Refining High-Level Specifications of Decentralized Finance Protocols to EVM bytecode using the K framework

Tiago Luis Barbosa
tiago {dot} l {dot} barbosa@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2021

Abstract

Blockchains enable a democratic, open, and scalable digital economy based on decentralized distributed consensus without a third-party trusted authority. They can be developed with a distributed execution environment, called virtual machines, which enables executing arbitrary programs, called Smart Contracts. On Ethereum, the Ethereum Virtual Machine is the global virtual machine whose state is stored and agreed upon by all network participants. In recent years, the amount of Smart Contracts deployed on Ethereum has rapidly increased. The composability that the Ethereum Virtual Machine offers has led to an emerging ecosystem of financial applications and protocols, termed Decentralized Finance (DeFi). Because these protocols secure vast amounts of capital, bugs or unintended behavior frequently leads to catastrophic financial losses for users. Consequently, formal verification methods for these protocols have been a recent focus of research. Among those methods, the K Framework is one of the most sophisticated and capable frameworks for defining and verifying programs. It allows defining arbitrary executable specifications of protocols as well as directly executing their bytecode with the KEVM implementation. In this dissertation, we aim to improve the security of these protocols. To achieve this we focus on MakerDAO, a pioneer protocol in DeFi, as well as its high-level K specification. We introduce new documentation for this protocol and we extend the high-level specification with a new liquidations module and a non-trivial system invariant. Finally, we develop and demonstrate refinement methods that enable refinement proofs which connect an high-level protocol's specification with the protocol's bytecode implementation.
Keywords: Blockchain, Ethereum, Smart Contracts, Formal Verification, Decentralized Finance

1. Introduction

1.1. Motivation

Blockchain technology possesses a wide range of attributes that make it a very appealing and efficient solution to a vast variety of issues and obstacles. Arbitrary programs that exist within a blockchain network, called Smart Contracts, inherit some of the Blockchain characteristics such as immutability, unforgeability and irrepudiability which are desired in many applications. Despite the demand for these attributes, they may also be considered weaknesses in some scenarios. An existing flaw in an arbitrary program that exists within a blockchain network, a Smart Contract, may be found and consequently exploited, or unexpected non-reversible errors in user-defined logic may occur.

As interest rises in Blockchain technology and the possibilities it entails grow [1] activity in decentralized ledgers increases its pace [2]. The web of Smart Contracts and their interactions present in Ethereum keeps increasing in complexity as programmers create protocols, groups of bundled Smart Contracts that serve a certain purpose [3, 4]. These protocols serve many different purposes, whether it be lending and borrowing of capital, decentralized exchanges, or insurance, etc. Due to the nature of composability of these Smart Contracts and their critical purposes, hard to spot errors in these contracts lead to catastrophic scenarios, which previously and currently results in immense capital lost. These exploits happen on a regular basis and the figure in appendix **??** discloses the list of major DeFi 2020 hacks. Previous errors and exploits such as the famous DAO reentrancy attack [5] are decreasing, and current hacks are now non-intuitive, deeply semantic related, and require high-level of expertise.

Despite the fact that some improvements on these protocols can be made after deployment, by redeploying certain Smart Contracts and configuring the remainder to use the most recent ones, there might still be persistent errors, and whilst these errors are not fixed the likelihood of an exploit by an ill intentioned actor increases over time. To de-

velop trust in Smart Contracts even before they are deployed, traditional verification methods such as symbolic analysis approaches, including fuzzing [6], static analysis, and regular code testing coverage are regularly studied and implemented. However these do not offer complete reliability on semantic properties and are bound by computation power and execution time, frequently generating false negatives.

By virtue of this escalation in sophistication in Smart Contract protocol architecture and academic advances in mechanism design [7] it has become necessary to verify that constructed abstract high-level models of these systems adhere to their concrete implementation and vice-versa.

1.2. Objectives

Taking into account the issues presented and the insufficiency of investigation on Formal Verification applied to this domain, we focus our research on high-level modeling of Smart Contract systems, as well as refining these with their concrete bytecode implementation. In particular we aim to introduce detailed approaches on how to properly break down Smart Contract protocols, modeling these systems with regards to different layers of abstraction, and refinement techniques between abstraction layers for use in the K framework.

2. Background

2.1. Blockchain

In the area of distributed systems byzantine fault tolerant protocols for decentralized consensus have always been a topic of high interest. Consequently, the first distributed decentralized consensus mechanism achieved, known as *blockchain*, was introduced by Satoshi Nakamoto in his Bitcoin whitepaper [8] that has subsequently become remarkably influential. Due to its versatility, blockchain related papers are increasing [9] and new employments of this technology emerge regularly.

Blockchain's qualities as an immutable, decentralized, and efficient settlement layer allow for the consensual and deterministic execution of arbitrary programs, referred as *Smart Contracts*, on which all of the network participants agree, without the need for a centralized source of trust, on the network states before, during and after their execution.

2.2. Smart Contracts

The concept of Smart Contracts was invented by Nick Szabo, in 1996, [10] long before blockchain existed, and it was described as "a set of promises, specified in digital form, including protocols within which the parties perform on these promises".

