may then also trade confidence in *Quartz*'s validation for longer model checking wait times.

6.4 Example

Figure 6.1 shows portions of the PlusCal specification generated by *Quartz* for the `Auction` case study, with small simplifications for clarity. The original *Quartz* code is listed above in Figure 4.4. Figure 6.1a shows a sender and `duration` parameter chosen nondeterministically before the state machine's initial transition is invoked on lines 2–4. Lines 6–20 form

the main invocation loop. Each iteration through the loop adds a new transition to the ongoing sequence. The transition itself, its sender, and its parameter values are all chosen nondeterministically, meaning the model checker explores all possible selections.

Figure 6.1b shows the PlusCal description of the `submitBid` transition. The conditional on lines 3–5 verifies that the state machine is in the proper starting state, while lines 6–8 are a translation of the transition's guard from the original *Quartz* description. Finally, lines 10–14 are a translation of the transition's body.

6.5 Model Checking Results

Here, we describe experiences model checking two contracts using *Quartz*. For both contracts, *Quartz* helps to surface non-obvious bugs that could easily be overlooked during contract development. Below, we report the time required for model checking to find invariant violations. These times were obtained on a workstation with an Intel i7-6700 CPU and 32 GiB of RAM running version 2.13 of the TLC model checker with 8 worker threads.

Model Checking an Auction Implementation

We introduced the `Auction` case study in detail in Section 4.3. Its state machine appears in Figure 4.3 and *Quartz* code appears in Figure 4.4. While it may appear perfectly logical, the contract as presented above features multiple security vulnerabilities, related to it distribution of repayments back to surpassed bidders and to the seller. These vulnerabilities are particularly insidious because they emerge from code that appears innocuous. They are good examples of how Ethereum's execution semantics differ from those of traditional software and can trip up contract authors.

To begin, consider the following invariant for the auction:

$$p_1 : \textbf{Closed} \Rightarrow \texttt{HighestBid} \geq \max(\texttt{submitBid.tokens})$$

This property takes advantage of a number of *Quartz*'s features for writing invariants. It states that if the auction reaches the **Closed** state, then the value of `HighestBid` should be greater than or equal to the maximum value ever assigned to the `tokens` parameter for the `submitBid` transition.

Recall that *Quartz* may generate a contract that uses either Ethereum's `transfer` construct or the `call` construct for dispensing currency. The choice is configurable by the user, and *Quartz* is fully capable of generating TLA$^+$ to model either. If `transfer` is used, the model checker finds a violation of p_1. This requires only 2 seconds to complete on our test system. The model checker presents the violation as an execution trace in the context of the contract's TLA$^+$ specification. This corresponds to an execution trace at the level of abstraction of the original state machine, which we present below and in future examples for clarity:

```
1  begin Main:
2    with sender ∈ IDENTITIES, duration ∈ 0..MAX_INT do
3        call initialize(sender, duration);
4    end with;
5
6  Loop:
7    either
8      with sender ∈ IDENTITIES, bid ∈ 0..MAX_INT do
9        call initialBid(sender, bid);
10     end with;
11   or
12     with sender ∈ IDENTITIES, bid ∈ 0..MAX_INT do
13       call submitBid(sender, bid);
14   or
15   ⋮
16   or
17     with sender ∈ IDENTITIES do
18       call redeem(sender);
19     end with;
20   end either;
```

(a) Main Invocation Loop

```
1  procedure submitBid(sender, bid)
2  begin submitBid:
3    if currentState ≠ OPEN then
4      return;
5    end if;
6    if bid ≤ HighestBid ∨ currentTime > Deadline then
7      return;
8    end if;
9
10   balance := balance + bid;
11   call send(HighestBidder, HighestBid);
12   HighestBid := bid;
13   HighestBidder := sender;
14   return;
15 end procedure;
```

(b) `submitBid` Transition

Figure 6.1: PlusCal Code for an Auction Contract

1. Identity I_1 deploys a new auction contract. The auction enters the **Init** state.
2. I_2 submits an initial bid of 2 tokens and is recorded as the highest bidder. The auction enters the **Open** state.
3. I_3 submits a new bid of 4 tokens. The auction sends 2 tokens back to I_2 as a refund, since they are no longer the highest bidder.
4. I_2 reacts by throwing an exception. This propagates back to the auction contract (due to the use of `transfer`) and the current transition is aborted. I_3's bid is lost.
5. No additional bids are submitted before I_1 moves to close the auction and I_2 is declared the winner.

Here, I_2 is able to hijack the auction and prevent itself from being supplanted as the highest bidder, rigging the results of the auction.

p_1 is also violated if we use `call` rather than `transfer`, again because of an issue in refunding a previous bidder. TLC identified the following scenario leading to a violation in 6 seconds on our test system.

1. Identity I_1 deploys a new auction contract. The auction enters the **Init** state.
2. I_2 submits an initial bid of 2 tokens and is recorded as the highest bidder. The auction enters the **Open** state.
3. I_3 submits a new bid of 3 tokens. The auction sends 2 tokens to I_2 as a refund.
4. I_2 responds to the receipt of tokens by submitting a new bid of its own, with a value of 4 tokens, creating a reentrant invocation of the *submitBid* transition.
5. The auction accepts I_2's bid, sets `HighestBidder` to I_2 and `HighestBid` to 4.
6. Control returns to the parent transition, which has just completed its `send`. It updates `HighestBidder` to I_3 and `HighestBid` to 3, accounting for I_3's bid but overwriting I_2's bid.
7. No further bids arrive. The auction reaches the **Closed** state.

Here, we see that contract re-entrancy, an issue best known for its exploitation by malicious actors, can also lead to undesirable outcomes for well-intentioned actors. One could easily imagine a developer seeking to create a contract that submits a bid on her behalf in reaction to having just been displaced as an auction's winner, possibly to implement some bidding strategy, yet that would go awry in this implementation.

To address either of these bugs, the contract author could instead store a *Quartz* `Map[Identity, Uint]` tracking pending refunds that is updated when a newly winning bid is submitted. A previous bidder must invoke a separate transition to ask the contract to send her a refund,

decoupling this from bidding. This is a well-known design pattern in Solidity [35], although it is prone to its own re-entrancy issues, which *Quartz* can also identify through its verification. Say we modify *submitBid* accordingly and add the following transition to allow bidders to claim refunds once the seller has redeemed their winnings:

```
refund: redeemed -> redeemed
requires [ Balances[sender] > 0 ] {
    send Balances[sender] to sender
    Balances[sender] = 0
}
```

Consider the following new invariant for the auction:

$$p_2 : \texttt{balance} \geq 0$$

This states that the contract's balance cannot go negative, i.e., it cannot dispense more tokens than it receives. While this is impossible for a contract running on the Ethereum blockchain, negative contract balances are within the search space defined by *Quartz* for model checking because they can usefully indicate a contract's vulnerability to unbounded token withdrawals. Indeed, if *Quartz* is configured to model behavior of token sends using `call`, model checking finds a violation in 15 seconds. Again, the model checker's output presents an execution trace in terms of the contract's TLA$^+$ specification which can be abstracted into the following trace against the contract's original state machine:

1. Identity I_1 deploys a new auction contract.
2. I_2 submits an initial bid of 1 token. The auction enters the **Open** state with `balance = 1`.
3. I_3 submits a new bid of 4 tokens, hence `balance = 5`.
4. No subsequent bids are submitted before the deadline, and I_1 moves to close the auction. The auction enters the **Closed** state.
5. I_1 invokes the *redeem* transition, receiving its winnings. Now, `balance = 1` and the auction enters the **Redeemed** state.
6. I_2 invokes the *refund* transition and is sent the 1 token recorded in `Balances[`I_2`]`. Now, `balance = 0`.
7. In reaction to this receipt of tokens, I_2 makes a reentrant invocation of *refund*. `Balances[sender]` has not yet been updated in the parent transition, so the child transition's guard is satisfied.
8. Another `send` of 1 token to I_2 is attempted, and `balance = -1`, violating the invariant.

This execution trace illustrates the fundamental vulnerability behind the famous compromise of the `DAO` contract [26]. The usual advice to Solidity developers is to set a temporary variable to the amount of tokens to send, then subtract from the appropriate contract field *before* executing a `send` referencing the temporary variable. *Quartz* offers an alternative `sendAndConsume` construct that will generate such code.

Model Checking ERC-1540

Quartz is useful not just for identifying subtle consequences of a contract's execution semantics, but also for identifying more routine logic errors that occur during the development process. Unlike our auction contract, which we initially developed as a litmus test for *Quartz*'s ability to surface reentrancy and exception issues, we drafted an initial implementation of ERC-1540 after *Quartz* was fairly mature, chose an invariant to verify, and used *Quartz* to refine the contract.

`ERC-1540` is a proposed Ethereum standard interface for an asset management contract. Among other capabilities, it allows an owner to sell shares, issue dividends, or transfer control of the asset, all of which is tracked on the blockchain. Investors issue transactions against the contract to buy and sell shares. The *Quartz* implementation of `ERC-1540` is considerably more complex than the auction seen previously. It uses five states and 16 transitions. The portion of the state machine relevant for the following discussion is shown in Figure 6.2.

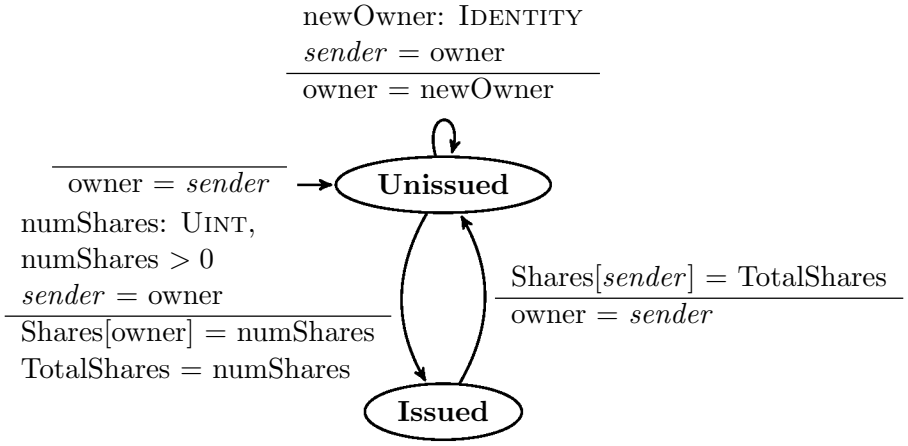


Figure 6.2: Part of a State Machine for ERC-1540

When the contract is initialized, its creator is recognized as the asset's owner. It begins in the **Unissued** state, meaning there are no outstanding shares. If this is the case, the owner is free to transfer possession of the asset to another party, as shown in the transition at the top of Figure 6.2. The owner may choose to move the asset to the **Issued** state by enacting the release of a fixed number of shares. The contract features additional transitions to exchange shares not shown in the figure. If any single party accumulates all outstanding

shares, they are allowed to declare themselves as the new owner and convert the asset back to the **Unissued** state.

