TÉCNICO LISBOA

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

Tiago Luís Sardinha Bernardo Cabral Barbosa

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Fernando Peixoto Ferreira
Prof. Alexandra Sofia Ferreira Mendes

Examination Committee

Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. Pedro Miguel dos Santos Alves Madeira Adão

November 2021

Acknowledgments

I want to show appreciation to those who accompanied me during this journey to become the person I am today. The fiercest battles are not fought alone. Special thanks to my mom, dad, brother and sister. With regards to this thesis special thanks to Professor João Ferreira and Connoisseur of the fine arts of Formal Verification Everett Hildenbrandt. 3 σ gmi

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

Resumo

Blockchains possibilitam uma economia democrática, aberta, e escalável baseada em mecanismos de consenso distribuídos descentralizados sem a necessidade de confiar em terceiros. Estas podem ser desenvolvidas com máquinas virtuais que permitem a execução de programas arbitrários, chamados de Smart Contracts. Em Ethereum, a Ethereum Virtual Machine é a máquina virtual global cujo estado é guardado e concordado por todos os participantes da rede. Recentemente, a quantidade de Smart Contracts implementados em Ethereum e a sua complexidade aumentou rapidamente. A interoperabilidade entre eles que a EVM oferece levou ao emergir de um ecossistema de aplicações e protocolos financeiros, chamado de Finanças Descentralizadas. Considerando que estes protocolos asseguram vastas quantidades de capital, erros ou comportamento não intencional frequentemente leva a perdas financeiras catastróficas para os seus utilizadores. Consequentemente, métodos de verificação formal para estes protocolos têm sido alvo de estudo recentemente. A K Framework é uma das mais sofisticadas e capazes frameworks para definir e verificar linguagens. Esta permite definir especificações executáveis arbitrarias de protocolos tal como executar diretamente o seu bytecode com a implementação da EVM em K. Nesta dissertação o objetivo é melhorar a segurança destes protocolos. Para o atingir introduzimos nova documentação para a MakerDAO, um protocolo pioneiro em DeFi, tal como para a sua especificação abstrata em K. Adicionalmente, estendemos a especificação abstrata com um novo modulo e uma invariante do sistema não trivial. Finalmente, desenvolvemos e demonstramos métodos de refinação que permitem provas de refinamento que conectam especificações abstratas de protocolos com a sua implementação em bytecode.

Palavras Chave

Blockchain, Ethereum, Smart Contracts, Verificação Formal, Finanças Descentralizadas

Contents

List of Figures

List of Tables

Listings

Acronyms

DeFi	Decentralized Finance
DAO	Decentralized Autonomous Organization
CDP	Collateralized Debt Position
EVM	Ethereum Virtual Machine
FSM	Finite State Machine
KEVM	Ethereum Virtual Machine specification on the K framework
MCD	Multi-collateral DAI
SAI	Single-collateral DAI

1

Introduction

Contents

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 [5] activity in decentralized ledgers increases its pace [6]. 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 [7, 8]. 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 A discloses the list of major Decentralized Finance (DeFi) 2020 hacks. Previous errors and exploits such as the famous Decentralized Autonomous Organization (DAO) reentrancy attack [9] are decreasing, and current hacks are now non-intuitive, deeply semantic related, and require high-level of expertise as demonstrated in appendix B.

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 develop trust in Smart Contracts even before they are deployed, traditional verification methods such as symbolic analysis approaches, including fuzzing [10], 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 [11] 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.

Our work was conducted in collaboration with the MakerDAO Protocol [12] and Runtime Verification [1]. The MakerDAO protocol was the first DeFi protocol to exist, regularly on the forefront of innovation in the Decentralized Finance field. Runtime Verification is a leader company in formal verification of Smart Contracts known for plentiful large collaborations using K, a semantic framework for design, implementation and formal reasoning.

1.3 Contributions

Our main contributions are summarized as follows:

- We improved MakerDAO's documentation, focusing on refinement proofs, in Chapter 4.
- We introduced documentation for MakerDAO's high-level K framework specification, in Chapter 5.
- We expanded MakerDAO's high-level K specification to include the new Liquidations 2.0 module [1], in chapter 6.
- We formalized a non trivial high-level property over the high-level MakerDAO specification [2], in Chapter 6.
- We developed the software necessary for refinement proofs between the high-level MakerDAO specification, modeled in K, and the EVM bytecode implementation, [3] [4] [5], im Chapter 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 Chapter 7.

1.4 Thesis Outline

The current chapter clarifies this thesis' motivation, its goals, and summarizes the rest of the document. Chapter 2 acquaints the reader with the concept of Blockchain, Smart Contracts, Ethereum and Decentralized Finance. It also introduces Formal Verification and presents a review of the literature on Formal

[1] https://github.com/makerdao/mkr-mcd-spec/pull/250
[2] https://github.com/makerdao/mkr-mcd-spec/pull/250
[3] https://github.com/makerdao/mkr-mcd-spec/tree/mcd-to-kevm-transactions
[4] https://github.com/makerdao/mkr-mcd-spec/tree/simple-refinement
[5] https://github.com/makerdao/mkr-mcd-spec/tree/step-subsorts

Verification of Smart Contracts. Chapter 3 explains the K framework, its underlying rewriting logic, and the Ethereum Virtual Machine implementation in K. Chapter 4 presents MakerDAO and introduces a structured documentation of the protocol. Chapter 5 explains the high-level MakerDAO specification in K, creating documentation, and detailing implementation. Chapter 6 describes the extension of the high-level MakerDAO K framework specification with the Liquidations 2.0 module and a non-trivial system property. Chapter 7 introduces refinement techniques that show the connection between a high and low level model of a system's specifications on the K framework, using MakerDAO's protocol specifications as example. Chapter 8 provides analysis on the extensions of the high-level model, and exemplifies the refinement techniques between the two different implemented K models of the MakerDAO protocol. Chapter 9 summarizes the work presented and contemplates future work.

2

Background

Contents

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 [13] that has subsequently become remarkably influential. Due to its versatility, blockchain related papers are increasing [14] 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, [15] 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 [13] 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 [16], which materialized as Solidity, Ethereum's [17] 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 Ethereum Virtual Machine (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 associated with the program's behavior, in essence what the program means and what is does. One theorem that is significant in this subject is Rice's theorem [18] given that it proved that for all non-trivial semantic properties of a turing-complete program there is no automated algorithm that can decide if such a program has the specified semantic properties. In other words, it is impossible to be fully certain about what a program will do without executing it. As a way of circumventing this theoretical restriction on semantics, programmers realized that if a program is structured as a formal mathematical model then the semantic properties of the program can be reasoned with, using suitable mathematical structures and logic. 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.