As a mean to fully extract benefit from blockchain's characteristics, they are conceptualized and deployed with a program execution layer. This abstract program execution layer abides by consensus rules. Therefore, programs meant for execution are written and interact following the blockchain rules, on which different programming languages, depending on the goal, may be implemented.

Furthermore, Bitcoin's non-turing complete Bitcoin Script [8] was the first Smart Contract language to exist. It has limited expressiveness and computational power, which is considered a choice in its implementation. Subsequent work from Vitalik Buterin and Gavin Wood proposed the use of a general-purpose Turing complete Smart Contract language [11], which materialized as Solidity, Ethereum's [12] Smart Contract language in 2015.

2.3. Ethereum

As we deepen our knowledge, a common and intuitive question arises due to Turing's halting problem: if the Smart Contract language is turing complete how can it be guaranteed its execution termination in the blockchain?

Many blockchains such as Ethereum use *gas* as a way of providing metaphorical fuel to the execution of Smart Contracts, thus computation is always bounded by the gas limit associated with its execution.

Moreover, another crucial question in this subject is: how can it be guaranteed that the execution of Smart Contracts and its results affecting the blockchain state are what was intended and designed by Smart Contract programmers?

This question leads to the area of study known as *Formal Verification*.

2.4. Decentralized Finance

Ethereum's Smart Contract execution layer, termed Ethereum Virtual Machine or EVM, not only maintains the state and code of all the Smart Contracts deployed on Ethereum, but also supports interoperability between them. This composability between Smart Contracts results in extremely sophisticated protocols with complex non-trivial behaviors. A vast majority of these protocols are financial applications, and so the combination of them is called *Decentralized Finance*. Currently, new financial primitives made possible by the composability of blockchain's virtual machines are heavily researched, and although these represent a breakthrough in financial tooling they become increasingly troublesome to verify and prove correct.

2.5. Formal Verification

Before comprehending formal verification it is necessary to first understand that a programming language is divided into its syntactic and semantic properties. While syntactic properties relate to how the program is written, the semantic ones are asso-

ciated with the program's behavior, in essence what the program means and what is does. This process, known as formal verification, is the only method to inspect, understand and guarantee the correct and intended behavior of a program. This means that one can structure the property that wants proven as a mathematical specification in the desired logic system and soundingly prove that a program has or has not such property. Moreover, mathematical models and property specifications can be designed using different types of abstractions, logics, axioms and formalisms which are regularly the focus of study of programmers interested in formal verification.

As mentioned previously, the immutability nature of blockchain makes Smart Contract properties and execution correctness crucial for further development and application of this technology.

3. K Framework

3.1. Introduction

Framework languages, used to define and reason about programming languages, must be: user-friendly, so language designers can use these frameworks to create and experiment; mathematically rigorous, so that the language definitions can be used to support formal reasoning about programs; modular, such that they can be extended with new features without needing to revisit existing features; expressive, in order to easily define programming languages with any number of complex features. The K framework [13] was created with this design in mind and reflects these characteristics.

K's goal is to distinguish specification of analysis tools from specification for particular programming languages or other models, which makes specifying both analysis tools and programming languages easier [13]. Furthermore, in order to trust the results, the generic tools instantiated for any given language must be correct-by-construction, and should also be efficient, so that there is no need to implement language-specific tools, which in turn reduces human effort and time-consumption.

3.2. Reachability Logic

K's foundation, Reachability Logic, is a logic for symbolically reasoning about possibly infinite transition systems [14]. This logic is equipped with a sound and nearly complete inference system which allows efficient implementations.

3.3. Proving with Reachability logic

Reachability logic is extremely advantageous as it can be used to prove statements, known as reachability claims, syntactically structured as $\phi \Rightarrow \psi$, where ϕ and ψ are formulae in static logic. The static logic applied is a subset of Matching Logic [15].

Matching logic formulae are called patterns and may be viewed as state configurations, using symbolic variables for unknown values. Restrictions to these patterns might also be applied implying that it is not characterized by the set of possible configurations that match it, but by the subset of configurations in this set that also respect said restrictions. Code may also be represented as algebraic data in Matching Logic which causes the patterns ϕ and ψ to regularly incorporate it. Figure **??** presents the formal axioms that comprise the reachability logic system [16].

3.4. KEVM

As explained previously, by virtue of the K Framework's structure, one can provide a formal definition that can be mechanically transformed to a reference interpreter for the EVM and benefit from a range of analysis tools.

In 2017, [16] a formally rigorous executable instance of the EVM semantics in the K framework was implemented, covering all of the EVM instructions. Known as KEVM, this implementation is open-source and can be found on the respective repository [1].