We tested our initial version of the contract by verifying the following invariant.

$$p_3 : \text{sum}(\texttt{Shares}) = \texttt{TotalShares} \tag{6.1}$$

That is, we wanted to verify that all of the asset's shares are properly conserved as they change hands.

After generating a TLA$^+$ specification and running it through TLC, we were informed that many of the contract's states, Including `Issued` were not reachable. This was because we failed to properly initialize the contract's `owner` field in the state machine's initial transition. We were also able to uncover an error in our arithmetic when transferring shares.

More interestingly, we initially forgot to update the asset's shares when an entity converts it from **Issued** to **Unissued**. Initially, we simply reassigned the `owner` field to the transition's sender as shown in the figure, forgetting to zero out `TotalShares` and `Shares[Sender]`. This enables the following trace, identified by TLC when given a *Quartz*-generated contract spec and simplified for brevity:

1. I_1 deploys a new ERC-1540 contract and is recorded as the owner.
2. I_1 issues 2 shares for the asset, initially owning both of them. The asset enters the **Issued** state, and we have `TotalShares = 2` and `Shares[`I_1`] = 2`.
3. I_1 converts the asset back **Unissued**.
4. I_1 transfers ownership to I_2.
5. I_2 decides to issue 3 shares for the asset, all initially assigned to itself. In the transition body, we set `Shares[`I_2`] = 3` and `TotalShares = 3`. However, we still have `Shares[`I_1`] = 2`. Thus, sum(`Shares`) $\neq$ `TotalShares`.

The solution is to add two lines to the transition from **Issued** to **Unissued**, `Shares[sender] = 0` and `TotalShares = 0`, to properly reflect the fact that the asset no longer has shares. Because ERC-1540 is more complex than the auction above, model checking takes longer. This trace was produced after TLC ran for 4 minutes on our test workstation, while the arithmetic error in share transfers required 27 minutes to find. Running times of this magnitude are not atypical for model checking.

6.6 Related Work: Contract Analysis

Given the potentially high stakes of deploying smart contracts in production, it is no surprise that there is considerable interest in developing systematic and reliable methods of surfacing and patching contract vulnerabilities. There have been several one-off efforts to verify the properties of a specific contract [13, 71], but a number of more general tools have

been developed in both industry and academia to facilitate contract verification. Here, we summarize the research efforts behind these tools.

Tools for Manual Proof Construction

One class of contract analysis tool is meant to assist developers in constructing manual proofs of correctness. These follow in the tradition of computational proof assistants like Coq[1] or Isabelle[2] and often require familiarity with these systems and their use. An early version of the Scilla language [85] focused on translation from contract source code to a Coq formalization, which the contract's author can then use to construct proofs by hand, e.g. an inductive proof of a safety property over execution traces. Solidity* [12] is a tool that translates Solidity code into F*, a functional programming language and associated proof assistant developed by Microsoft. An author runs her contract through Solidity* and then manually builds proofs relying on the generated F* as a formal description of the contract's behavior. With either of these approaches, building a proof can be labor intensive. Moreover, each proof establishes a single property for a single contract. New proofs are needed when considering additional properties, the same property in the context of a different contract, or even the same property after modifying a contract's implementation.

Automated Tools

A second class of software tools require little intervention on the part of the user, but tend to be less powerful as a result. Two of the most common techniques are symbolic execution, usually of a contract's EVM bytecode [62, 65, 69, 74] and static analysis of the contract's AST [93]. These tools offer fully automated analysis, but they are largely restricted to identifying a fixed collection of generic vulnerabilities. Zeus [58] is a tool to convert a Solidity contract into LLVM bytecode for model checking. Securify [96] analyzes EVM bytecode to extract control flow and data flow graphs. It then verifies contract properties written in a datalog-based DSL, although these properties are intended to be generalized across many contracts rather than customized to a specific contract as in *Quartz*.

VerX [79] is a verification tool for Ethereum contracts that uses predicate abstraction and symbolic execution of EVM bytecode. Like *Quartz*, it offers automated verification of contract-specific properties, written in a variant of linear temporal logic. This makes VerX arguably the most comparable analysis tool to *Quartz*. However, VerX makes the assumption that all of the contracts it analyzes are *effectively callback free* [43], meaning it cannot model how something like a token `send` may disrupt a contract, either through reentrancy or exceptions. *Quartz* does not have this restriction, meaning it can find violations to properties p_1 and p_2 discussed in Section 6.5 that VerX cannot.

[1] `https://coq.inria.fr/`

[2] `https://isabelle.in.tum.de/`

6.7 Summary

In this Chapter, we made our proposed strategy of applying model checking to contract state machines concrete. We presented the details and associated challenges of translating a *Quartz* state machine to a TLA^+ specification. This translation proceeds without any intervention on the part of the contract author, sparing her the burden of formalizing logic and execution semantics that normally serves as an impediment to the use of model checking in practice. Additionally, we demonstrated the system's ability to identify contract flaws through two examples drawn from our set of case study contracts. We saw that *Quartz*-supported model checking can identify problems stemming from both logic errors and subtleties in contract control flow. In the following chapter, we will examine the second translation path, targeting Solidity.

Chapter 7

Translation to Solidity

Quartz features a second translation path, targeting Solidity, to produce contract implementations suitable for deployment on any Ethereum-based blockchain. We chose Solidity because it is the de facto standard contract programming language for Ethereum, currently the most widely used platform for smart contracts. It presents a paradigm for contract development that is very similar to object-oriented programming, meaning several constructs in the *Quartz* DSL have direct equivalents in Solidity.

Quartz does not target Ethereum Virtual Machine (EVM) bytecode directly for several reasons. The process of generating high-level Solidity code shares many similarities with the PlusCal generation described in Chapter 6, allowing *Quartz* to use similar logic in both translation paths. Solidity is more human readable than EVM bytecode, which means a contract author may easily inspect an auto-generated implementation if she chooses to. Finally, there is an ongoing effort within the Ethereum community to replace the original EVM with a new virtual machine based on WebAssembly.[1] By targeting Solidity, *Quartz* remains agnostic to this potential change.

7.1 Data Types

The data types for state machine fields and transition parameters in *Quartz* each correspond to a type in Solidity, shown in Table 7.1. Arithmetic and logical operations on these data types also have direct equivalents in Solidity, as does assignment. Every user-defined struct in the *Quartz* source is translated to a similar struct definition in Solidity, with each field of the struct translated to its analogous type in Solidity. Each of the *Quartz* contract's field definitions are similarly translated, including structs. The generated Solidity contract is augmented with a number of extra fields that are not explicitly represented in the original *Quartz* source code. Most importantly, an enumerated type is generated to represent the possible states of the contract, and a field of this type is used to track the contract's current

[1] `https://github.com/ewasm/design`

state. Finally, additional fields may be needed to track authorization information for state transitions, described in detail below.

***Quartz* Type**	**Solidity Type**
`Int`	`int`
`Timestamp`	`uint`
`Timespan`	`uint`
`Bool`	`bool`
`Identity`	`address`
`Sequence`	Array
`Map`	`mapping`
`Struct`	`struct`
`HashValue`	`bytes32`

Table 7.1: Mapping Between *Quartz* and Solidity Types

One data type of note is `Sequence`, translated as a Solidity array. As discussed below, `Sequence` types play an important role in transition authorization checks. Recall from Figure 4.2 that `Sequence` instances support operations like accessing individual elements, appending a new element to the end, and removing all entries. They also support element membership checks through the `in` and `not in` operators.

There is no Solidity data type that efficiently supports all of these operations. A Solidity array supports fast random access, appending, and clearing elements. However, checking for membership requires a potentially inefficient linear traversal of the array. A Solidity mapping, where the key type is the element type of the original *Quartz* sequence and the value type is a placeholder such as `bool`, makes membership checks very efficient and also supports easy addition and removal of elements. However, unlike associative data structures in traditional programming languages, a Solidity mapping does not enable enumeration of its elements, so there is no way to efficiently traverse the data structure. This has given rise to design patterns such as the iterable mapping library,[2] where a struct combining an array and mapping represents a data type supporting efficient membership checks and traversal.

This situation involves a number of tradeoffs. The iterable mapping pattern, with both an array and mapping, trades increased storage for reduced computation, as no loops are necessary for membership checks. At first glance, gas considerations also make this approach attractive. Recall that each operation in the execution of a transaction incurs a gas cost charged to the invoking party. Loops tend to be avoided in Solidity code where possible because it can be difficult to reason about their prospective gas costs. In the case of a membership check, the maximum possible gas costs are proportional to the size of the array in question. An iterable mapping avoids this issue entirely. However, increased storage

[2] `https://github.com/ethereum/dapp-bin/blob/master/library/iterable_mapping.sol`

also manifests as increased gas costs in two ways. The first, and more obvious, is that additional bookkeeping operations are required, which incur their own costs. The second comes from contract creation. A new contract is deployed to the blockchain in a transaction that initializes all of its fields and charges gas accordingly. More fields mean higher gas costs for this transaction.

The optimal Solidity representation of a `Sequence` type therefore depends on how large the `Sequence` is likely to become, which tells us how expensive a membership check implemented as a linear traversal is likely to become, and on how frequently membership checks are likely to occur, which tells us to what extent the higher upfront costs of initializing a structure like an iterable mapping can be recouped later on through cheaper membership checks. Obviously, *Quartz* cannot predict either of these factors from a contract's state machine description alone. Instead, it opts to use the simpler array-only representation with loops for membership checks. As we will show in Section 7.5, this induces reasonable gas costs under realistic workloads.

Analyzing Token Flows

More recent versions of Solidity require `address` variables that are the target of a `send` or `transfer` to be explicitly annotated with the `payable` keyword. One could imagine naively marking all fields and parameters of type `address`, as well as array, struct, and mapping instances with `address` elements, as `payable` in the generated Solidity code to circumvent this requirement. However, this would defeat the purpose of `payable` as a both a safety measure and a means of making Solidity code more explicit. Instead, *Quartz* precisely infers the occasions in generated code where `payable` is necessary. This is by a pre-generation AST traversal to identify all *Quartz* `Identity` instances that are targets of a `send` statement. Each target, or the array, struct, or mapping that contains the target, is then flagged for `payable` annotation during Solidity generation.

7.2 State Transition Logic

The majority of the generated Solidity code is devoted to implementing the state transition logic specified in the contract's original *Quartz* description. Each transition is implemented as a Solidity function. A high-level template for these functions is shown in Figure 7.1. Each parameter p_i of the original state transition corresponds to a Solidity parameter $\hat{p}_i$ of the appropriate translated data type. The function verifies that the state machine is in the appropriate starting state and that the transition's guard, if any, is satisfied. This uses Solidity's `require` keyword, which evaluates a boolean expression and reverts the current transaction if it is not satisfied.