2.5.1 Formal Verification and Smart Contracts

There are many previous and on-going studies on smart contract verification, branching in methodology, mathematical structures, logic and implementation efforts. This sub-section describes different approaches in the literature, accentuating trends and challenges, characterizing their association with the proposed framework presented in this paper.

For clarity purposes this sub-section will answer the following research questions (RQ's):

RQ1: What are the formal methodologies that are utilized in the verification of semantic properties in smart contracts?

RQ2: How are specifications or program properties formalized?

RQ3: What types of specifications do these formal methodologies aim to verify?

RQ4: What is the quality of the solutions?

In pursuance of full exploration of papers published related to smart contract formal specification and verification, we queried multiple publication databases using combinations of keywords such as "smart contracts, formal verification, semantic properties, modeling". This databases were Google Scholar, Web of Knowledge and DBLP. In addition Github repositories such as *Smart Contract Languages* and *Ethereum Formal Verification* also contain organized lists of material, some of which is redundant but nevertheless there is a lot of information that hasn't yet been formalized on publications and is relevant as well. We also resorted to other research and survey papers [19–22] for an expeditious overview on the literature.

RQ1: What are the formal methodologies that are utilized in the verification of semantic properties in smart contracts? The answer can be found in the figure that represents a taxonomy of such tooling in F.

RQ2: How are specifications or program properties formalized? Distinct formalization of systems and properties based on the level of abstraction fit into one of two categories [19]: contract-level formalization that verifies high-level behaviour of the contract, typically modeled using a variant of Temporal Logic, and program-level formalization, which is more detailed and dependent on source-code and blockchain platform.

RQ3: What types of specifications do these formal methodologies aim to verify? There is a clear separation between frameworks who allow the user to specify their own smart contract properties [23–30], and the ones who only verify pre-established properties [31]. These predefined properties commonly are if the contract is greedy, the amount of currency out is less than the amount in, reentrancy, integer under/overflows and contract suicide, if the contract can self-destruct.

RQ4: How well do they solve these issues? The quality of the frameworks can be evaluated by their correctness, broadness and real world applicability. Correctness may be measured by the amount of false negatives or positives that a tool produces when verifying smart contracts. Obviously since

theorem proving cannot produce neither, tools that use this method [26–28, 32] are at an advantage. Some model checking and symbolic execution techniques [25] prune the search space using specific blockchain constraints, which allows them to validate properties with higher precision than others. The correctness of the tool is also limited by the quality of the property specification and so tools that use straightforward property formalization [24, 25, 27, 30, 33] have an edge. As mentioned in *RQ3* some frameworks are less broad than others on the formalization of properties, but some of these tools are blockchain agnostic [24, 26, 29] while others are blockchain specific [23, 25, 27, 28, 30, 31, 33, 34] limiting their use. As far as real world applicability goes, it is fair to say that most of these tools have not seen adoption by the community of smart contract programmers since there still aren't plenty formally verified smart contracts with the exception of Ethereum's 2.0 Deposit Contract [35].

3

K Framework

Contents

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 [36] 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 [36]. 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.

As shown in Figure 3.1 [36] the K framework is a semantics first approach, meaning it emphasizes the development and maintenance of a clear and complete formal semantics for the target language and platform, rather than the implementation of tool-level details. Moreover, K has many features that help language definition become simpler, including configuration abstraction and local rewriting, which are techniques that allow each rule to only include the necessary execution state parts for that transition/rule.

Besides the benefits already stated, the K framework defines its tools parametrically over the input language, reducing development and maintenance cost [29]. As a result, parsers, interpreters, debuggers, and verifiers are generated directly from the formal definition, syntax and semantics of the language, which is independent from the implementation details of these tools and auditable by the interested developer community [2].

Once a language is defined, K can read and execute programs in that language both on concrete and symbolic inputs, producing an interpreter and a symbolic execution engine, respectively. These can be regarded as formally derived reference implementations of the language generated automatically from a rigorous semantics and usable for testing this semantics [29].

3.2 Reachability Logic

K's foundation, Reachability Logic, is a logic for symbolically reasoning about possibly infinite transition systems [37]. This logic is equipped with a sound and nearly complete inference system which allows efficient implementations.

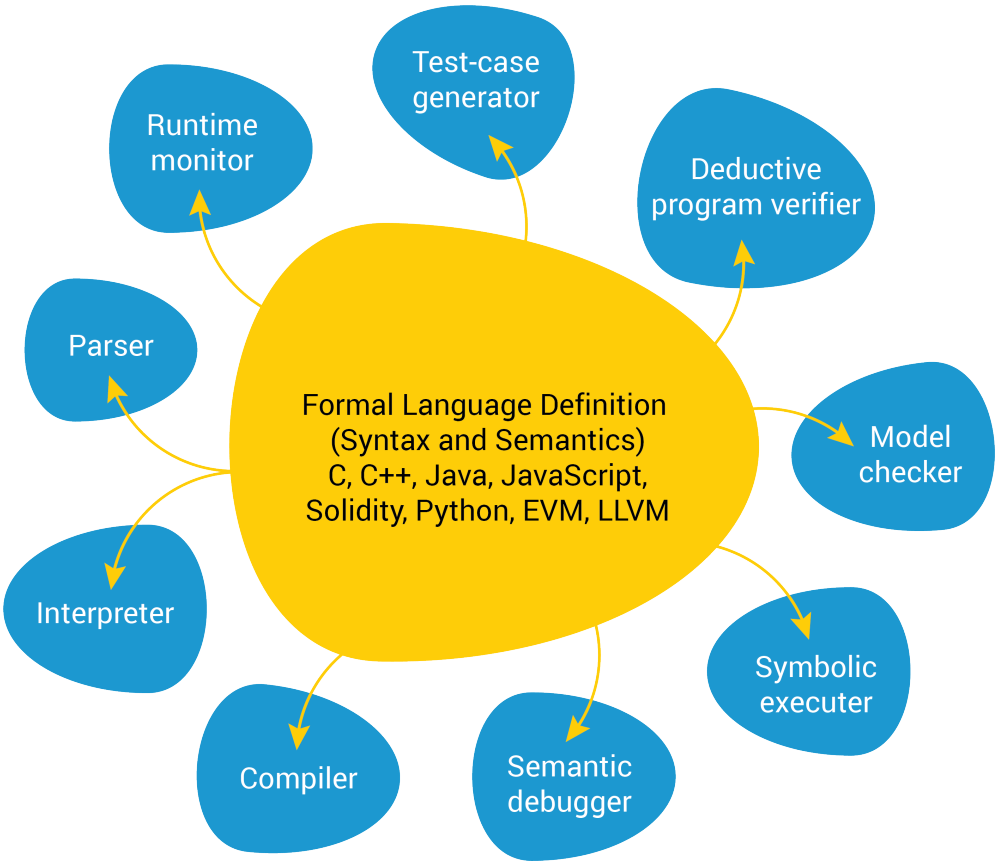

Figure 3.1: K Framework Fundamentals. [1]

3.2.1 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 [38].