4. MakerDAO

MakerDAO is a Decentralized Finance protocol. The project started in 2015 [17] when Rune Christiansen developed the MakerDAO Protocol on the Ethereum blockchain after first describing the system on a reddit post [18]. Christiansen's vision was for a decentralized financial system to be managed by its users. This would allow borrowers greater control over their assets, even in difficult economic conditions such as periods with high inflation. Firstly, the protocol started as a DAO created by Rune and a few other developers. Then, later, it was developed under the auspices of the Maker Foundation, and recently has come full circle by returning to a self-governed and self-operating DAO. The MakerDAO is made up of every type of entities, whether they be single individuals or groups. These entities are from all parts of the globe and own MakerDAO's governance token MKR, which gives them the right to vote for all kind of changes in the network.

In fact, three acknowledged versions of the protocol have been deployed on the Ethereum blockchain. Firstly, there was ProtoSai, then SAI, which was a single-collateral DAI. Both these versions have been deprecated. The last and current version is known as MCD.

As mentioned previously, this thesis aims to refine high-level systems models with their Smart Contract bytecode implementation counterparts, as a

[1] https://github.com/kframework/evm-semanticshttps://github.com/kframework/evm-semantics

result we decomposed the MakerDAO protocol into its goal, the mechanisms that can achieve it, as well as Smart Contract implementation. In this extended abstract, we provide an example of the decomposition of the protocol that was developed in the dissertation.

4.1. Intent

Motivation MakerDAO is a decentralized organization that aims to bring stability to the extremely volatile cryptocurrency market.

Goal The MakerDAO Protocol is a complex system which at a very high-level serves a specific primary purpose: create an asset, known as DAI, that has a particular economic value in terms of some reference asset, the United States Dollar. This is the definition of a *pegged* currency. DAI is also a *stablecoin*, since it is pegged to a *fiat* currency considered stable.

4.2. Mechanisms

DAI's stability is achieved through a dynamic system of collateralized debt positions, autonomous feedback mechanisms and incentives for external actors.

The MakerDAO Protocol employs a two-token system, DAI and MKR.

DAI a collateral-backed stablecoin that offers price stability.

MKR the governance token that is used by stakeholders to maintain, upgrade or stop the system as well as manage DAI. MKR token holders are the decision-makers of the MakerDAO Protocol, supported by the larger public community and various other external parties.

Any actor with the required knowledge can freely participate in any role, and they can occupy multiple roles at once.

4.3. Collateralized Debt Position

DAI is created by locking assets with economic value inside the protocol. Then, the system issues DAI as debt against such assets creating what is designated as a *collateralized debt position*, a CDP. The owner cannot retrieve these assets without paying back the DAI they owe. Each asset has its own associated system parameters. In fact, multiple representations of the same asset, each representing different parameters, can be found within the system, which is what defines *Ilks*. *Ilks* are the combination of asset and parameter types that determine collateral types. Each *ilk* is assigned a unique label.

Actor relationship to CDPs The system recognizes separation between accounts to which balances and positions are assigned. As a result an external actor such as a person or institution is able to control multiple accounts simultaneously.

Actor ownership of CDPs Given that the key terminology and basic concepts for this topic were presented previously, we now have the ability to explain the basic balances, and their position structures. Balances are made of collateral and DAI. Each user has a balance in each *ilk*. Every user also has a DAI balance. A position, formally termed vault or *urn*, requires an *ilk* and an account and refers to the amount of *ink*, the quantity of collateral locked, and the issued DAI, which is represented as debt. To complete our elementary descriptions, we add *vice*. *Vice* is debt, DAI, issued by the system that is not backed up by a CDP.

Fundamental invariant of the system Given that we've explained some dynamic properties now we can define the first non-trivial system invariant, and the most relevant one. It states that the sum of all issued DAI for every account must equal *vice* plus the sum of every debt for every type of *ilk* and account. This is known as **"The Fundamental Equation of DAI"**, represented in equation 1.

$$\sum_{u \in U} dai_u = vice + \sum_{i \in I, u \in U} dbt_{iu} \qquad (1)$$

Action dynamics of CDPs In this topic, it's important to begin by introducing the protocol's dynamics starting with the rules for balances and positions. The *ilk* balances can be changed by transferring assets in and out of the protocol, transfers between accounts within the protocol, or by adding or removing collateral from a position. DAI, on the other hand, cannot meaningfully flow into or out of the system since it is defined by it. For every *ilk* the system must have knowledge of some market price for that asset in terms of DAI. In fact, there's a per-*ilk* time-varying parameter that expresses the minimum ratio of collateral market value to debt known as the collateralization ratio. Moreover, a position is said to be safe if, at a certain time point, the debt of an *ilk*, for a given account, times the collateralization ratio is equal or less than the *ink* amount times the market price of the collateral.

Action Permissions These actions are restricted to certain external actors in the system in the following way: the change of DAI balances over time must respect the conservation

relationship defined earlier in **"The Fundamental Equation of DAI"**.

Furthermore, the debt of positions can be manipulated only according to certain rules that are described as follows. The *ilk* balances gem_{iu}, the sum of all vault's *ink* of a certain *ilk* i and account u, can be changed by transferring assets in and out of the protocol, transfers between accounts within the protocol, or by adding or removing collateral from a position.