Next, the Solidity function records approval for the associated state transition by the caller, i.e., the transition's invoking party. The transition's authorization predicate is evaluated, and execution of the transition stops here if the predicate is not satisfied. Authorization

Transition $T = \langle \text{name, src, dest, } g, a, P = \{p_1, \ldots, p_n\}, B = \{b_1, \ldots, b_n\} \rangle$

```
function name(p̂_1, ..., p̂_n) {
  require(current_state == src);
  require(g);
  Mark sender's approval of transition with parameters ⟨p̂_1, ..., p̂_n⟩;
  if (!a) {
    return;
  }
  b̂_1;
  ⋮
  b̂_n;
  current_state = dest;
}
```

Figure 7.1: Structure of a *Quartz*-Generated Solidity Function

record keeping and verification are described in more detail below. Note that we cannot use `require` to check the authorization predicate because we do not want to revert the update to authorization state on the previous line. This corresponds to the evaluation rule UpdateAuth in Chapter 5, i.e., rule 5.1.

The next portion of the function implementation is only reached if the transition's guard and authorization predicate are both satisfied. It consists of Solidity equivalents $\hat{b}_i$ for each statement b_i in the original *Quartz* transition's body. Assignments to variables are translated in the expected way while appending to and clearing sequences are accomplished with Solidity's `push` and `delete` primitives. A *Quartz* `send` operation is translated to either a Solidity `transfer` or `call` depending on the user's preference (recall that the two constructs take different approaches to transferring gas to the recipient contract). As explained in Chapter 6, *Quartz* can model the behavior of either construct.

7.3 Authorization

Recall that each transition in a *Quartz* contract may contain an authorization predicate that must be satisfied for the transition to fire. These predicates are written as conjunctions and disjunctions of terms that come in three possible forms: a stipulation that a specific `Identity` i must approve of the transition, a stipulation that any member of a `Sequence` of `Identity` elements must approve, or a stipulation that all members of such a `Sequence` must approve of the transition. More than one invocation of the same transition, by different

parties, may need to occur before the transition fires. Therefore, a generated Solidity contract must track these prior invocations to evaluate authorization predicates. This raises three main issues: determining where it is necessary to add state to the Solidity contract to track prior invocations, determining how to represent this state, and generating code to properly maintain this state over the lifetime of the contract.

Auxiliary Contract State

Quartz makes an effort to minimize the number of extra fields that are added to a generated Solidity contract for authorization tracking. It accomplishes this by examining the structure of each authorization predicate. Predicates composed of just a single term of the forms i or `any(`I`)` as well as disjunctions of these forms can be satisfied with a single invocation. Therefore, their evaluation does not rely on any persistence of state across multiple transition invocations. A predicate that contains a term of the form `all(`I`)` or a conjunction is only satisfied after the transition is invoked by multiple parties. In this case, *Quartz* adds a field to the Solidity contract for each term in the predicate which records whether or not that term is satisfied by invocations of the transition so far.

These fields are simplest when a transition has no parameters. For a reference directly to an identity or an `any` term, a simple flag is sufficient. For an `all` term, *Quartz* emulates a set data structure (which is not available in Solidity) with a mapping from addresses to booleans. Parameterized transitions require an additional layer of indirection. Recall that an entity's approval of a state transition is specific to a set of value assignments for the transition's parameters. *Quartz* generates Solidity code to concatenate and then hash parameter values when performing an authorization check. Identity references and `any` terms require a mapping from the `bytes32` type (the output of Solidity's built-in hash functions) to booleans, while `all` terms require a nested mapping from `bytes32` to address to boolean.

Maintaining Authorization State

Each Solidity function that implements a state transition with an authorization predicate contains logic to update and to check authorization state after the transition's guard and before its body. If the transition is parameterized, the parameter values are concatenated and hashed to enable parameter-specific updates and lookup. If the predicate is complex enough to require authorization fields, a conditional is generated for each of the predicate's terms to determine if the invoking party is relevant to the term and, if so, to update its corresponding authorization field. This involves a simple equality check for identity terms and a sequence membership check for `any` and `all` terms. Because `any` and `all` terms reference a *Quartz* sequence of identity elements, the efficiency of these membership checks depends on our choice of representation for sequence instances, as discussed above.

Next, a single conditional is used to evaluate the transition's authorization predicate and to return if it is not satisfied. Otherwise, execution reaches the transition's body statements and evaluates them in order. At the conclusion of the Solidity function, after the contract is

updated to reflect the fact that it is now in the transition's destination state, authorization state for this transition and combination of parameter values is cleared. This is to ensure that any future attempts to execute the same transition with the same parameters (e.g., i the state machine returns to the transitions' source state) requires reauthorization to proceed.

Membership checks for *Quartz*'s authorization checks (or evaluation of the `in` and `not in` operators) require loops. To simplify both the Solidity generation process and the code that it produces, these operations are implemented within contract-private functions. *Quartz* then only needs to generate calls to these functions within state transition functions and add implementations of these functions at the end of the contract. Solidity does not offer parametric polymorphism (like, e.g., Java's generics), so one membership check function is generated for each unique element type among the contract's sequence instances.

7.4 Example

Figure 7.1 shows the complete Solidity code generated from the Quartz state machine depicted in Figure 4.6. Fields are declared on lines 9–15. Only lines 9 and 10 correspond to fields from the original *Quartz* state machine, while lines 12–15 declare fields used to track transition authorization. The names of these authorization variables contain the name of the relevant transition, the name of the variable referenced by the corresponding authorization term, and a numerical index to avoid field name collisions where an authorization predicate refers to the same identity or identity sequence in multiple terms.

Lines 17, 23, 31, 40, 54, and 64 declare functions to emulate state transitions specified in the original *Quartz* source code. The contract's constructor is a special case and corresponds to the state machine's initial transition. The remaining functions follow the structure described above: a check that the emulated machine is in the proper starting state, authorization updates and checks, body statements, and finally the transition to the destination state. Note the differences in structure between the `addSigner` function, emulating a state transition with an authorization predicate consisting of a single `any` term, and the `declareOpen` function, which emulates a state transition with an authorization predicate requiring multiple invocations to be satisfied.

Finally, lines 77, 86, and 96 define internal utility functions that are used to perform membership checks and authorization checks. They perform linear traversals of array variables, checking if an element is present in an array or mapping instance. While not particularly complex, these functions serve to simplify the state transition functions defined above. The two `allApproved` functions are used to perform authorization checks for state transition functions without and with parameters, respectively.

```
1  pragma solidity >=0.5.7;
2
3  contract MultiSig {
4      enum State {
5          config,
6          open,
7          closed
8      }
9      address payable public Owner;
10     address[] public Signers;
11     State public __currentState;
12     mapping(address => bool) private
           __declareOpen_Signers_0Approved;
13     mapping(bytes32 => mapping(address => bool)) private
           __pay_Signers_0Approved;
14     mapping(address => bool) private __close_Signers_0Approved;
15     mapping(bytes32 => mapping(address => bool)) private
           __refund_Signers_0Approved;
16
17     constructor() public {
18         __currentState = State.config;
19         Owner = msg.sender;
20         Signers.push(msg.sender);
21     }
22
23     function addSigner(address newSigner) public {
24         require(__currentState == State.config);
25         if (!sequenceContains(Signers, msg.sender)) {
26             return;
27         }
28         Signers.push(newSigner);
29     }
30
31     function declareOpen() public {
32         require(__currentState == State.config);
33         require(sequenceContains(Signers, msg.sender));
34         __declareOpen_Signers_0Approved[msg.sender] = true;
35         if (!allApproved(Signers,
               __declareOpen_Signers_0Approved, 0)) {
36             return;
37         }
38         __currentState = State.open;
39     }
```

```
40      function pay(address payable recipient, int amount) public
           {
41          require(__currentState == State.open);
42          require(amount > 0);
43          require(sequenceContains(Signers, msg.sender));
44          __pay_Signers_0Approved[keccak256(abi.encodePacked(
               recipient, amount))][msg.sender] = true;
45          if (!allApproved(Signers, __pay_Signers_0Approved,
               keccak256(abi.encodePacked(recipient, amount)))) {
46              return;
47          }
48          recipient.transfer(uint(amount));
49          for (uint i = 0; i < Signers.length; i++) {
50              __pay_Signers_0Approved[keccak256(abi.encodePacked(
                   recipient, amount))][Signers[i]] = false;
51          }
52      }
53
54      function close() public {
55          require(__currentState == State.open);
56          require(sequenceContains(Signers, msg.sender));
57          __close_Signers_0Approved[msg.sender] = true;
58          if (!allApproved(Signers, __close_Signers_0Approved, 0)
               ) {
59              return;
60          }
61          __currentState = State.closed;
62      }
63
64      function refund(int amount) public {
65          require(__currentState == State.closed);
66          require(amount > 0);
67          require(sequenceContains(Signers, msg.sender));
68          __refund_Signers_0Approved[keccak256(abi.encodePacked(
               amount))][msg.sender] = true;
69          if (!allApproved(Signers, __refund_Signers_0Approved,
               keccak256(abi.encodePacked(amount)))) {
70              return;
71          }
72          Owner.transfer(uint(amount));
73          for (uint i = 0; i < Signers.length; i++) {
74              __refund_Signers_0Approved[keccak256(abi.
                   encodePacked(amount))][Signers[i]] = false;
75          }
76      }
```

```
77      function sequenceContains(address[] storage sequence,
           address element) private view returns (bool) {
78          for (uint i = 0; i < sequence.length; i++) {
79              if (sequence[i] == element) {
80                  return true;
81              }
82          }
83          return false;
84      }
85
86      function allApproved(address[] storage approvers, mapping(
           address => bool) storage approvals))
87                private view returns (bool) {
88          for (uint i = 0; i < approvers.length; i++) {
89              if (!approvals[approvers[i]]) {
90                  return false;
91              }
92          }
93          return true;
94      }
95
96      function allApproved(address[] storage approvers,mapping(
           bytes32 => mapping(address => bool)) storage approvals,
           bytes32 paramHash) private view returns (bool) {
97          for (uint i = 0; i < approvers.length; i++) {
98              if (!approvals[paramHash][approvers[i]]) {
99                  return false;
100             }
101         }
102         return true;
103     }
104 }
```

Figure 7.1: Generated Solidity Code for a Multi-Signature Wallet

7.5 Execution Overhead

Finally, we measured the execution efficiency of Solidity contracts generated from *Quartz* descriptions and those of equivalent handwritten Solidity contracts. The handwritten contracts are the same as those described in Section 4.4, meaning they are adapted from existing codebases when available and simplified if necessary, e.g., by removing extra getter functions, to form a fair comparison.

Execution of Ethereum contracts is metered by *gas*, a cost assigned to each virtual machine operation, to ensure termination and discourage unnecessarily expensive contract code. It is therefore natural to measure a contract's execution efficiency by the gas it consumes. To accomplish this, we deployed both generated and handwritten versions of all case study contracts to a small private blockchain backed by nodes hosted on Amazon EC2 virtual machines. All members of the network ran Geth version 1.8.26 and used Geth's *Clique* proof-of-authority consensus mechanism. This allowed us to configure the network to use a fixed `gasPrice`. As a result, the gas cost of a particular workload is deterministic and reproducible. It does not fluctuate with network load as it would in a proof-of-work Ethereum network.