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 3.2 presents the formal axioms that comprise the reachability logic system [2].

In the figure 3.2 $\mathcal{A}$ is the initial trusted execution semantics of the programming language, the axioms. The $\mathcal{C}$ on $\vdash_{\mathcal{C}}$ indicates that the circularities $\mathcal{C}$ are reachability claims conjectured but not yet proved. The Circularity proof rule allows us to conjecture any to-be-proven reachability claim as a circularity, while Transitivity allows us to use the circularities as axioms (only after we have made progress on proving them) [2, 37].

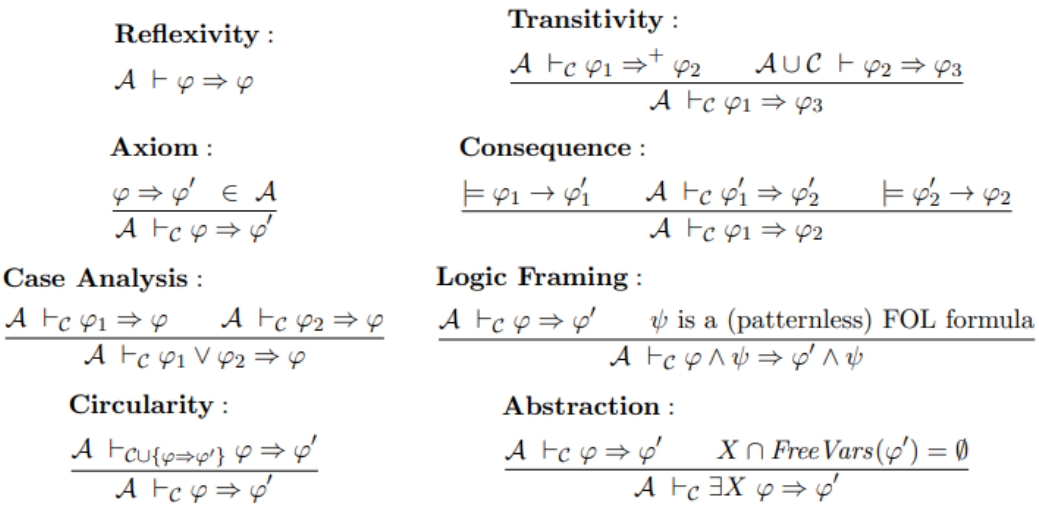

Figure 3.2: Sound and relatively complete proof system of Reachability Logic [2].

Semantically, reachability claims of the form $\phi \Rightarrow \psi$ imply that every configuration in the set of configurations described by pattern ϕ will, eventually, when taking steps described by an arbitrary language semantics, either reach a configuration in the set of configurations described by pattern ψ or not terminate.

Additionally, to better understand this topic, one can easily compare it to the well-known Hoare Logic triples $\{Pre\}Code\{Post\}$ by rewriting the triple as a reachability claim of the form $\overline{Code} \wedge \overline{Pre} \Rightarrow \epsilon \wedge \overline{Post}$, where ϵ is a configuration that represents the empty program. $\overline{Code}$, $\overline{Pre}$, and $\overline{Post}$ are minimal state patterns that respectively contain $Code$, Pre, and $Post$, but specific program variables are substituted with their logical variable counterparts given that Hoare Logic does not differentiate program and logical variables. Moreover, since reachability claims do not trade-off expressiveness they can be used to prove intricate properties, such as Hoare style functional correctness claims.

In fact, another benefit in K is that it is trivial to verify computation complexity in the same way as verifying functional correctness by simply adding a K cell as part of the configuration, which serves as a data structure that increments every time a state transition rule it is tracking is applied.

3.3 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, [2] a formally rigorous executable instance of the EVM semantics in the K framework was

implemented, covering all of the EVM instructions. Known as Ethereum Virtual Machine specification on the K framework (KEVM), this implementation is open-source and can be found on the respective repository [1].

This instance of the EVM semantics within K models transaction execution and network state using the configuration. Additionally, it details changes in transaction execution and network evolution using transition rules. To accomplish this, the state of EVM is divided into two main elements: the state of a currently executing transaction, Smart Contract execution; and the current state of the blockchain, detailing account information. To implement this on K, KEVM's configuration consists of the aforementioned two parts, we subsequently outline:

- **Execution state** Each execution of EVM has an associated account, a program counter, the current program, a word-stack, volatile local memory and *gas* counter. For each opcode in EVM execution the KEVM provides a pipeline of commands over the current opcode with each updating an appropriate piece of the state configuration.
- **Network state** The blockchain, which is a log of state updates, consisting of groupings of transactions called blocks, corresponds to a set of account states.

It is important to note that the K implementation is complete to hold full state information which can be found on appendix C and the remainder of the implementation in the public repository.

The implementation briefly described above is used to create a program verifier capable of demonstrating the complete verification of any example EVM program against a specification. Both the functional correctness of the EVM program as well as the *gas* complexity are among the properties present in the specification. Consequently, the KEVM is demonstrated as an EVM software analysis tools rigorously derived from and backed by a formal, complete, and human-readable EVM semantics.

A more detailed look into how the K formalization EVM is achieved can be found on the official paper [2].

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

4

MakerDAO

Contents

MakerDAO is a Decentralized Finance protocol. The project started in 2015 [39] when Rune Christiansen developed the MakerDAO Protocol on the Ethereum blockchain after first describing the system on a reddit post [40]. 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 Single-collateral DAI (SAI), which was a single-collateral DAI. Both these versions have been deprecated. The last and current version is known as Multi-collateral DAI (MCD).