Time dynamics of CDP's In this subject, the concept of a stability fee is crucial. This describes the time evolution of a position's debt. As stability fees accumulate, the additional debt is balanced by assigning an equivalent amount of DAI to one or more accounts in a special set of accounts, which belong to the system, in order to respect the **"Fundamental Equation of DAI"**.

Stability fee Incentives Stability fees can serve at least three different economic functions within the system: motivate the creation or destruction of DAI as needed to close the gap between the market price and target price, offsetting the danger posed by risky or volatile assets held as collateral, and providing financial capital necessary for the operation of the system (as well as profit to its stakeholders if income exceeds costs).

This summary of the decomposition of the different mechanisms that the MakerDAO protocol uses is further extended in the complete dissertation.

5. MakerDAO K specifications

The developed and refined models of the MakerDAO system that we document and improve on this document are formalized in the K framework, which, as explained previously, not only provides a formal semantics engine for analyzing and proving properties of programs but also allows developers to define models that are mathematically formal, machine-executable, and human-readable at different levels of abstraction.

The system specifications are formal, defining contracts as configuration patterns, and specifying system behaviors as transitions over patterns, modeled as K's rewrite rules. Specifications are executable in the K framework following the defined rules, which due to the benefits provided by the K framework's design immediately produce an execution engine for the protocol. When modeling systems, executable specifications enable running simulations on different levels of abstractions, which help prototyping and debugging different designs during the development process and after their deployment as well.

5.1. High-level model

This model abstracts the protocol's implementation on the Ethereum Virtual Machine and details the high-level mechanisms that comprise the MakerDAO system.

This high-level model may be used as a canonical specification for model-based test generation and for validating other implementations. Additionally, this method seamlessly facilitates the gradual improvement of the protocol's formal design without needing to alter the concrete Smart Contract implementation.

Finally, the executable high-level specification of the MakerDAO system in K can be immediately subjected to K's suite of reachability, model checking and theorem proving tools, promoting and aiding the verification of different formal analysis.

In order to further comprehend this topic, below, we give additional information regarding the abstraction level of this model. Alternatively to previous sections, where we explained how the system works, we now characterize where this high-level formalization of the MakerDAO system stands concerning abstraction with regards to the reference Smart Contract implementation, clearly delineating the similarities and differences between them.

Similarities:

- Notion of accounts, external actors, *Externally Owned Accounts* in the EVM
- Non-concurrency of state manipulation
- Model is composed by files that use the same naming conventions as in the Smart Contract implementation with similar, but abstract, state manipulation
- Naming of data structures

Differences:

- No bytecode manipulation
- Typing of variables is independent of implementation
- The configuration of the model only references the MakerDAO system and not the complete EVM state
- Notion of time not provided by the EVM

Summary:

This high-level model of the system could describe the MakerDAO system on another blockchain that presents similar characteristics to the Ethereum Virtual Machine such as non-concurrency and an account-based computational model. However, without the finer-grained characteristics of Ethereum or the EVM, such as typing

and consensus mechanisms, which is faithful to the high-level design of the system that is already built on top of these assumptions. In conclusion, it is a high-level representation of the Solidity contracts that comprise the MakerDAO protocol.

We must emphasize that the chosen level of this system specification in the spectrum of abstraction not only allows for testing the model with regards to high-level properties, such as Finite State Machines, but also enables using some of the concrete tests that are part of the implementation, further validating the design decision.

Given that the reader now comprehends various crucial aspects of this high-level model, we can proceed to summarize the components of the K MakerDAO model specification which is open-source and can be found at https://github.com/makerdao/mkr-mcd-spechttps://github.com/makerdao/mkr-mcd-spec.

5.2. Low-level model

This model of the system faithfully reproduces the system's implementation on Ethereum as it uses the compiled Solidity bytecode that is available on-chain and executes it on the reference EVM implementation in K, the KEVM, already detailed in the respective Section 3.4. Moreover, using directly the compiled bytecode eliminates the need to trust the Solidity compiler, maximizing the functional guarantees provided by its verification.

The bytecode is made available for use in the KEVM by inserting it in K syntax.

6. High-level Specification Extension

In this chapter we present our contributions to the high-level model of the MakerDAO protocol.

6.1. Liquidations 2.0

The MakerDAO protocol uses liquidations of uncollateralized vaults as a mechanism for maintaining DAI's peg to the dollar [19]. During the initial phase of the development of this thesis, the liquidations system of the Maker protocol was upgraded replacing the old liquidations with new more efficient ones. This improvement to the system, known as "Liquidations 2.0", redesigned liquidations by replacing the previous English auction with a new Dutch auction system. This upgrade to the protocol was led by the motivations to reduce: the reliance on DAI liquidity, the likelihood of auctions settling far from the market price, and the barriers to entry. An in-depth view of the research and analysis that resulted in this change can be found on the MakerDAO governance website [2] and the complete Improvement Proposal, detailing implementation, can be also found online [3].

We used this opportunity to familiarize with the inner workings of the K framework and the KEVM, understand how the MakerDAO protocol works, from its mechanism design to implementation, and gain thorough insight of the publicly available codebases that model the MakerDAO protocol in the K framework, both at the high-level and low-level specifications, all while contributing directly to the continuous development of all the aforementioned technologies.