We wrote a contract client script for each case study using Python's `Web3` library. Each script deploys the generated and handwritten versions of Solidity code, invokes an equivalent sequence of transactions against both versions, and tallies all gas costs. For example, the script for the Auction case study initializes each contract and submits the same sequence of bids to both. The results of these measurements are shown in Figure 7.2. Each case study is represented along the x axis by a pair of bars. The number above each pair is the ratio of total gas costs for the generated Solidity code to total gas costs for the handwritten equivalent.

Gas costs for *Quartz*-generated Solidity contracts are competitive with those of handwritten contracts. While the overhead is 53% for the simple `ERC-1630` contract, for more substantial contracts it never exceeds 20%. Interestingly, there are some case studies where the generated contract actually has *lower* gas costs than the handwritten equivalent. Upon further investigation, we found that this was usually due to *Quartz*'s use of fewer, simpler data structures in its generated code. This typically gave the generated contract a cheaper constructor and, for some case studies, cheaper transactions when operating on a smaller body of state.

The multi-signature wallets are a good example. The handwritten wallets are based on a design used in production by Parity [78] and OpenZeppelin [77] where approvals by designated signers are tracked with both a Solidity mapping instance, to emulate a set and thus enable fast membership checks, and a Solidity array to allow iteration over all signers. This design avoids loops in the critical path but also requires bookkeeping to manage both the mapping and array. *Quartz* takes the simpler approach of using an array of signers and loops to check if enough signers have approved a transaction. This makes the *Quartz* wallets' constructors cheaper (there are fewer fields and less bytecode) and makes transactions cheaper when the total number of signers is small. The advantage of the *Quartz* wallets decreases

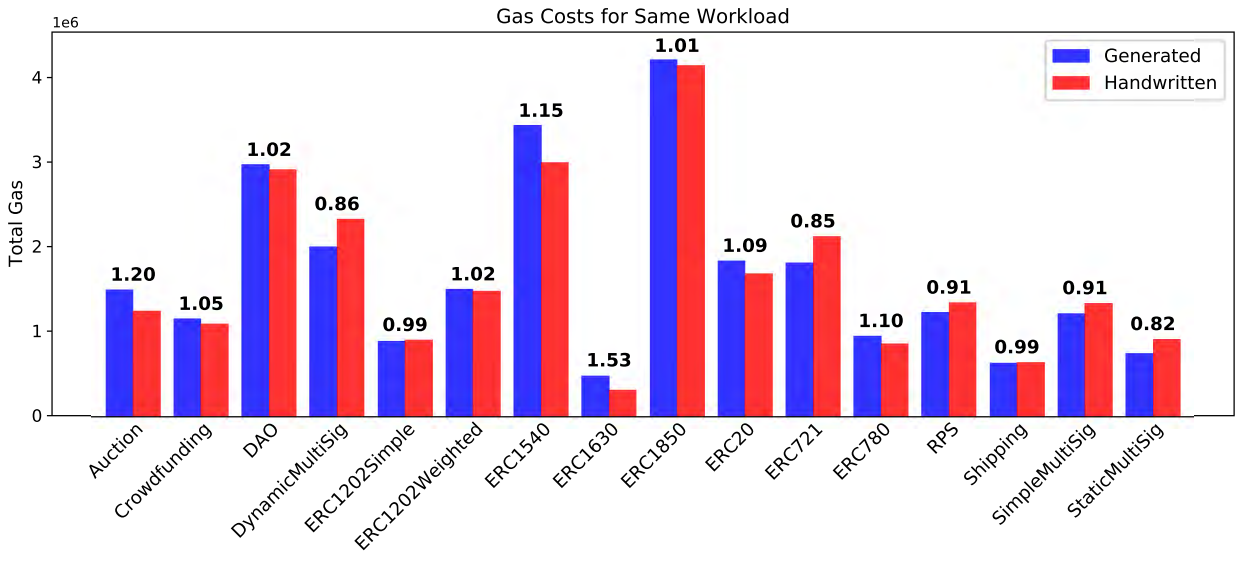


Figure 7.2: Gas costs when executing equivalent generated and handwritten Solidity code

under workloads with more signers.

The disadvantage of the *Quartz* approach is its use of loops, which means gas costs for wallet transactions grow with the number of signers. The advantage, however, is that the code generated by *Quartz* is flexible, because it must accommodate any valid sequence of *Quartz* operations against the group of signers and authorization checks against it, i.e., it cannot exploit domain knowledge and optimize based on assumptions of how authorized signers are added or removed over time.

7.6 Summary

This chapter completes the discussion of our approach, covering its second translation path from contract state machine written in *Quartz* to Solidity. It establishes two important results. First, it is feasible to seamlessly generate complete contract implementations from a simpler state machine description, meaning a contract author can work and reason within *Quartz*'s programming model and still run their contracts on Ethereum-based blockchains. Second, this chapter quantified the cost of operating at this higher level of abstraction.. We saw that the gas costs when executing a workload against a *Quartz*-generated contract was never more than 1.53× the cost of the same workload run against a handwritten equivalent.

Next, we present a promising direction of future work that extends the current application of model checking to cover economic notions of contract correctness before concluding.

Chapter 8

Incentive-Based Analysis

In this chapter, we present preliminary ideas and results around extending the approach to contract verification described thus far to include economic elements, which stand to improve the model checking process by guiding its search. We first motivate these ideas, provide results for simple contracts intended to serve as minimal working examples, and finally look ahead to next steps.

8.1 Incentive-Aware Contract Verification

Blockchains bring economic considerations to the fore of software design. Consensus algorithms like proof of work and proof of stake are engineered specifically to ensure that each participant's most profitable course of action is to engage honestly with one another, e.g., to reject invalid blocks and to share knowledge of new blocks with one another. Smart contracts inherit this perspective. A contract's users are expected to act strictly in their own interests to whatever extent the contract's code permits. This focus on economically informed design is part of what distinguishes blockchains and contracts from previous distributed systems.

As a result, there is a notion of contract correctness that relies on a consideration of user incentives rather than a purely mechanical analysis of contract code as this thesis has presented thus far. Specifically, even when a contract may permit a user to invoke a particular transformation (or, for a *Quartz* contract, a state transition), that user may have no incentive to do so, particularly if the execution of a transaction on the underlying blockchain incurs a cost for its sender. As a result, certain code paths may be reachable from a purely semantic perspective, but they are not viable from an economic perspective.

We can account for this perspective when model checking a smart contract. Recall that we can view model checking as a graph search problem: each node in the graph represents a particular contract state, while edges correspond to execution steps that move the contract from one state to another. A path in this graph corresponds to an execution trace. Normally, *Quartz*'s model checking fully traverses this graph, considering all possible actions permitted by contract code. However, if we were to augment the model checker's search logic to ignore

cases where a user's action would be against her own self interest, we can prune the search space, traversing only a subgraph of economically viable execution traces.

This type of analysis can identify two situations that are important to contract authors as they check their code for correctness:

1. Desirable execution traces that are never traversed because they correspond to a sequence of transactions with at least one instance of a user taking an economically unfavorable action.

2. Undesirable execution traces (e.g., a user exploits a reentrancy bug) that can be safely ignored because there is no incentive for users to exploit them.

Under normal model checking, the former case can be seen as a false negative where there is a problem with the contract that goes overlooked, specifically a case where desired forward progress is lost because the contract reaches a state where no user has an incentive to invoke a transaction to advance it any further. The latter case can be seen as a false positive where model checking identifies a bug that cannot be viably exploited.

The notion of an economically guided search is a natural fit for the blockchain and smart contract domain, but it is an idea that has also been well explored in prior work. Classical artificial intelligence offers a number of search algorithms such as A* and greedy breadth-first search that are guided by heuristics which attempt to predict the potential value of future actions, often in the context of a game setting. Another well-known approach is the minimax algorithm, where each participant in a game works backwards from the its end states to determine which of their actions will guide the game to the most favorable outcome.

8.2 Preliminary Results

As an initial proof of concept, we explored two potential techniques that account for user incentives when analyzing contract behavior. We performed experiments on minimal working example contracts defined using *Quartz*. These example contracts are designed to be as simple as possible while still inducing situations where an economically informed analysis of behavior differs from a purely semantic analysis. While they may not resemble fully formed use cases like the contracts we presented in Chapter 3, they are meant to resemble portions of larger contracts that may actually arise in practice.

Rather than implementing a new model checker, we chose instead to emulate what an incentive-based search process would achieve by post-processing the contract execution graph that is generated by *Quartz*'s existing model checking workflow. We configured the TLC model checker to output its search graph as a DOT file, a common format for representing graph structures as text. We then built Python scripts to parse these files and to apply the techniques described below.

Pruning Immediately Unfavorable User Actions

Consider the *Quartz* contract in Figures 8.1 and 8.2, which respectively represent a state machine and *Quartz* code implementing a two-player game. While it is simple, this game is designed to resemble real contracts deployed on the Ethereum [9]. The contract implements a guessing game which goes through the following sequence of steps:

1. The contract's creator, or the *owner*, initializes the contract with a stake of virtual currency, a designated *player*, and a hash value, produced from the concatenation of the owner's chosen number and a nonce. This forms a commitment scheme, i.e., the owner has bound herself to her chosen number without revealing it.

2. Next, the player submits her own stake of virtual currency and a guess at the owner's chosen value.

3. Finally, the owner reveals her chosen value and declares the outcome of the game. If the player guessed the owner's original value, she is the winner and receive the contract's balance of currency as a reward. Otherwise, the owner wins and receives these funds.