As mentioned previously, this thesis aims to refine high-level systems models with their Smart Contract bytecode implementation counterparts, as a result we decomposed the MakerDAO protocol into its goal, the mechanisms that can achieve it, as well as Smart Contract implementation.

4.1 Intent

4.1.1 Motivation

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

4.1.2 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.

4.2.1 Characterization of actors

An extended table which elaborates on the characteristics, risks, benefits and interactions of each kind of actor that interacts with the system which should be consulted while reading the following mechanisms descriptions is represented in appendix D.

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

4.2.2 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 Collateralized Debt Position (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 4.1.

$$\sum_{u \in U} dai_u = vice + \sum_{i \in I, u \in U} dbt_{iu} \tag{4.1}$$

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. This can be represented with a temporal change identity that holds for any *ilk i* between any time *t2* and *t1*:

$$\sum_{i,u}(gem_{iu}(t_2) - gem_{iu}(t_1)) = inflow_i(t_2, t_1) - outflow_i(t_2, t_1) + \sum_{u}(ink_{iu}(t_2) - ink_{iu}(t_1)) \tag{4.2}$$

In the equation 4.2, $inflow_i(t_2, t_1)$ indicates the total amount *ilk i* put into the system via inbound asset transfers in the time interval [$t1$, $t2$], and $outflow_i(t_2, t_1)$ represents the total amount of *ilk i* removed from the system via outbound asset transfers in the time interval [$t1$, $t2$].

Additionally, the balance of a collateralized debt position may only decrease if the action is performed by the owning account or by an account authorized by such, which we'll denominate as proprietors of said position. Increasing a position's debt is allowed by the proprietors and accounts with administrative privilege, which, generally, are other components of the system or governance

entities. Decreasing the debt of a position is allowed by its proprietors and must respect the "Fundamental equation of DAI". Removing collateral from a position requires that the position remains safe and can be done by its proprietors or privileged accounts. Adding collateral needs to respect 4.2 and be done by its proprietors. More general actions that simultaneously modify both the collateral and debt of a position are permitted, as long as the resulting position is safe, and all permissioning rules regarding decreasing collateral and increasing debt are obeyed.

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).

Assumption that legitimizes incentives The core assumption justifying this is that entities that draw DAI against collateral derive economic benefit from doing so, and are thus willing to pay the system for the service of being able to borrow in a price-stable asset. The debt of a position can of course change discontinuously whenever new DAI is generated from it or repayed; in between these discontinuous jumps, it evolves according to a differential equation for the fees.

Governance dynamics over CDP's Governance can authorize new modules against the system. This allows governance the possibility of stealing collateral or minting unbacked DAI. Should the crypto economic protections that make these types of governance attacks prohibitively expensive fail, the system may be vulnerable and open for bad actors to drain collateral. Governance is also responsible for fine-tuning the risk parameters associated with each type of collateral that is to be included in the system.

Oracle dynamics over CDPs An oracle module is deployed for each collateral type, feeding it the price data for a corresponding collateral type to the system. Should these price feeds fail or provide byzantine values, it would become possible for unbacked DAI to be minted.

4.2.3 Liquidations

Regarding the MakerDAO protocol, a liquidation is the automatic transfer of collateral from an insufficiently collateralized CDP, along with responsibility for the CDP's debt, to the protocol.

Auction type Liquidations 2.0 uses Dutch auctions which feature instant settlement when a bid is made.

Auction mechanism An auction is started promptly to sell the transferred collateral for DAI in an attempt to cancel out the debt now assigned to the protocol. It does so according to a price calculated from the initial price and the time elapsed since the auction began. It is important for bidders to take into account that while the price almost always decreases with time, there are infrequent occasions on which the price of an active auction may increase, and this could potentially result in collateral being purchased at a higher price than intended. Auctions can reach a defunct state that requires resetting for a reason: too much time has elapsed since the auction started and therefore the ratio of the current price to the initial price has fallen below a certain level.

Auction incentives Currently, liquidations use Dutch auctions, which benefit from the lack of a lock-up period. As a result, not only much of the price volatility risk for auction participants is mitigated, but also enables faster capital recycling.

Governance dynamics over Liquidations Governance is responsible for fine-tuning the auction parameters associated with each type of collateral that is part of the system.

Oracle dynamics over Liquidations The system relies upon a set of trusted oracles to provide price data. Should these price feeds fail or provide byzantine values, coordination problems could arise, including tragedy of the commons or miner collusion. This issue could also lead to negative outcomes such as inappropriate liquidations or the prevention of liquidations that should be possible.

4.2.4 DAI Savings Rate - DSR

The DAI savings rate allows users to deposit DAI and earn savings on it.

Goal The purpose of the DSR is to offer another incentive for holding DAI.

Dynamics of DSR DSR dynamics are quite simple, allowing users to deposit their DAI, earn a certain rate on their principal, and then withdraw the total DAI.

Governance dynamics over DAI Savings Rate The DSR rate is set by MakerDAO Governance, and will typically be less than the base stability fee to remain sustainable.

4.2.5 System Stabilizer and Rates

When the MakerDAO Protocol receives either system debt (from the DAI Savings Rate) or system surplus (from the CDP stability fee accumulation) it will deviate from system equilibrium. In order to deter-

mine whether the system has a net surplus, both the income and debt in the system must be reconciled.

Goal The job of the system stabilizer is to bring it back to system equilibrium. As mentioned previously, there are two types of system unbalance, excess debt or surplus of cash-flow.

Incentives The system stabilizer module creates incentives for Auction Keepers (external actors) to step in and drive the system back to a safe state (system balance) by participating in both debt and surplus auctions and, in return, earn profits.

Debt Auction

Goal A debt auction attempts to raise an amount of DAI equivalent to the amount of debt as fast as possible while minimizing the amount of MKR inflation.

Mechanism Debt Auctions are triggered when the system has DAI debt that has passed a specific debt limit. It is a reverse auction, where keepers bid on how little MKR they are willing to accept for the fixed DAI amount they have to pay at auction settlement.

Governance dynamics over Debt Governance voters determine the system debt limit and various auction parameters.

Surplus Auction

Goal A surplus auction attempts to sell a fixed amount of the surplus DAI in the system for as much MKR as possible. This surplus DAI will come from the Stability Fees that are accumulated from Vaults.