6.2. Liquidations 2.0 High-level K Specifications

Liquidations 2.0 introduces three new contracts to the MakerDAO protocol Ethereum implementation, each responsible for a certain key component of the liquidations system. These three contracts replace the two previous contracts in charge of liquidations, *cat.sol* and *flip.sol*. Each of the new contracts, *dog.sol*, *clip.sol* and *abaci.sol*, provide different functionality to the system, described below:

The *dog.sol* contract is responsible for liquidating vaults and initiating a Dutch auction to sell the vault's collateral for DAI. The liquidation is triggered by an external user who signals that a particular vault is uncollateralized. After verifying if the vault is indeed uncollateralized, it then also decides whether the vault should be entirely liquidated or whether it should only be performed a partial liquidation.

The *clip.sol* contract is responsible for the Dutch auctions that receive a certain amount of DAI for the confiscated collateral of the liquidated vault. After a Dutch auction has been initiated, external users can bid with DAI to buy the respective collateral. Since the auction style is Dutch there might be no bids for the collateral and when a price threshold is met, then an external user is encouraged to instruct the contract to reset the auction.

Lastly, the *abaci.sol* contract is responsible for calculating the price of the collateral to be sold, at each time step on the Dutch auction. It is the responsibility of the *clip* contract to query the abacus whenever it needs a new price for the currently auctioned collateral.

Each of these contracts was specified and added to the high-level K model of the Maker protocol, extending the codebase with these new files. Respectively, the *dog.sol* was formalized into *dog.md*, *clip.sol* into *clip.md*, and *abaci.sol* into *abaci.md*, all of which can be found here https://github.com/makerdao/mkr-mcd-spec/pull/250https://github.com/makerdao/mkr-

[2] [https://forum.makerdao.com/t/a-liquidation-system-redesign-a-pre-mip-discussion/2790]https://forum.makerdao.com/t/a-liquidation-system-redesign-a-pre-mip-discussion/2790

[3] [https://forum.makerdao.com/t/mip45-liquidations-2-0-liq-2-0-liquidation-system-redesign/6352]https://forum.makerdao.com/t/mip45-liquidations-2-0-liq-2-0-liquidation-system-redesign/6352

mcd-spec/pull/250. All of the high-level behavior described above was captured in the K high-level specification of these contracts, representing a suitable abstraction on par with the rest of the codebase and following the conventions defined in the section highlevel.

In order to integrate these files into the system, some changes to the execution framework and data types of the system had to be made to accommodate new behavior and data manipulation.

Liquidations 2.0 introduced two new modifiers to the system, a reentrancy call prevention mechanism called *lock*, and a four stage liquidation circuit breaker mechanism called *stop*.

The locking mechanism was formalized in the general execution framework of the high-level specification by introducing the modifier at the call boundaries, faithfully capturing the intended high-level behavior of the mechanism, correctly locking and unlocking the intended rules, making reentrant calls impossible. The code necessary to do so has already been shown and explained when detailing the high-level model.

The circuit breaker mechanism only applies to the *clip* contract and, therefore, it was formalized using a new sort, consisting of terminal strings, and then defining the rules necessary for comparisons between members of the sort, which yield a Boolean value. This formalization captures the intended behavior of the modifier, while only requiring the intended comparison to be made at the *requires* part of a rule.

It was also necessary to add some data conversion rules between arithmetic types but without precision loss, which would make the system's behavior to be wrongly captured by the specification.

6.3. Fundamental Equation of DAI

We extended the high-level MakerDAO K specification with a non trivial property of the protocol modeled as a Finite State Machine. This property is the *Fundamental Equation of DAI* which, as mentioned earlier, is an invariant of the dynamic system that states the following: The Sum of DAI of all users must be equal to *vice* plus the sum of debts of all *ilks* of all users, represented in equation 1.

In order to express this property it was necessary to extend the specification's measured events to encompass the sum of the total DAI issued over the vat.

As it is shown in the repository [4], even though this is a non trivial property of the system, it is formalized over the high-level specification in a succinct and clear way. This difference in the expressiveness of properties between various abstraction levels of a system's specification reinforces the motivation behind refinement proofs, which are discussed on the next chapter.

[4] https://github.com/makerdao/mkr-mcd-spec/pull/250https://github.com/makerdao/mkr-mcd-spec/pull/250

7. Refinement Proofs

Refinement is the process of moving from an abstract specification, termed the *model*, to a concrete specification, termed *implementation*. *Refinement proofs* demonstrate that the abstract model accurately captures behaviors of the concrete implementation. Note that this allows the implementation to exhibit behaviors not captured by the model. To disallow this, one can do a refinement proof in the other direction: show that the implementation accurately captures the behavior of the abstract model. An *equivalence proof* can be constructed by showing refinement proofs in both directions.