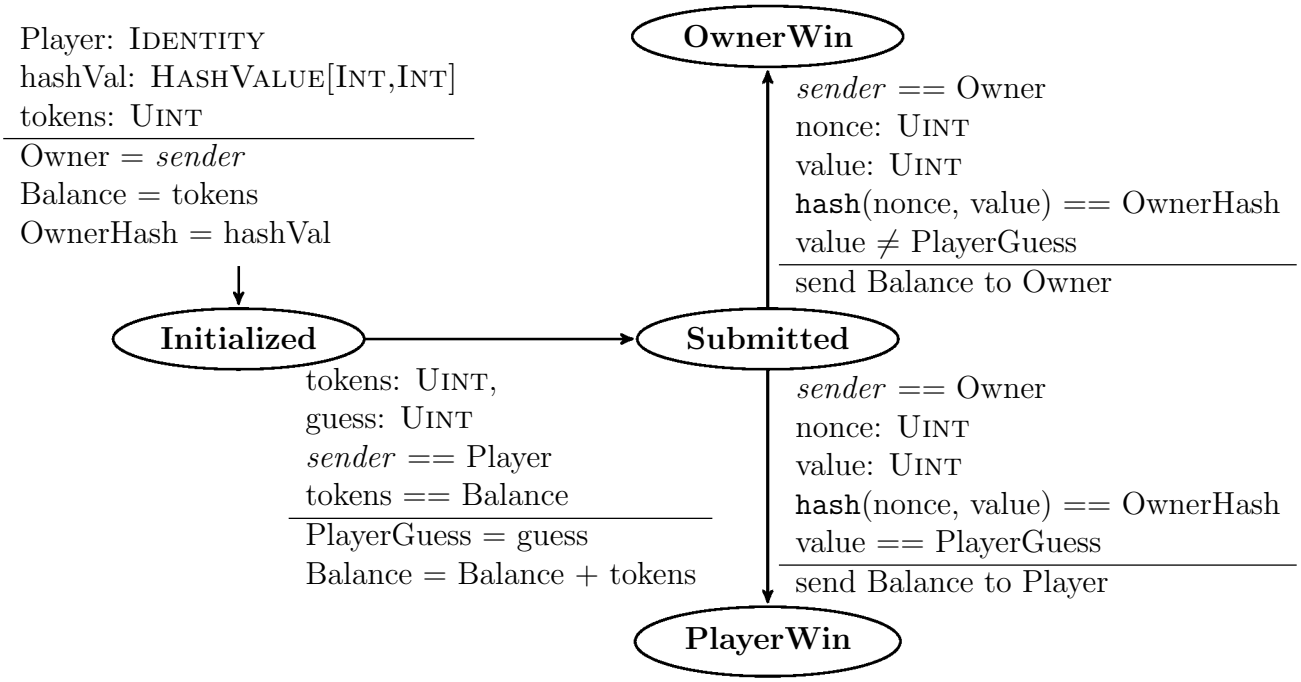


Figure 8.1: A Simple Two-Player Game – State Machine Representation

The flaw in this design is that the owner is the only party who can finish the game by revealing her original choice of number, yet she would only have an incentive to do this if she knew she would be the game's winner. The owner can see the player's guess and

```
contract SimpleGame {
    data {
        OwnerHash: HashValue[Uint, Uint]
        PlayerGuess: Uint
        Owner: Identity
        Player: Identity
        Balance: Uint
    }

    init: ->(tokens: Uint, hVal: HashValue[Uint, Uint], pl: Identity) initialized {
        OwnerHash = hVal
        Owner = sender
        Player = pl
        Balance = tokens
    }

    submitGuess: initialized ->(tokens: Uint, guess: Uint) submitted
    authorized [Player]
    requires [tokens == Balance] {
        PlayerGuess = guess
        Balance = Balance + tokens
    }

    revealValueOwnerWin: submitted ->(actualValue: Uint, nonce: Uint) ownerWin
    authorized [Owner]
    requires [hash(actualValue, nonce) == OwnerHash && PlayerGuess != actualValue] {
        send Balance to Owner
    }

    revealValuePlayerWin: submitted ->(actualValue: Uint) playerWin
    authorized [Owner]
    requires [hash(actualValue) == OwnerHash && PlayerGuess == actualValue] {
        send Balance to Player
    }
}
```

Figure 8.2: A Simple Two-Player Game – *Quartz* Implementation

therefore will know the game's outcome ahead of time. If she sees that revealing her choice can only trigger a loss, she may instead choose to do nothing, leaving the contract stuck in its `submitted` state with its balance of virtual currency frozen in place.

Even if the contract's author tried to use a commitment scheme for the player's guess as well as the owner's original number, one of the two parties must reveal first, inducing an asymmetry of knowledge where the other party now knows the game's only possible outcome and can behave accordingly. This stems from two aspects of smart contract execution. First, there is no notion of simultaneous action where players choose actions at the same time. Transactions are initiated by a single party and occur in a specific order. Second, contract users are never obligated to perform an action. They may simply choose to do nothing, leaving the contract stuck in its current state.

How might model checking surface this problem? One simple approach is to verify that all of the machine's states, as defined in the *Quartz* contract, are reachable when users avoid actions they can immediately identify as unfavorable. To emulate this kind of reasoning, we perform a breadth-first search over the graph produced by TLC for the *Quartz* code given in Figure 8.2. To model user incentives, we define a utility function of the form $U : S \times N \to \mathbb{R}$, where S is the set of possible contract states and N is a set containing its users. In other words, we assign each contract state a score specific to each of its users.

As a simple starting point, we can simply say that a state has utility 1 for a player when they have won the guessing game, -1 for a player when they have lost the guessing game, and 0 otherwise:

$$U(s,n) = \begin{cases} 1 & \text{if } s.\text{state} = \texttt{PlayerWin} \wedge n = \texttt{Player} \\ -1 & \text{if } s.\text{state} = \texttt{OwnerWin} \wedge n = \texttt{Player} \\ 1 & \text{if } s.\text{state} = \texttt{OwnerWin} \wedge n = \texttt{Owner} \\ -1 & \text{if } s.\text{state} = \texttt{PlayerWin} \wedge n = \texttt{Owner} \\ 0 & \text{otherwise} \end{cases} \tag{8.1}$$

The breadth-first search traversal is defined mostly as expected, with the additional element of checking for a non-negative change in utility before traversing an edge. Pseudocode for the traversal is shown in Figure 8.3. It assumes an adjacency-list representation of the model checker's execution graph, A, a root node r, and a utility function of the structure defined above, U. It simply returns the set of nodes seen during the traversal. We implemented this algorithm in Python, along with code to parse graphs produced by the TLC model checker. When applied to the graph for the two-player game defined in 8.2, we observe that the traversal never reaches the `PlayerWins` state, as expected. All edges in the graph corresponding to instances where the owner would reveal their choice and confirm their own loss would correspond to a decrease in utility for the owner.

```
BFS-INCENTIVE(A, r, U)
 1  visited = {r}
 2  Let Q be a new queue
 3  Q.PUSH(r)
 4  while Q.length > 0
 5        current = Q.POP()
 6        sender = current.sender
 7        for each neighbor ∈ A[current]
 8        if neighbor ∉ visited ∧ U(neighbor, sender) − U(current, sender) ≥ 0
 9              visited = visited ∪ {neighbor}
10              Q.PUSH(neighbor)
11  return visited
```

Figure 8.3: A Simple Breadth-First Search with Utility-Based Pruning

Pruning Unfavorable Execution Paths

Assuming that contract users only look one step ahead offers a simple starting point but is not very realistic. There are other cases when a contract's design may include similar flaws, where no eligible user has an incentive to advance the contract's state, but where that user has to deduce the results of multiple actions in sequence to reach this conclusion. Here, we can draw inspiration from the minimax algorithm from classical AI to similarly prune the model checking search space. Rather than pruning only user actions that result in an immediate decrease to utility, we prune actions that unfavorably constrain potential outcomes. In other words, we ignore actions where user works against her own interest by setting the contract on a course where all outcomes are unfavorable.

Consider a modified version of the two-player game initially defined above where instead of revealing and declaring the game's outcome in one step, the game contract's "owner" reveals her choice of number in a standalone step, and then any user is free to declare the game's outcome. This alternative is shown as a state machine in Figure 8.4. This contract is functionally equivalent to the original game contract, but it represents a variant of the game that a real contract author may implement in practice. It motivates an approach to incentive analysis that is robust to these kinds of variations and therefore needs to look more than one action ahead in its search space pruning decisions.

The flaw in this design is quite similar to that of the previous contract, but only becomes known when considering sequences of actions. Once the contract is in the `Submitted` state, we have the same asymmetry of information as before: the owner already knows the outcome of the game when the player does not and can therefore choose never to trigger the transition to the `Revealed` state. This transition does not induce an immediate decrease in utility for

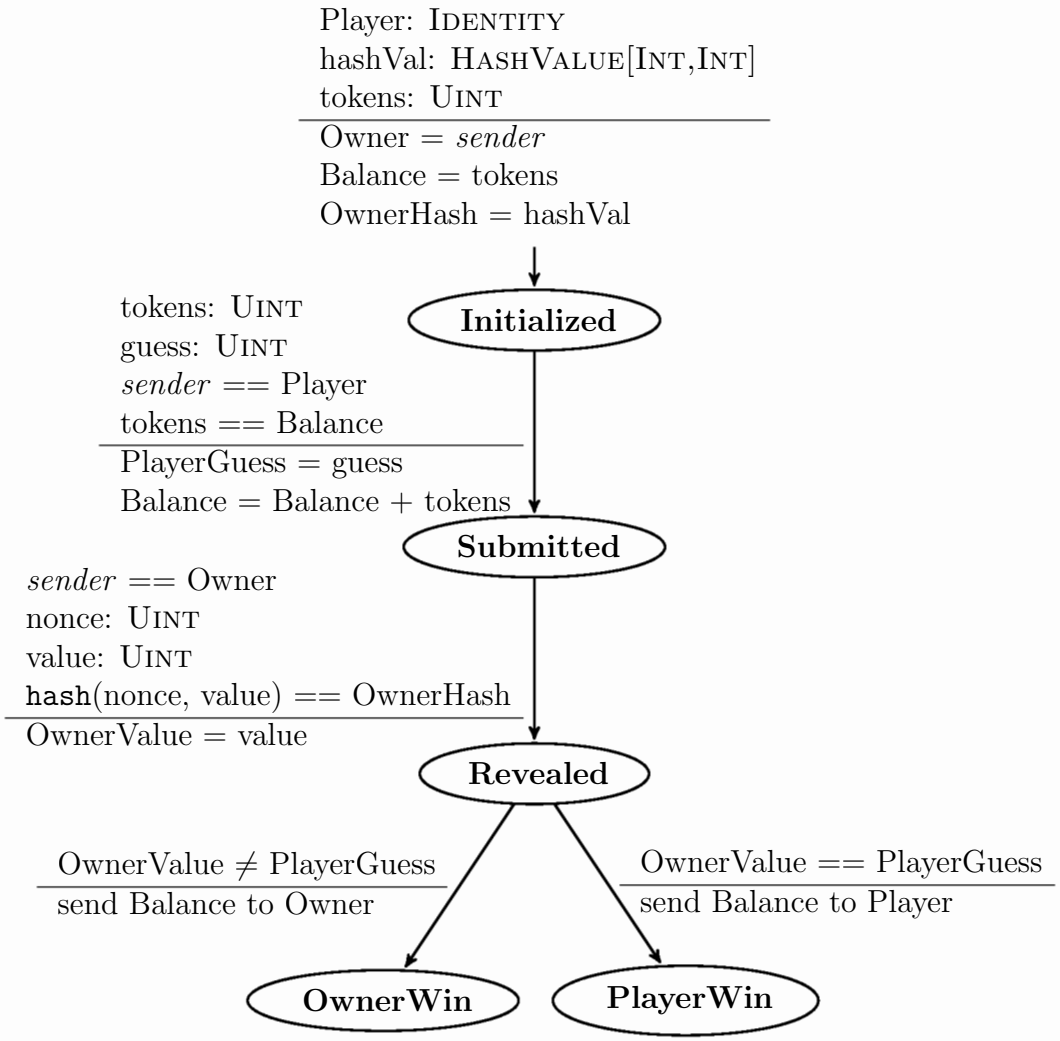


Figure 8.4: An Alternative Version of the Two-Player Guessing Game

the owner, but rather it may constrain her prospects to be entirely unfavorable. If the owner reveals that her originally chosen number matches the player's guess, then only the player would benefit from this action.