Mechanism Surplus auctions are triggered when the system has a surplus of DAI above the amount decided during the vote. In this auction type, bidders compete with increasing amounts of MKR. Once the auction has ended, the DAI auctioned off is sent to the winning bidder.

Autonomous Feedback Mechanism The system burns the MKR received from the winning bid, which, theoretically should increase perceived market value of MKR.

Governance dynamics over Surplus Governance voters will specify the amount of surplus allowed in the system and various auction parameters.

4.2.6 Oracles

Goal Broadcast on-chain price updates off-chain.

Mechanism A group of authorized whitelisted price feed contracts periodically deliver a new list of prices which is received by the system. Then, it proceeds to compute a median and uses the result to update the perceived value of a collateral.

Governance Dynamics over Oracles Enables the addition and removal of whitelisted price feed addresses and also fine-tunes median calculation parameters.

4.2.7 Governance

Goal Governance facilitates voting by MKR holders, proposal execution, and voting security of the Maker Protocol.

Mechanism Changes to the system are discussed in forums and community chats which are then formalized into proposals. They are precise and leave no space for ambiguity since alterations to a smart contract system are done programmatically and made public to everyone, including non MKR holders. Voters will then proceed to lock their voting tokens, MKR, signaling their decision, giving their votes some weight in the system. If the proposal passes, then there will be a period until the system can be modified, after which the changes to the system can be made permissionessly.

Incentives Modifications of the MakerDAO protocol should lead to improvements which over time will benefit MKR holders.

4.2.8 Emergency Shutdown

Shutdown is a process that can be used as a last resort to directly enforce the Target Price to holders of DAI and CDPs, and protect the Maker Protocol against attacks on its infrastructure.

Goal Allows DAI holders to directly redeem DAI for collateral after an Emergency Shutdown processing period.

Mechanisms Initiating an Emergency Shutdown is decentralized and controlled by MKR holders, who can trigger it instantly by depositing a threshold amount of MKR into the Emergency Shutdown Module, or via a governance vote. CDP owners can retrieve excess collateral from their vaults immediately after initialization of Emergency Shutdown. DAI holders can, after a waiting period, swap their DAI for a relative share of all types of collateral in the system. DAI holders always receive the same relative amount of collateral from the system, whether they are among the first or last people to process their claims.

Incentives This mechanism exists to be activated in the case of serious emergencies, such as long-term market irrationality, hacks, or security breaches.

Governance Dynamics over Emergency Shutdown Governance can determine the period for which DAI holders must wait until they can swap their DAI for collaterals.

An extended formalization, parametrization, and research that pursues the fine-tuning of such parameters that comprise the protocol can be found on MakerDAO's github [41], documentation [4], and governance forum [42].

4.3 Implementation

The concrete implementation of these mechanisms is divided into different modules of Smart Contract collections that together comprise the MakerDAO protocol. In the implementation, actors are considered to be *EOA's* or contracts exterior to the system's contracts. Each of the modules and contracts is ordered in a way, and contains a brief description, that ties them to the mechanisms section, described above.

DAI Module The origin of DAI was designed to represent any token that the core system considers equal in value to its internal debt unit.

- **DAI Contract** The DAI token contract.

MKR Module The MakerDAO governance token module.

- **MKR Contract** The MKR token contract.

Core Module The Core Module is crucial to the system as it contains the entire state of the Maker Protocol and controls the central mechanisms of the system while it is in the expected normal state of operation.

- **Vat Contract** The Vat is the core vault engine of the system. It stores CDPs and tracks all the associated DAI and collateral balances. It also defines the rules by which Vaults and balances can be manipulated. The rules defined in the Vat are immutable (unless the system is redeployed), so in some sense, the rules in the Vat can be viewed as the constitution of the system.
- **Spot Contract** The Spot liaison between the oracles and the core contracts. It functions as an interface contract and stores the price source and collateralization ratio for each *ilk*.

Collateral Module The collateral module is deployed for every new *ilk* (collateral type) added to Vat. It contains all the adapters and auction contracts for one specific collateral type.

- **Join Module** Each join contract is created specifically to allow the given token type to be join'ed to the vat. Hence, various versions of the Join contract have slightly different logic to account for the differences between token implementations.
  - **GemJoin Contract** Allows standard ERC20 tokens to be deposited for use with the system.
  - **ETHJoin Contract** Allows native Ether to be used with the system. It is unused as wrapped Ether (WETH) which is preferred for security reasons.

  - **DaiJoin Contract** Allows users to withdraw their DAI from the system into a standard ERC20 token.

- **Liquidation Module** In the context of the MakerDAO protocol, a liquidation is the automatic transfer of collateral from an insufficiently collateralized Vault, along with the transfer of that Vault's debt to the protocol.
  - **Clip Contract** Handles the Dutch auctions for the liquidated collateral. A different contract is deployed for each collateral type.
  - **Dog Contract** In the dog contract, an auction is started promptly to sell the transferred collateral for DAI in an attempt to cancel out the debt now assigned to the protocol. Unlike other collateral module contracts, there is a single, global Dog.
  - **Abacus Contract** Calculates the prices for the bids of a collateral in a Dutch Auction.

Rates Module A fundamental feature of the MakerDAO system is to accumulate stability fees on Vault debt balances, as well as interest on DSR deposits.

- **Pot Contract** The Pot is the core of the DAI Savings Rate. It allows users to deposit DAI and earn interest.
- **Jug Contract** The primary function of the Jug smart contract is to accumulate stability fees for a particular collateral type whenever its drip() method is called.

System Stabilizer Module It's purpose is to correct the system when the value of the collateral backing DAI drops below the liquidation level (determined by governance) when the stability of the system is at risk.

- **Vow Contract** The Vow represents the overall Maker Protocol's balance (both system surplus and system debt). The purpose of the vow is to cover deficits via debt auctions and discharge surpluses via surplus auctions.
- **Flopper Contract** The Flopper (Debt Auction) is used to get rid of the Vow's debt by auctioning off MKR for a fixed amount of internal system Dai. Firstly, when flop auctions are kicked off, bidders compete with decreasing bids of MKR. After the auction settlement, the Flopper sends received internal Dai to the Vow in order to cancel out its debt. Lastly, the Flopper mints the MKR for the winning bidder.
- **Flapper Contract** The Flapper (Surplus Auction) is used to get rid of the Vow's surplus by auctioning off a fixed amount of internal Dai for MKR. When Flap auctions are kicked off, bidders compete with increasing amounts of MKR. After auction settlement, the Flapper burns the winning MKR bid and sends internal DAI to the winning bidder.