7.1. Motivation

When formalizing system's designs at higher-levels of abstraction it is always desirable to guarantee that its behavior is captured by the implementation. The MakerDAO high-level specification of the system is no exception. When it was first formalized, its creators, the MakerDAO and Runtime Verification teams at the time, sought to ensure that certain behaviors of this model were present in the implementation. This was achieved by: firstly, executing the high-level model, checking for the violations of properties and state updates. During execution, the high-level model collects the sequence of contract interactions and state changes. Afterwards, it constructs a Solidity unit test with the equivalent sequence of calls and assertions about the state changes. Finally, the generated Solidity test is ran against the Solidity implementation to validate conformance between the model and the implementation on that execution trace.

However, this implementation of the refinement does not directly refine the high-level specification with the bytecode implementation. It requires trusting the external python library and also the Solidity compiler for the tests. Moreover, this procedure could be described as *refinement testing*, as it only ensures that the set of behaviors exhibited when fuzzing the high-level model are in fact captured by the implementation. A sound proof of refinement between the high-level specification and bytecode implementation in the K framework removes the necessity of using external tooling, trust in the Solidity compiler, and also ensures that the implementation captures all of the high-level specification behaviors.

The method described here is an attempt to prove refinement of the high-level model to the implementation and is pioneer work in the area of DeFi [20], an up to date model, and until now on the forefront

of verification in Decentralized Finance.

Additionally, as discussed in the previous chapter, when documenting the MakerDAO protocol and finding interesting properties to model over the high-level specification, it became clear that sound refinement proofs were necessary.

Having this in sight, in the following sections we present methods and examples that not only make this sound refinement in the K framework possible but also intuitive and approachable.

7.2. Refinement Methods

The refinement technique presented in this thesis uses the MakerDAO protocol as an example, but it can be modified with low overhead to refine other protocol's high-level models to EVM bytecode.

In order to formalize the refinement proofs we must first define how will the refinement method between the two specifications work.

Our refinement method is based on cut-bisimulation, introduced by Daejun et al. [21]. Cut-bisimulation allows two programs to semantically synchronize at relevant "cut" points, but to evolve independently otherwise. We now outline the cut-bisimulation mechanism and correctness guarantees for our refinement model.

Inspired by the cut-bisimulation method explained above we model our refinement proofs using a slightly different technique. Instead of specifying pairs of cuts on which the simulations states should be equivalent throughout execution, we start with a symbolic state in the high-level model, construct a refined state in the implementation directly, execute the low-level state symbolically to completion, then map the final state back to the high-level model where we check it for correctness. By providing a constructive translation between the model states, we can refrain from having to execute both models to prove refinement.

Summarizing, our refinement proofs follow the ensuing procedure:

1. Start with a transition in the high-level model, which consists of an initial symbolic state and a final symbolic state.

2. Initiate a transaction on the high-level specification.

3. Symbolically execute the implementation state to completion.

4. Map the final symbolic implementation state back to a model state, proving that it is identical to the final state described by the high-level transition.

5. Restart high-level execution on item 2 until the entire behavior specification is proven.

In the refinement process, by symbolically updating the storage of the KEVM we can then demonstrate that these updates are equal to the symbolic updates defined on the high-level specification. This ensures that a subset of possible behaviors of the implementation is captured by the high-level model. Repeating this process for every possible high-level behavior proves that every behavior in the high-level specification is captured by the implementation, meaning that the high-level behavior is a strict subset of the bytecode implementation. We note that this refinement model does not state any conclusion about additional behavior in the implementation not contemplated in the high-level specification.

Implementing this refinement can be divided into two major issues. Firstly, how the transactions should be refined to the bytecode, execution refinement. Secondly, how the symbolic storage updates should be verified equivalent, state refinement.

As we can observe the refinement between specifications is non-trivial. Therefore, throughout the rest of this chapter we break apart the issues presented and demonstrate our solutions for the concrete implementation details of the proposed refinement model.

7.3. Execution Refinement

The execution refinement between the high-level specification execution framework and the KEVM can be re-used with minor adaptations for other low-level models.

Model Configuration

At the bottom of the presented high-level specification configuration we include the KEVM configuration, $\langle kevm \rangle$. This allows accessing KEVM state and concurrent state manipulation between both specifications. Additionally, we also add a helper configuration which translates between users in the high-level model and accounts in EVM, $\langle mcd - accounts \rangle$. This represents the initial configuration on step 1.

Data Structures

As discussed previously, it is necessary to introduce a configuration that enables translating between users in the high-level specification and accounts in EVM. This configuration is a pair consisting of a single user and account. It ties both of these together and each unique pair is identified by its high-level specification user id.

Specification Transition Functions

As mentioned in step 3 of the refinement proofs methodology it is necessary to refine transactions

from the high-level specification into equivalent bytecode transactions accepted by the EVM. In order to achieve this we could manually specify a translation for every high-level transaction, but it would not be modular to use as refinement for other specifications other than the MakerDAO one.

7.4. State Refinement

In the refinement technique declared above we must be able to verify that the symbolic updates to storage made by the low-level specification match the storage update claims of the high-level model.