We can address this by computing, for each node in the model checker's graph, the set of utility values for terminal states that are reachable from that node. A terminal state, in this context, is a node with no outgoing edges, which indicates the end of contract execution. This can be accomplished with a depth-first traversal, defined in Figure 8.5. It takes as parameters a set of nodes N, an adjacency list A, a set of identities I, root node r at which to start the search, and a utility function U of the form $U : N \times I \to \mathbb{R}$.

Once again, we implemented a prototype of this approach in Python. We generated an

execution graph for a *Quartz* contract that is equivalent to the state machine shown in Figure 8.4 and then post-processed this graph with the DFS-UTILITY algorithm of Figure 8.5. We were successful in pruning unfavorable paths, but also identified several shortcomings that motivate future work.

```
DFS-UTILITY(N, A, I, r, U)
1  for each n ∈ N
2       n.visited = FALSE
3       for each i ∈ I
4            n.utilities[i] = ∅
5  VISIT(r, A, I, U)

VISIT(n, A, I, U)
 1  n.visited = TRUE
 2  if A[n] == ∅
 3       for each i ∈ I
 4            n.utilities[i] = {U(n, i)}
 5  else
 6       for each neighbor ∈ A[n]
 7            if !neighbor.visited
 8                 VISIT(neighbor, A, I, U)
 9            for each i ∈ I
10                 n.utilities[i] = n.utilities[i] ∪ neighbor.utilities[i]
```

Figure 8.5: A Depth-First Graph Traversal to Compute Node Utilities

8.3 Fixing Economic Flaws

We have seen how a failure to account for user incentives in contract design can lead to a loss of forward progress. How does a contract author address this issue? One of the most straightforward approaches is to create an explicit incentive for all parties involved to run a contract to its completion. This can be achieved by asking users to put down an initial security deposit. For example, in the two-player game discussed above, the contract's owner initializes the contract with a deposit of virtual currency (on top of the wager that is paid out to the game's winner), and the second player makes a similar deposit when she submits her guess. The two players may recover their deposits only once the game has reached its conclusion. If we were to perform an economically informed search of the contract's execution

graph, we would find that even when a player must take an action that will cause her to lose the game, the loss of the wager is outweighed by the potential loss of deposit (assuming this deposit is large enough), meaning this action would not be pruned.

8.4 Future Work

In this section, we discuss ideas for following up this initial exploration of incentive-based contract analysis with more substantial work that could eventually be incorporated into *Quartz*.

Directly Incorporating Incentives into Model Checking To push these ideas further, we will need to change our approach to *Quartz*'s model checking, potentially by altering the structure of its generated TLA$^+$ graphs to make them more amenable to incentive analysis or by incorporating some of the pruning logic described above directly into the search logic. First, we need to address shortcomings surfaced by our Python prototype for pruning unfavorable execution paths. Currently, our approach to model checking produces a search graph that makes it easy to verify safety properties, which are state-oriented, rather than liveness properties, which are path-oriented. This creates difficulties when we want to a perform similarly path-oriented incentive analysis.

One example is the need for more intelligence concerning exceptions. The way in which we have specified a contract's behavior around exceptions in *Quartz*'s auto-generated TLA$^+$ means that there are paths in the model checking search graph where the contract's state machine transitions from state s to state t, an exception is thrown during a transfer of funds in t's body, and the contract reverts to s. A naive traversal of the execution graph then surmises that state s is reachable from state t and that utility calculations should reflect this, when in reality this is meant to reflect control flow rather than an element of the contract's permanent state and transaction history.

Tracking Flows Virtual Currency One clear need is the ability to automatically account for the movement of cryptocurrency into and out of a contract when tracing its execution. Payment (or loss) of virtual currency is an explicit incentive (or disincentive) for blockchain users that is often used to ensure the integrity of a contract's proceedings. Tracking currency flows would allow *Quartz* to reason about a wider range of situations. For example, the established design pattern in the Solidity community is to construct contracts such that users must manually invoke a transaction to withdraw any funds they are owed. [35] Strictly speaking, a recipient of funds could throw an exception to disrupt their own refund, but this would prevent movement of currency from the contract into their account, which is presumably counter to their own interests, meaning this scenario could be pruned from the search space.

Contract authors are also interested in verifying properties concerning currency flows. For example, a contract is written with the goal of ensuring that currency is conserved,

i.e., all currency that flows into a contract is eventually paid out to users. Model checking could examine all terminal contract states to ensure there are no residual funds "trapped" in the contract's possession. Similarly, a common contract design pattern to ensure forward progress is to require all parties to place an initial deposit of currency that is held by the contract and only returned when its proceedings have completed. One could imagine redesigning the two-player guessing game given above by requiring the contract owner to place such a deposit on initialization. *Quartz* could then automatically validate this approach by checking that the owner always has incentive to reveal her guess, whether by the winner's profits or avoiding loss of her deposit.

Interface for Contract Authors While currency flows are an obvious and explicit incentive, a contract may involve implicit incentives that are specific to its particular use case. For example, a vendor in a supply chain contract does not want to see that contract reach some state indicating they will have to pay a cost assessed out of band (independently of the blockchain). Incentive analysis will require input from the contract author that defines these implicit incentives. We alluded to one potential approach above: the contract author defines a utility function on the contract's state and users. Integrating this into *Quartz* would involve expanding its domain-specific language so that a contract state machine can be accompanied by such a definition. This would involve its own set of considerations around language design and the appropriate primitives for this task.

8.5 Summary

Here, we have seen how an approach to contract verification that relies on model checking, which determines correctness through a purely mechanical treatment of contract logic, can be naturally extended to incorporate economic reasoning. This lends further credence to our choice of model checking as the primary verification technique underlying our approach. Model checking offers automated analysis, the flexibility to verify arbitrary invariants, and the potential to assess contract behavior within the incentive-focused environment that characterizes blockchains. In the next, and last, chapter, we summarize our results and enumerate other directions for future work.

Chapter 9

Conclusion

In this chapter, we summarize results, list some of the important lessons learned from this work, propose future directions that address existing shortcomings or new issues raised by *Quartz*, and offer concluding remarks on the future potential of our proposed approach as the blockchain space continues to evolve.

9.1 Results

This thesis offers a methodology for the implementation and systematic testing of smart contracts, enabling an engineering workflow in which contracts are iteratively improved and validated before final deployment and consumption by end users. Our approach is organized around a domain-specific language in which contracts are defined as finite-state machines and a verification backend leveraging model checking. We have designed this language based on the insights gleaned from a survey of 16 case study smart contracts. These are assembled from a variety of sources including standardized contracts, use cases featured in related work on contract verification, and real contracts on the Ethereum blockchain that were compromised due to flaws in their implementations. Each of these case studies also serves as an evaluation benchmark.

We put this methodology into practice through *Quartz*, a prototype system complete with a contract description language and translation paths that enable verification through model checking and generation of a deployment-ready implementation. The construction of this language and its translators is informed by formal definitions of a state machine's structure and execution on a blockchain. *Quartz* is able to express all 16 contracts included in our case study set. Additionally, *Quartz* contracts are reasonably concise. Logic- and arithmetic-heavy contracts are on par in length to Solidity equivalents, while contracts with sophisticated authorization logic or specifically defined lifecycles tend to be shorter than Solidity equivalents. As in many other settings, operating at a higher level of abstraction can reduce execution efficiency, naturally quantified in the blockchain setting as transaction gas costs. For our case study settings, we find that a *Quartz* contract consumes at most about

1.5× the gas of an equivalent handwritten in Solidity, and that gas costs are competitive between the two types of contracts overall.

Model checking proves to be an effective means of elucidating contract flaws. We are able to translate state machine descriptions written in *Quartz* to TLA$^+$ specifications that allow verification of invariants in light of both the state machine's logic and the control-flow issues inherent in blockchain-based execution. For example, *Quartz* identifies an occasion where the proceedings of an auction are compromised due to an unanticipated transfer of control to an external party, and it identifies a case where a contract fails to properly track ownership of a finite supply of virtual shares due to a logic error. Finally, our approach offers interesting prospects for future work in which we can integrate an analysis of user incentives directly into the model checker's search logic.

9.2 Lessons Learned

Here, we discuss some of main ideas learned in the course of this work.

Simplifying Implementation and Analysis Using a state machine-based DSL allows us to confront two challenges at once: giving more structure to contract definitions to simplify their implementation, and constraining the behavior of these contracts compared to those written in general-purpose languages to facilitate their verification. Under our design, these two goals reinforce one another. There are a number of other areas where domain specific languages may be able to play a similar role of streamlining development while also supporting the construction of ancillary tools for tasks like debugging, formal analysis, and linting, such as defining machine learning and data processing pipelines.

The Different Roles of Language Translation The translation from *Quartz* code to a Solidity implementation is fairly traditional in that it is primarily concerned with equivalence and efficiency, i.e., that it produces Solidity code that faithfully recreates the semantics of the original state machine and that is not significantly more verbose or less performant than handwritten Solidity. The TLA$^+$ translation is slightly different in that it embeds the functionality of the original state machine into a formal specification that also captures the blockchain's execution semantics. Expressing these semantics in TLA$^+$ is feasible but does have challenges, such as accurately modeling exceptions, which motivates the idea of eliminating the system's reliance on TLA$^+$ described below.

The Value of Model Checking A search-based approach to contract verification like model checking has multiple benefits. First, it is a good fit for validating contract code against the non-deterministic events that characterize execution on a blockchain - arbitrary transition ordering, different possible responses by the recipient contract when a transfer of Ether occurs, etc. Most systems based around alternative techniques for contract verification do not fully account for this non-determinism, ignoring important issues like reentrancy.

Second, as described in Chapter 8, the graph produced by the model checker doubles as a sort of game tree, making it natural to analyze execution paths from an economic standpoint.

9.3 Future Directions

Quartz lays a foundation for authoring and analyzing smart contracts, but there is much still to be done in this space. Below, we offer a few potential areas where *Quartz* could be extended or improved.

More Complete Evaluation Arguably the most prominent gap in the work presented here is that it does not constitute a full empirical validation of the approach. The same set of case studies is used as the basis for *Quartz*'s design as well as for its evaluation. We still need to assess how well our approach generalizes to contracts outside of this sample. Additionally, while our simplified contract description language is intended to be less error-prone and easier for developers to learn and effectively use, these claims need to be substantiated with experimental evidence.