Oracle Module An oracle module is deployed for each collateral type, feeding it the price data for a

corresponding asset to the Vat. The Spotter will then proceed to read from the OSM and will act as the liaison between the oracles and the Vat.

- **Oracle Security Contract** Ensures that new price values propagated from the Oracles are not taken up by the system until a specified delay has passed.
- **Median Contract** The median provides MakerDAO's trusted reference price. In other words, it works by maintaining a whitelist of price feed contracts which are authorized to post price updates. Every time a new list of prices is received, the median of these is computed and used to update the stored value.

Governance Module The Governance Module contains the contracts that facilitate MKR voting, proposal execution, and voting security of the MakerDAO Protocol.

- **Spell Contract** A *Spell* is a contract that performs one or more atomic actions one time only.
- **Pause Contract** The Pause is a delegatecall-based proxy with an enforced delay. This allows authorized users to schedule function calls that can only be executed once a predetermined waiting period has elapsed.
- **Chief Contract** The Chief provides a method to elect a governing address (the "hat") via an approval voting system. Typically the *hat* is a *spell*, but it could in principle be any Ethereum address.

Emergency Shutdown Module Emergency Shutdown stops and gracefully settles the MakerDAO Protocol while ensuring that all users, both Dai holders and Vault users, receive the net value of assets they are entitled to.

- **End Contract** The End's purpose is to coordinate Shutdown. In short, Shutdown closes down the system and reimburses DAI holders.
- **ESM Contract** The Emergency Shutdown Module is a contract with the ability to call End.cage() to trigger the Shutdown of the MakerDAO Protocol.

Proxy Module The system also features a proxy module that enables easier interactions with the Maker protocol. However, given it only contains contract interfaces, proxies, and aliases to functions necessary for both DSR and Vault management and Maker governance it is unimportant in the description of the system. In particular, this code is outside the control of MakerDAO governance and alternate versions can be used permissionlessly.

A graph representing the MakerDAO Protocol Smart Contract Modules System, which was described beforehand, can be found in the appendix E.

5

MakerDAO K specifications

Contents

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-spec.

The system's semantics is divided into sub-modules, each with it's own goal that tie together the mechanism design and implementation of the system at the Solidity contracts level. All files that make up the model, with an exception for the tests, are Markdown files, which allow for inline explanations of the K code within them. In fact, we omit the *.md* at the end of filenames to avoid cluttering. It is also important to note that this high-level model does not specify and take into account the governance mechanisms nor their implementation. The files are explained below:

- **Utility Files** These files are different from the others since they manage what is external to the system's design itself and model accounts, variable types, time, and the general execution semantics.
  - **kmcd-driver** common functionality in all modules.
  - **kmcd** union of all sub-modules.

  - **kmcd-props** statement of properties that we would like to hold for the model.
  - **kmcd-prelude** random testing harness.
  - **fixed-int** fixed point integers in K.

- **Accounting System** Collateralized Debt Positions, Rates, DAI Savings Rate
  - **vat** tracks deposited collateral, open CDPs, and borrowed DAI.
  - **pot** interest accumulation for saved DAI.
  - **jug** stability fee collection.

- **Collateral** DAI, Join
  - **dai** DAI ERC20 token standard.
  - **spot** price feed for collateral.
  - **gem** abstract implementation of collateral.
  - **join** plug collateral into MCD system.

- **Liquidations**
  - **dog** Start liquidations.
  - **clip** Liquidation auction houses.
  - **abacus** Dutch auction price calculator.
  - **cat** deprecated liquidations.
  - **flip** deprecated liquidations.

- **Auction Houses** System Stabilizer
  - **vow** manage and trigger liquidations.
  - **flap** surplus auctions (Vat DAI for sale, bid increasing Gem MKR).
  - **flop** deficit auctions (Gem MKR for sale, lot decreasing Vat DAI).

- **Global Settlement - Emergency Shutdown**
  - **end** close out all CDPs and auctions, attempt to re-distribute gems fairly according to internal accounting.

5.1.1 Types and Basic Operations

The high-level model of the system, as described in the documentation, employes several different data types that need to be formalized into the system's K specification. In K, the Backus-Naur form [43] is used to model types and data structures definitions, specifically by adding non-terminals and suitable production rules. The majority of these type syntax declarations employ a combination of the already built-in K types, booleans as *Bool*, strings as *String*, integers as *Int*, maps as *Map*, and lists as *List*.

As an illustration we show the basic data type *Address* defined in the *kmcd-driver.md* file as:

```
1     syntax Address ::= Int | String
```

Listing 5.1: Address syntactic definition

There are plenty more other intricate data types that are defined by new type declarations and definitions that do not just include the built-in K data types. Whenever one of these new types is implemented it is introduced in the specification a new non-terminal syntax type. Such an example is fixed point integers. Representing fixed point numbers as a tuple of their integer value and their one-point allows the definition of more advanced data types on top, their arithmetic, and projections back into integers which are further defined in the document *fixed-int.md* as:

```
1     syntax FInt ::= "(" FInt ")"                    [bracket]
2                   | FInt ( value: Int , one: Int ) [klabel(FInt), symbol]
```

Listing 5.2: Fixed Point Integers syntactic definition

```
1     syntax Int ::=    baseFInt ( FInt ) [function]
2                     | decimalFInt ( FInt ) [function]
3   // ----------------------------------------------
4     rule    baseFInt(FI) => value(FI) /Int one(FI)
5     rule decimalFInt(FI) => value(FI) %Int one(FI)
```

Listing 5.3: Syntax and Semantic definitions to translate Fixed Point Integers back to Integers