We implement a new technique introduced in this dissertation. This technique allows manipulating abstract storage when executing bytecode on the low-level implementation. It abstracts the EVM storage mapping, making it possible to define arbitrary storage configurations in K. It trades off being able to execute the KEVM over typical bytecode storage for the possibility of directly verifying that symbolic storage updates of the low-level specification match the expected high-level model claims, proving the refinement correct. In our refinement example, we substitute the KEVM representation of storage with the same structured representation of the Smart Contract storage from the high-level model and adapt the KEVM storage reads and writes to work over this representation. This allows to trivially check that symbolic storage updates match claims, while minimizing the changes to KEVM necessary in order for it to properly read and write from storage. This new technique results in a much more practical method for refining large codebases such as the MakerDAO protocol.

The validity of the storage updates performed over an abstract storage assumes the correctness of the storage layout regarding its equivalence to the actual bytecode storage. In Solidity, variables present in the storage are declared at the beginning of a contract along with their identifier and data type, making it trivial to confirm that they are properly expressed in the high-level configuration's cells. The Solidity compiler uses hashed locations to ensure that there are no data collisions [22]. At the moment, every Solidity contract and Solidity developer assume that this statement is true, and although bugs in other EVM bytecode compilers like the Vyper compiler have been found in previous work with the K framework [23, 24, 25] it is still an accepted assumption in the Ethereum community. This trust model implies that the abstract storage we are using does not use additional trust assumptions to ensure correctness.

8. Analysis

We now analyze the extension of the high-level MakerDAO model and the new abstract storage technique.

8.1. Liquidations 2.0

The Liquidations 2.0 module was properly specified in the high-level model [5], maintaining the same abstraction from the Solidity implementation as the rest of the codebase. The codebase was not extended with randomized concrete tests for this module as refinement proofs and formalization of system properties were prioritized.

8.2. Fundamental DAI Equation

Following the definition of the *Fundamental DAI Equation* invariant as a Finite State Machine on the high-level model in Section 6 we executed the already defined randomized concrete tests present in the specification [6], explained in Section 5. Running this test suite ensured that for the set of behaviors tested the non-trivial invariant of the system remained true.

8.3. Abstract Storage

In the complete dissertation we present an example of a reachability claim successfully proven using KEVM with the abstract storage model discussed in section 7.

9. Conclusions

This chapter summarizes this dissertation's major contributions and addresses future research and development.

9.1. Contributions

This thesis provides several contributions on the directions described in Section 1.2.

Concretely, our research used MakerDAO as the Smart Contract system to model and the K framework to formally specify this system, both at high and low levels of abstraction. Firstly, we extended MakerDAO's documentation, detailing it's goal, mechanisms that are designed to achieve it, and implementation of such mechanisms. Afterwards, we created documentation for the high-level K specification of MakerDAO. Subsequently, we extended MakerDAO's current high-level K specification with the Liquidations 2.0 module in order to correctly represent the currently deployed system's architecture and a non-trivial property of the system. Later, we refined this high-level model to match the Smart Contract EVM representation of the system in the same semantic framework K, leveraging the existing implementation of EVM in K, introducing novel K modelling techniques to achieve this result. Consequently, this refinement leads to being certain that proofs over the high level model of the system

[5] https://github.com/makerdao/mkr-mcd-spec/pull/250https://github.com/makerdao/mkr-mcd-spec/pull/250

[6] [https://github.com/makerdao/mkr-mcd-spec/tree/master/tests]https://github.com/makerdao/mkr-mcd-spec/tree/master/tests

are also correct over the bytecode implementation of the system.

Our main contributions are summarized as follows:

- We improved MakerDAO's documentation, focusing on refinement proofs, in Section 4.
- We introduced documentation for MakerDAO's high-level K framework specification, in Section 5.
- We expanded MakerDAO's high-level K specification to include the new Liquidations 2.0 module [7], in section 6.
- We formalized a non trivial high-level property over the high-level MakerDAO specification [8], in Section 6.
- We developed the software necessary for refinement proofs between the high-level MakerDAO specification, modeled in K, and the EVM bytecode implementation, [9] [10] [11], im Section 7.
- We created a blueprint for refinement proofs between high-level specifications of system's models and their implementation that can be generalized for different semantics, in Section 7.

9.2. Future Work

As blockchain and Smart Contract development are rapidly growing industries it is necessary to constantly keep improving research. Direct directions for future work are:

Finish refining the codebases The refinement techniques presented in this thesis and working examples of their utilization show the benefit and practical use of it in this codebase. However, the full refinement proof of all the behaviors of the high-level specification has not yet been completed and will continuously be implemented in the upcoming months.

Continue modeling protocol properties In this document we provided documentation of MakerDAO that clearly separates design and implementation. Continuing to reason about the mechanism design of the protocol will lead to the formalization of more system invariants and properties which can be specified on the high-level model and refined to the implementation.

Automate the high-level k model directly from contract source The execution abstraction of the presented MakerDAO high-level K specification can be re-used by other protocols with minimal effort. A tool that attempts to automate as much as possible the creation of a high-level model of a protocol directly from Solidity source may also be developed.