There are two potential ways to address these concerns. First, we can perform a user study in which one pool of subjects are taught Solidity while a second pool are taught about *Quartz*. Then, each participant must implement equivalent contracts in her assigned language. Several measurements could come out of this process such as the total amount of time required to complete a contract or the number of errors contained in each user's final implementation. These measurements would quantify concepts such as ease of use and developer error rates when working in Solidity versus in *Quartz*. This experiment could then be extended to incorporate a validation or testing element in which we compare techniques from prior work for finding bugs in Solidity contracts to *Quartz* and its testing methodology grounded in model checking.

Second, we can offer *Quartz* to practicing contract developers for use in the field. We can then study how well *Quartz* is applied to implementation tasks we did not originally anticipate through our case studies. From this, we can observe issues like the ease or the extent of *Quartz*'s adoption and any shortcomings in its contract description language. One particularly important outcome of this work is greater insight into the utility of *Quartz*'s contract validation process. Can contract developers determine a fruitful set of properties to verify? In other words, how much of an impediment is *Quartz*'s expectation that its users must identify the properties that are likely to surface contract flaws?

Eliminating the Need for TLA$^+$ Currently, a *Quartz* contract must be translated to TLA$^+$ for model checking. While this allowed us to more quickly develop an end-to-end system, it makes the correctness of this system reliant on the accuracy of our TLA$^+$ translation. However, it would be possible to avoid this translation entirely and instead search potential execution paths by operating directly on the abstract syntax tree parsed from the original contract. This would be akin to abstract interpretation as it would directly

implement and evaluate *Quartz*'s operational semantics. However, execution would remain organized around a loop in which any state transition may be invoked by any user, much like the existing approach based around TLA$^+$, and thus would involve a search space of potential execution paths as in model checking.

Continuing to Explore Incentive-Based Analysis We presented a preliminary exploration of this idea in Chapter 8. If we were to eliminate *Quartz*'s reliance on TLA$^+$ as described above, it would have the added benefit of allowing us to incorporate incentive-based reasoning directly into the search process. As described earlier, this kind of reasoning might exclude paths in which a user must act against her own self-interest to identify issues with contract liveness.

Proving Correctness of the Implementation Generation By translating *Quartz* to Solidity, we avoid committing the system to any particular bytecode representation. This is important given Ethereum's expected transition from the EVM bytecode to a restricted dialect of WebAssembly. Under our current approach, *Quartz* need only change compilers to accommodate this change. However, we could also choose to bypass Solidity and generate EVM bytecode directly. Then, given that we have formally defined semantics for both *Quartz* and for EVM bytecode [98], we could prove the correctness of our bytecode generation by establishing an isomorphism between evaluation of each *Quartz* construct and the evaluation of the equivalent body of bytecode generated by our translator.

Targeting Additional Blockchains Ethereum is not the only blockchain that supports smart contracts. There are several other platforms that seek to serve this need, most notably Hyperledger Fabric. It would be interesting to build translators that produce contract implementations suitable for these alternative platforms. We could then make the argument that a higher-level contract description language like *Quartz* also offers the benefit of portability across blockchains.

Enabling Multi-Contract Applications Currently, *Quartz* is restrictive in that it only permits the development, analysis, and deployment of individual, monolithic contracts. However, blockchain applications can be built from the composition of multiple independent contracts. Therefore, it would be valuable to extend *Quartz* to support the development of contract groups, each still defined as a state machine, but analyzed together where we account for situations in which one state machine is able to invoke another.

9.4 Final Remarks

What are the long-term prospects of blockchain technology and smart contracts? At this point, it is still too early to say. While the initial hype has arguably diminished, blockchains

remain an active area of research and development, and the conventional wisdom around their design and use continues to evolve. Since *Quartz* was initially conceived, a number of changes have already begun to occur in the blockchain space. For example, Ethereum is working towards a new public blockchain implementation [32] that replaces proof of work with proof of stake and the custom Ethereum virtual machine with a new runtime based on WebAssembly, which already has well-established compiler toolchains that can accommodate a variety of high-level source languages like C++ and Rust. Similarly, Hyperledger's Fabric [6] project has focused on permissioned blockchains and offers a runtime that supports contracts written in several traditional programming languages.

There is also growing interest in using a blockchain in a more judicious manner by separating the execution of contract transactions from the consensus protocol, avoiding an approach like Ethereum's in which the two are coupled. Instead, the blockchain stores only cryptographically signed and auditable transaction receipts that record the results of transaction execution. Several systems have adopted this approach such as Oasis (formerly Ekiden) [22], Hyperledger Avalon [64], and Algorand [41,68]. This line of inquiry is also pushing the field towards contracts written in general-purpose, high-level programming languages rather than contract-specific programming languages like Solidity.

This shift alters the considerations for a system like *Quartz* but does not eliminate its value. While traditional languages have far better tooling for compilation, debugging, and verification than contract languages, they do not replace a more simplified framework based on state machines as offered by *Quartz*. However, this would require us to revisit our choice of target implementation language, as neither Solidity nor EVM bytecode would be obvious candidates. Instead, we would need to identify a target language that is well supported by blockchain platforms and amenable to a proof of equivalence with *Quartz*'s formal semantics.

Blockchains have generated renewed interest in distributed systems that offer cryptographically auditable and tamper-resistant data storage. However, it is important to distinguish these capabilities from those of decentralization and strict enforcement of rules on state updates, as enforced by smart contracts. Systems built on top of cryptographic primitives [4, 42] can achieve auditability and tamper-resistance without relying on fully fledged blockchains and the scalability issues they currently bring with them. Work is ongoing to make blockchains more scalable, [39, 59, 89] but the extent to which true decentralization is required or desirable across a broad array of applications remains to be seen.

What is clear is that smart contracts are crucial to the blockchain vision. This work is intended neither as an endorsement nor critique of that vision, but rather a step in the collective effort to weigh the merits and limitations of smart contracts in practice. We have explored the value of adopting a more constrained programming model to facilitate both expression and analysis of smart contracts. *Quartz*'s use of model checking was inspired by this technique's effective application to systems like distributed cache coherence protocols and device drivers, where correctness is prized and the system may be faced with arbitrary sequences of events. Our hope is that *Quartz* may similarly inspire future work as the blockchain space continues to mature.

Bibliography

[1] Abdulla, P. A., Jonsson, B., Nilsson, M., and Saksena, M. A survey of regular model checking. In *CONCUR 2004 - Concurrency Theory* (Berlin, Heidelberg, 2004), P. Gardner and N. Yoshida, Eds., Springer Berlin Heidelberg, pp. 35–48.

[2] Aeternity. The sophia language. `https://github.com/aeternity/aesophia/blob/lima/docs/sophia.md`, 2020.

[3] Andersen, M. P., Kolb, J., Chen, K., Fierro, G., Culler, D. E., and Popa, R. A. Wave: A decentralized authorization system for iot via blockchain smart contracts. Tech. Rep. UCB/EECS-2017-234, EECS Department, University of California, Berkeley, Dec 2017.

[4] Andersen, M. P., Kumar, S., AbdelBaky, M., Fierro, G., Kolb, J., Kim, H.-S., Culler, D. E., and Popa, R. A. WAVE: A decentralized authorization framework with transitive delegation. In *28th USENIX Security Symposium (USENIX Security 19)* (Aug. 2019), pp. 1375–1392.

[5] Andrews, T., Qadeer, S., Rajamani, S. K., Rehof, J., and Xie, Y. Zing: A model checker for concurrent software. In *International Conference on Computer Aided Verification* (2004), Springer, pp. 484–487.

[6] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S., and Yellick, J. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (2018), EuroSys '18, ACM, pp. 30:1–30:15.

[7] Atzei, N., Bartoletti, M., and Cimoli, T. A survey of attacks on ethereum smart contracts. In *Principles of Security and Trust*, vol. 10204 of *Lecture Notes in Computer Science*. Springer, 2017, pp. 164–186.

[8] Atzei, N., Bartoletti, M., and Cimoli, T. A survey of attacks on ethereum smart contracts. In *Principles of Security and Trust*, vol. 10204 of *Lecture Notes in Computer Science*. Springer, 2017, pp. 164–186.

[9] Bartoletti, M., and Pompianu, L. An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns. In *Financial Cryptography and Data Security* (2017), Springer, pp. 494–509.

[10] Bayer, D., Haber, S., and Stornetta, W. S. Improving the efficiency and reliability of digital time-stamping. In *Sequences II*. Springer, 1993, pp. 329–334.

[11] Ben-Ari, M. A primer on model checking. *ACM Inroads 1*, 1 (Mar. 2010), 40–47.

[12] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., and Zanella-Béguelin, S. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security* (2016), PLAS '16, ACM, pp. 91–96.

[13] Bigi, G., Bracciali, A., Meacci, G., and Tuosto, E. Validation of decentralised smart contracts through game theory and formal methods. In *Programming Languages with Applications to Biology and Security*. Springer, 2015, pp. 142–161.

[14] Biryukov, A., Khovratovich, D., and Tikhomirov, S. Findel: Secure Derivative Contracts for Ethereum. In *Financial Cryptography and Data Security* (2017), Springer, pp. 453–467.

[15] Black, M., and Cai, T. ERC-1850 hashed time-locked principal contract. `https://github.com/ethereum/EIPs/pull/1850`, 2019.

[16] Black, M., and Liu, T. ERC-1630 hashed time-locked contracts. `https://github.com/ethereum/EIPs/issues/1631`, 2018.

[17] Blackshear, S., Cheng, E., Dill, D. L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Russi, D., Sezer, S., Zakian, T., and Zhou, R. Move: A Language With Programmable Resources. `https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources.pdf`, 2020.

[18] Bliudze, S., Cimatti, A., Jaber, M., Mover, S., Roveri, M., Saab, W., and Wang, Q. Formal verification of infinite-state BIP models. In *International Symposium on Automated Technology for Verification and Analysis* (2015), Springer, pp. 326–343.

[19] Buterin, V. A Next-Generation Smart Contract and Decentralized Application Platform. `https://github.com/ethereum/wiki/wiki/White-Paper`, 2014.

[20] Cachin, C., and Vukolic, M. Blockchain Consensus Protocols in the Wild. *ArXiv* (7 2017).

[21] Chandra, S., Richards, B., and Larus, J. R. Teapot: A domain-specific language for writing cache coherence protocols. *IEEE Transactions on Software Engineering 25*, 3 (1999), 317–333.

[22] Cheng, R., Zhang, F., Kos, J., He, W., Hynes, N., Johnson, N., Juels, A., Miller, A., and Song, D. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)* (2019), pp. 185–200.

[23] Coblenz, M. Obsidian: A Safer Blockchain Programming Language. In *Proceedings of the 39th International Conference on Software Engineering Companion* (2017), ICSE-C '17, ACM, pp. 97–99.