The MakerDAO protocol defines different precision types for numeric values within the system. Concretely, these types are named: *Wad*, numbers with precision to the 18th digit, used for basic quantities; *Ray*, defined as numbers with precison to the 27th digit, used for precise quantities; *Rad*, numbers with precison to the 45th digit, which are the result of multiplying *Wad* with *Ray*; *Bln*, numbers with precision

to the 9th digit, used for conversions between *Wad* and *Ray*. These types are defined as abstractions over the fixed point integers, described above in the *kmcd-data.md* file:

```
    syntax Value ::= Wad | Ray | Rad | Int
// ---------------------------------------

    syntax Int ::= "BLN" | "WAD" | "RAY" | "RAD"
// -------------------------------------------
    rule BLN => 1000000000                                          [macro]
    rule WAD => 1000000000000000000                                 [macro]
    rule RAY => 1000000000000000000000000000                        [macro]
    rule RAD => 1000000000000000000000000000000000000000000000000  [macro]

    syntax Bln = FInt
    syntax Bln ::= Bln ( Int )
// ---------------------------
    rule Bln(I) => FInt(I *Int BLN, BLN) [macro]

    syntax Wad = FInt
    syntax Wad ::= wad ( Int )
// ---------------------------
    rule wad(0) => 0FInt(WAD)            [macro]
    rule wad(1) => 1FInt(WAD)            [macro]
    rule wad(I) => FInt(I *Int WAD, WAD) [macro, owise]

    syntax Ray = FInt
    syntax Ray ::= ray ( Int )
// ---------------------------
    rule ray(0) => 0FInt(RAY)            [macro]
    rule ray(1) => 1FInt(RAY)            [macro]
    rule ray(I) => FInt(I *Int RAY, RAY) [macro, owise]

    syntax Rad = FInt
    syntax Rad ::= rad ( Int )
// ---------------------------
    rule rad(0) => 0FInt(RAD)            [macro]
    rule rad(1) => 1FInt(RAD)            [macro]
    rule rad(I) => FInt(I *Int RAD, RAD) [macro, owise]
```

Listing 5.4: Syntactic and Semantic definitions of different Integer precision types

There are also rules that define diverse basic operations on data types that are useful when executing or reasoning about the specification. Two concrete examples are type conversion rules between different value precision types and arithmetic, defined in the *kmcd-data.md* file below:

```
    syntax Wad ::= Rad2Wad ( Rad ) [function]
// ----------------------------------------
    rule Rad2Wad(FInt(R, RAD)) => FInt(R /Int RAY, WAD)

    syntax Ray ::= Wad2Ray ( Wad ) [function]
// ----------------------------------------
    rule Wad2Ray(FInt(W, WAD)) => FInt(W *Int BLN, RAY)

    syntax Rad ::= Wad2Rad ( Wad ) [function]
                 | Ray2Rad ( Ray ) [function]
// ----------------------------------------
    rule Wad2Rad(FInt(W, WAD)) => FInt(W *Int RAY, RAD)
    rule Ray2Rad(FInt(R, RAY)) => FInt(R *Int WAD, RAD)
```

Listing 5.5: Conversion between different precision types

```
    syntax Rad ::= Wad "*Rate" Ray [function]
// ----------------------------------------
    rule FInt(W, WAD) *Rate FInt(R, RAY) => FInt(W *Int R, RAD)

    syntax Wad ::= Rad "/Rate" Ray [function]
// ----------------------------------------
    rule FInt(_ , _  ) /Rate ray(0)        => wad(0)
    rule FInt(R1, RAD) /Rate FInt(R2, RAY) => FInt(R1 /Int R2, WAD) [owise]
```

Besides the aforementioned types, the MakerDAO system also defines intricate data structures that maintain the registers of the system current state. In Solidity, the keyword *struct* followed by the desired variables implements these data structures, available on the file *vat.sol* as such:

```
pragma solidity >=0.5.12;

contract Vat {

    /*
    ...
    */

  //
  // --- Data ---
    struct Ilk {
        uint256 Art;   // Total Normalised Debt     [wad]
        uint256 rate;  // Accumulated Rates         [ray]
        uint256 spot;  // Price with Safety Margin  [ray]
        uint256 line;  // Debt Ceiling              [rad]
        uint256 dust;  // Urn Debt Floor            [rad]
    }
    struct Urn {
        uint256 ink;   // Locked Collateral  [wad]
        uint256 art;   // Normalised Debt    [wad]
    }

    /*
    ...
    */
}
```

Listing 5.6: Solidity Vat Data Structure

The high-level specification counterpart, designated in the K specification as *records*, consists of a syntax production that embody the same registers, implemented on the file *vat.md* as such::

```
    syntax VatIlk ::= Ilk ( Art: Wad , rate: Ray , spot: Ray , line: Rad , dust:
      Rad ) [klabel(#VatIlk), symbol]

    syntax VatUrn ::= Urn ( ink: Wad , art: Wad ) [klabel(#VatUrn), symbol]
```

Listing 5.7: Example syntax of a record definition

In order to access any of the attributes inside these records, the K framework has built-in operations that automatically give semantic meaning to the also automatically created syntactic definitions:

```
    syntax Wad ::= ink ( VatUrn )
                 | art ( VatUrn )
// ------------------------------
```

Listing 5.8: Example access of record attributes

Without going into detail on how the K backend works, one can abstractly consider the semantic rules to be implemented as:

```
    rule ink ( Urn (ink: INK, art: ART ) ) => INK
    rule art ( Urn (ink: INK, art: ART ) ) => ART
```

Listing 5.9: Semantics of accessing record attributes

Finally, after these declarations the specification also defines syntactic and semantic rules to calculate values that are to be used by the system based on the former records. Some examples are the urn balances, urn debts, and urn collaterals, which are calculated on the *vat.md* file by these definitions:

```
    syntax Rad ::= urnBalance     ( VatIlk , VatUrn ) [function, functional]
                 | urnDebt        ( VatIlk , VatUrn ) [function, functional]
                 | urnCollateral  ( VatIlk , VatUrn ) [function, functional]
// ------------------------------------------------------------------------
    rule urnBalance  (ILK, URN) => urnCollateral(ILK, URN) -Rad urnDebt(ILK, URN)
    rule urnDebt     (ILK, URN) => art(URN) *Rate rate(ILK)
    rule urnCollateral(ILK, URN) => ink(URN) *Rate spot(ILK)
```

Listing 5.10: Record manipulation example

Furthermore, each *MCDContract*, the high-level specification of each solidity contract, is an *Address* as well under the assumption that there is a unique live instance of each one at a time, so we complement the *Address* definition with:

```
    syntax Address ::= MCDContract
```

Listing 5.11: Syntactic definition of Contracts

An example of a *MCDContract* can be found in the file *vat.md* which defines the vat:

```
1     syntax MCDContract ::= VatContract
2     syntax VatContract ::= "Vat"
```