Automate the refinement proofs Most of the refinement technique we presented can be re-used with a similar high-level K specification. If a tool that automatizes the specification of a high-level model from Solidity is created then defining the appropriate rules for each particular contract storage lookup and writes should be trivial to implement as well.

Ensure reverting behavior in implementation Further research should look to ensure that implementation behaviors not captured by the high-level model should lead to reverting states, not affecting storage.

Summing up, if research continues in this direction, the Ethereum ecosystem will benefit from being able to automatically specify and refine high-level models of Solidity Smart Contracts with their produced bytecode and vice versa. Advances in this area leads to immensely improving quality standards of Decentralized Finance protocols, reducing economic risk for users and enabling the secure design of new financial primitives.

[7] https://github.com/makerdao/mkr-mcd-spec/pull/250https://github.com/makerdao/mkr-mcd-spec/pull/250

[8] https://github.com/makerdao/mkr-mcd-spec/pull/250https://github.com/makerdao/mkr-mcd-spec/pull/250

[9] https://github.com/makerdao/mkr-mcd-spec/tree/mcd-to-kevm-transactionshttps://github.com/makerdao/mkr-mcd-spec/tree/mcd-to-kevm-transactions

[10] https://github.com/makerdao/mkr-mcd-spec/tree/simple-refinementhttps://github.com/makerdao/mkr-mcd-spec/tree/simple-refinement

[11] https://github.com/makerdao/mkr-mcd-spec/tree/step-subsortshttps://github.com/makerdao/mkr-mcd-spec/tree/step-subsorts

References

[1] W. Chen, Z. Xu, S. Shi, Y. Zhao, and J. Zhao, "A survey of blockchain applications in different domains," *Proceedings of the 2018 International Conference on Blockchain Technology and Application - ICBTA 2018*, 2018. [Online]. Available: http://dx.doi.org/10.1145/3301403.3301407

[2] A. Anoaica and H. Levard, "Quantitative description of internal activity on the ethereum public blockchain," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2018, pp. 1–5.

[3] F. Schär, "Decentralized finance: On blockchain- and smart contract-based financial markets," 03 2020.

[4] D. Zetzsche, D. Arner, and R. Buckley, "Decentralized finance (defi)," *SSRN Electronic Journal*, 01 2020.

[5] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," 03 2017, pp. 164–186.

[6] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: fuzzing smart contracts for vulnerability detection," *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Sep 2018. [Online]. Available: http://dx.doi.org/10.1145/3238147.3238177

[7] A. Mamageishvili and J. C. Schlegel, "Mechanism design and blockchains," *CoRR*, vol. abs/2005.02390, 2020. [Online]. Available: https://arxiv.org/abs/2005.02390

[8] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[9] L. Zhou, L. Zhang, Y. Zhao, R. Zheng, and K. Song, "A scientometric review of blockchain research," *Information Systems and e-Business Management*, 02 2020.

[10] N. Szabo, "Smart contracts : Building blocks for digital markets," 2018.

[11] V. Buterin, "A next-generation smart contract and decentralized application platform," 2015.

[12] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[13] G. Rosu, "K framework - an overview," https://runtimeverification.com/blog/k-framework-an-overview/, 2018.

[14] A. Stefanescu, Ş. Ciobâcă, R. Mereuta, B. M. Moore, T. Serbanuta, and G. Rosu, "All-path reachability logic," *CoRR*, vol. abs/1810.10826, 2018. [Online]. Available: http://arxiv.org/abs/1810.10826

[15] G. Rosu, "Matching logic - extended abstract," in *26th International Conference on Rewriting Techniques and Applications, RTA 2015*, ser. Leibniz International Proceedings in Informatics, LIPIcs, M. Fernandez, Ed. Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, Jun. 2015, pp. 5–21, 26th International Conference on Rewriting Techniques and Applications, RTA 2015 ; Conference date: 29-06-2015 Through 01-07-2015.

[16] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Roşu, "Kevm: A complete formal semantics of the ethereum virtual machine," 07 2018, pp. 204–217.

[17] R. Christensen, "Makerdao has come full circle," https://blog.makerdao.com/makerdao-has-come-full-circle//, 2021.

[18] ——, "Introducing edollar, the ultimate stablecoin built on ethereum," https://www.reddit.com/r/ethereum/comments/30f98i/introd

[19] MakerDAO, https://docs.makerdao.com/, 2021.

[30] R. Burton, "Formal verification, virtual hardware, and engineering for blockchains," https://medium.com/balance-io/formal-verification-virtual-hardware-and-engineering-for-blockchains-51d07abdc934, 2019.

[21] D. Park, T. Kasampalis, V. S. Adve, and G. Rosu, "Cut-bisimulation and program equivalence," 2020.

[22] E. Foundation, https://docs.soliditylang.org/en/v0.8.9/internals/la

[23] D. Park, https://github.com/ethereum/deposit$_c$*ontract*/*issues*/27,

[24] ——, https://github.com/ethereum/deposit$_c$*ontract*/*issues*/28, 201

[25] ——, https://github.com/ethereum/deposit$_c$*ontract*/*issues*/38, 201