[24] Coblenz, M. J., Aldrich, J., Sunshine, J., and Myers, B. A. User-Centered Design of Permissions, Typestate, and Ownership in the Obsidian Blockchain Language. In *HCI for Blockchain: Studying, Designing, Critiquing and Envisioning Distributed Ledger Technologies Workshop at CHI 2018* (2018), ACM.

[25] DappHub. dapp.tools. `http://dapp.tools/`, 2020.

[26] del Castillo, M. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft. `https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft`, 2016.

[27] Delmolino, K., Arnett, M., Kosba, A., Miller, A., and Shi, E. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security* (2016), Springer, pp. 79–94.

[28] Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., and Zufferey, D. P: Safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 321–332.

[29] Dill, D. L., Drexler, A. J., Hu, A. J., and Yang, C. H. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors* (1992), vol. 92, pp. 522–525.

[30] Embark Labs. Embark. `https://framework.embarklabs.io/`, 2020.

[31] Entriken, W., Shirley, D., Evans, J., and Sachs, N. ERC-721 non-fungible token standard. `https://eips.ethereum.org/EIPS/eip-721`, 2018.

[32] Ethereum Foundation. Ethereum 2.0 (eth2). `https://ethereum.org/en/eth2`.

[33] Ethereum Foundation. Serpent. `https://github.com/ethereum/serpent`, 2017.

[34] Ethereum Foundation. Common Patterns. `http://solidity.readthedocs.io/en/v0.4.24/common-patterns.html`, 2018.

[35] Ethereum Foundation. Common patterns – solidity documentation. `https://solidity.readthedocs.io/en/latest/common-patterns.html`, 2019.

[36] Ethereum Foundation. Solidity. `https://solidity.readthedocs.io/en/latest/`, 2019.

[37] Ethereum Foundation. Vyper. `https://github.com/ethereum/vyper`, 2019.

[38] Ethereum Foundation. Solidity by example: Simple open auction. `https://solidity.readthedocs.io/en/v0.6.6/solidity-by-example.html`, 2020.

[39] Eyal, I., Gencer, A. E., Sirer, E. G., and Renesse, R. V. Bitcoin-NG: A Scalable Blockchain Protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), Usenix, pp. 45–59.

[40] Frantz, C. K., and Nowostawski, M. From Institutions to Code: Towards Automated Generation of Smart Contracts. In *2016 IEEE First International Workshops on Foundations and Applications of Self* Systems (FAS*W)* (9 2016), IEEE, pp. 210–215.

[41] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., and Zeldovich, N. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, ACM, pp. 51–68.

[42] Google. Certificate transparency. `https://www.certificate-transparency.org/`, 2020.

[43] Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., and Zohar, Y. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang. 2*, POPL (Dec. 2017), 48:1–48:28.

[44] Gurfinkel, A., Kahsai, T., Komuravelli, A., and Navas, J. A. The seahorn verification framework. In *Computer Aided Verification* (Cham, 2015), D. Kroening and C. S. Păsăreanu, Eds., Springer International Publishing, pp. 343–361.

[45] Haber, S., and Stornetta, W. S. How to Time-Stamp a Digital Document. In *Conference on the Theory and Application of Cryptography* (1990), Springer, pp. 437–455.

[46] Haber, S., and Stornetta, W. S. Secure Names for Bit-Strings. In *Proceedings of the 4th ACM Conference on Computer and Communications Security* (1997), ACM, pp. 28–35.

[47] Hashfuture Inc. Etherescan: ERC-1540 contract. `https://etherscan.io/address/0x565b7bd8056322f96dac28345245aead44f24ff2#code`, 2018.

[48] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. Software verification with BLAST. In *Model Checking Software* (Berlin, Heidelberg, 2003), T. Ball and S. K. Rajamani, Eds., Springer Berlin Heidelberg, pp. 235–239.

[49] Herlihy, M. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing* (2018), pp. 245–254.

[50] Herlihy, M. Blockchains From a Distributed Computing Perspective. `https://cs.brown.edu/courses/csci2952-a/papers/perspective.pdf`, 2018.

[51] Hertig, A. Ethereum's Two Ethereums Explained. `https://www.coindesk.com/ethereum-classic-explained-blockchain`, 2016.

[52] Hirai, Y. Bamboo: A Morphing Smart Contract Language. `https://github.com/pirapira/bamboo`, 2018.

[53] Holzmann, G. J. The model checker SPIN. *IEEE Transactions on Software Engineering 23*, 5 (1997), 279–295.

[54] IETF. Transmission control protocol: DARPA internet program protocol specification. `https://tools.ietf.org/html/rfc793`, 1981.

[55] IOHK Foundation. Plutus introduction. `https://cardanodocs.com/technical/plutus/introduction`, 2018.

[56] IOHK Foundation. Marlowe: A contract language for the financial world. `https://testnets.cardano.org/en/marlowe/`, 2020.

[57] Jiang, A., Jia, Y., Ren, Y., and Dong, J. ERC-1540 asset token standard. `https://github.com/ethereum/EIPs/pull/1540`, 2018.

[58] Kalra, S., Goel, S., Dhawan, M., and Sharma, S. Zeus: Analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS* (2018), pp. 18–21.

[59] Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., and Ford, B. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), pp. 583–598.

[60] Kolb, J., AbdelBaky, M., Katz, R. H., and Culler, D. E. Core concepts, challenges, and future directions in blockchain: A centralized tutorial. *ACM Comput. Surv. 53*, 1 (Feb. 2020).

[61] Konnov, I., Kukovec, J., and Tran, T.-H. TLA+ model checking made symbolic. *Proc. ACM Program. Lang. 3*, OOPSLA (Oct. 2019), 123:1–123:30.

[62] Krupp, J., and Rossow, C. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 1317–1333.

[63] Lamport, L. *Specifying Systems: the TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[64] Linux Foundation. Hyperledger avalon. `https://www.hyperledger.org/use/avalon`, 2020.

[65] Luu, L., Chu, D. H., Olickel, H., Saxena, P., and Hobor, A. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS '16, ACM, pp. 254–269.

[66] Mavridou, A., and Laszka, A. Designing secure ethereum smart contracts: A finite state machine based approach. In *International Conference on Financial Cryptography and Data Security* (2018), Springer, pp. 523–540.

[67] Mavridou, A., Laszka, A., Stachtiari, E., and Dubey, A. Verisolid: Correct-by-design smart contracts for ethereum. In *International Conference on Financial Cryptography and Data Security* (2019), Springer, pp. 446–465.

[68] Micali, S. Algorand's smart contract architecture. `https://www.algorand.com/resources/blog/algorand-smart-contract-architecture`, 2020.

[69] Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., and Dinaburg, A. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. arxiv 2019. *arXiv* (2019).

[70] Nakamoto, S. Bitcoin: A Peer–to–Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, 2009.

[71] Nehai, Z., Piriou, P.-Y., and Daumas, F. Model-checking of smart contracts. In *IEEE International Conference on Blockchain* (2018), pp. 980–987.

[72] Newcombe, C. Why amazon chose TLA+. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z* (Berlin, Heidelberg, 2014), Y. Ait Ameur and K.-D. Schewe, Eds., Springer Berlin Heidelberg, pp. 25–39.

[73] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., and Hobor, A. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference* (2018), ACM, pp. 653–663.

[74] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., and Hobor, A. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference* (2018), ACM, pp. 653–663.

[75] Nomic Labs. Buidler. `https://buidler.dev/`, 2020.

[76] OpenSea, Inc. OpenSea. `https://opensea.io`, 2020.

[77] OpenZeppelin. OpenZeppelin Contracts. `https://github.com/OpenZeppelin/openzeppelin-contracts`, 2020.

[78] Palladino, S. The parity wallet hack explained. `https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7`, 2017.

[79] Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., and Vechev, M. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy* (2020), p. 17.

[80] Peyton Jones, S., Eber, J.-M., and Seward, J. Composing contracts: An adventure in financial engineering. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), ICFP '00, ACM, pp. 280–292.

[81] Popejoy, S. The pact smart contract language. `https://kadena.io/docs/Kadena-PactWhitepaper.pdf`, 12 2017.

[82] Rust Team. The Rust Programming Language. `https://www.rust-lang.org/`, 2020.

[83] Schrans, F., Eisenbach, S., and Drossopoulou, S. Writing safe smart contracts in flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming* (New York, NY, USA, 2018), Programming'18 Companion, Association for Computing Machinery, p. 218–219.

[84] Sergey, I., and Hobor, A. A concurrent perspective on smart contracts. In *Financial Cryptography and Data Security*, vol. 10323 of *Lecture Notes in Computer Science*. Springer, 4 2017, pp. 478–493.

[85] Sergey, I., Kumar, A., and Hobor, A. Scilla: a Smart Contract Intermediate-Level LAnguage. *ArXiv* (Jan. 2018).

[86] Sergey, I., Kumar, A., and Hobor, A. Temporal properties of smart contracts. In *International Symposium on Leveraging Applications of Formal Methods* (2018), Springer, pp. 323–338.

[87] Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., and Hao, K. C. G. Safer smart contract programming with scilla. *Proceedings of the ACM on Programming Languages 3*, OOPSLA (2019), 1–30.

[88] SLOCK.IT. DAO. `https://github.com/slockit/DAO`, 2018.

[89] SOMPOLINSKY, Y., AND ZOHAR, A. PHANTOM, GHOSTDAG: Two Scalable BlockDAG protocols. *IACR Cryptology ePrint Archive* (2018).

[90] SZABO, N. Formalizing and securing relationships on public networks. *First Monday 2*, 9 (1997).

[91] TEZOS. The michelson language. `https://www.michelson-lang.com/`, 2019.

[92] TEZOS. Ligo. `https://ligolang.org/`, 2020.

[93] TIKHOMIROV, S., VOSKRESENSKAYA, E., IVANITSKIY, I., TAKHAVIEV, R., MARCHENKO, E., AND ALEXANDROV, Y. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain* (New York, NY, USA, 2018), WETSEB '18, Association for Computing Machinery, p. 9–16.

[94] TORSTENSSON, J. ERC-780 ethereum claims registry. `https://github.com/ethereum/EIPs/issues/780`, 2017.

[95] TRUFFLE BLOCKCHAIN GROUP. Truffle Blockchain Suite. `https://www.trufflesuite.com/`, 2020.

[96] TSANKOV, P., DAN, A., DRACHSLER-COHEN, D., GERVAIS, A., BÜNZLI, F., AND VECHEV, M. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), CCS '18, pp. 67–82.

[97] VOGELSTELLAR, F., AND BUTERIN, V. ERC-20 token standard. `https://eips.ethereum.org/EIPS/eip-20`, 2015.

[98] WOOD, G. Ethereum: A Secure Decentralised Generalised Transaction Ledger. `http://gavwood.com/paper.pdf`, 2014.

[99] ZHOU, Z. V., BOTELLO, E., AND XU, Y. ERC-1202 voting standard. `https://eips.ethereum.org/EIPS/eip-1202`, 2018.