Listing 5.12: Syntactic definition of the Vat Contract

5.1.2 The Maker Configuration

The MakerDAO system state implementation on the Ethereum blockchain is scattered across the many Solidity contracts that together constitute the MakerDAO protocol. In this K specification, each file that models a Solidity contract also contains its own state through its configuration. In each of the file's configuration, each piece of state is modeled by the proper K cell, as demonstrated using *vat.md* as an example:

```
1     configuration
2       <vat>
3         <vat-wards> .Set   </vat-wards>
4     // mapping (address (address => uint))       Address |-> Set
5         <vat-can>   .Map   </vat-can>
6     // mapping (bytes32 => Ilk)               String  |-> VatIlk
7         <vat-ilks>  .Map   </vat-ilks>
8     // mapping (bytes32 => (address => Urn))  CDPID   |-> VatUrn
9         <vat-urns>  .Map   </vat-urns>
10    // mapping (bytes32 => (address => uint256)) CDPID   |-> Wad
11        <vat-gem>   .Map   </vat-gem>
12    // mapping (address => uint256)              Address |-> Rad
13        <vat-dai>   .Map   </vat-dai>
14    // mapping (address => uint256)              Address |-> Rad
15        <vat-sin>   .Map   </vat-sin>
16    // Total Dai Issued
17        <vat-debt>  rad(0) </vat-debt>
18    // Total Unbacked Dai
19        <vat-vice>  rad(0) </vat-vice>
20    // Total Debt Ceiling
21        <vat-Line>  rad(0) </vat-Line>
22    // Access Flag
```

```
23          <vat-live>  true   </vat-live>
24        </vat>
```

Listing 5.13: Vat Contract K configuration

When representing types in K it is common to designate the empty type using the dot construct, which is represented in the previous configuration as *.Map* or *.Set*. For built-in types, like Map and Set, this production is also built-in however for user-defined types it is also required that the user manually defines this syntax.

In the K specification of the system, these initially independent configurations are conjoined in the *kmcd.md* file in order to allow each of the contracts to access the state of the other contracts, as shown below.

```
1     configuration
2       <kmcd>
3         <kmcd-driver/>
4         <kmcd-state>
5           <abaci/>
6           <cat/>
7           <clip-state/>
8           <dai/>
9           <dog-state/>
10          <end-state/>
11          <flap-state/>
12          <flips/>
13          <flop-state/>
14          <gems/>
15          <join-state/>
16          <jug/>
17          <pot/>
18          <spot/>
19          <vat/>
20          <vow/>
21        </kmcd-state>
22      </kmcd>
```

Listing 5.14: MCD system configuration

This method also allows the state to be explicitly altered by the state transition functions, explained in the next section, which would not be possible otherwise.

The following configuration, encompassed in the configuration above by inclusion of the cell $\langle kmcd-driver/\rangle$ from the file *kmcd-driver.md*, defines the execution frame state when executing a transaction on the system, making it possible to revert to the initial state before trying to apply the transaction, in case the transaction reverts. This configuration also stores the necessary information to provide to the system who initiated the attempt to change the system's state. Finally, the state transitions are logged making it easier to debug and reason about the system.

```
configuration
  <kmcd-driver>
    <return-value> .K </return-value>
    <msg-sender> 0:Address </msg-sender>
    <this> 0:Address </this>
    <current-time> 0:Int </current-time>
    <mcd-call-stack> .List </mcd-call-stack>
    <pre-state> .K </pre-state>
    <events> .List </events>
    <tx-log> .Transaction </tx-log>
    <frame-events> .List </frame-events>
    <kevm/>
  </kmcd-driver>
```

Listing 5.15: Execution framework configuration

It is crucial to notice that the special cell $\langle k\rangle$, that is responsible for the operations that are meant to be executed, is included in the KEVM configuration with the $\langle kevm/\rangle$ cell. The KEVM has already been introduced in this document and further details about its configuration may be found in the open-source repository https://github.com/kframework/evm-semantics.

The preceding configurations are further grouped in the *kmcd-props.md* file which adds the state needed for property checking and testing, described in the Model Validation section.

```
configuration
  <kmcd-properties>
    <kmcd/>
    <processed-events> .List </processed-events>
    <properties> #violationFSMs </properties>
```

```
6       </kmcd-properties>
```

Listing 5.16: High-level properties configuration

Finally, the last configuration declaration that congregates all the previously mentioned configurations and encompasses all of the state fragments of the model is found in the *kmcd-prelude.md* file. This configuration adds the possibility of creating snapshots of the system to aid debugging and the state necessary to randomize concrete testing of the system.

```
1     configuration
2       <kmcd-random>
3         <kmcd-properties/>
4         <kmcd-snapshots> .List </kmcd-snapshots>
5         <kmcd-gen>
6           <random> String2Bytes($RANDOMSEED:String) </random>
7           <used-random> .Bytes </used-random>
8           <generator-next> 0 </generator-next>
9           <generator-current> 0 </generator-current>
10          <generator-remainder> .GenStep </generator-remainder>
11          <generators>
12            <generator multiplicity="*" type="Map">
13              <generator-id> 0 </generator-id>
14              <generator-steps> .GenStep </generator-steps>
15            </generator>
16          </generators>
17        </kmcd-gen>
18      </kmcd-random>
```

Listing 5.17: Test randomizer configuration

5.1.3 State Transition Functions

The MakerDAO system specification defines progress in the system's state with state transition functions. In Solidity, these are implemented as functions that can be called by an Externally Owned Account to trigger changes within the system. In this section, we describe how the high-level model of the system handles state changes, which are modeled as an abstraction over the execution framework of the Ethereum Virtual Machine and Solidity's semantics.

In the high-level model of the system, these state changes are implemented as K rules that change certain parts of the K system configuration, known as transactions, which are also initiated by a user. As discussed earlier, one of the advantages of the K framework is the possibility of abstracting parts of the configuration that are left unchanged when writing a rule. In complex models, such as this one, if for every rule every K cell had to be mentioned, it would become unfeasible to read and understand the code. Consequently, whenever parts of the configuration are omitted in a rule, it should be clear that they remain unmodified. It is essential to note that transitions within the MakerDAO system are deterministic, implying that a resulting state of the system is only determined by the initial state and applied transitions.

An *EthereumSimulation*, also extended in the EVM semantics formalization in K, has its definition extended here as an *MCDSteps*, which is the type that defines every state transition rule to the MakerDAO system. It is fundamental to note that the *EthereumSimulation* is introduced here in conjuntion with the KEVM to allow the refinement proofs that are presented in chapter 7.

```
1     syntax EthereumSimulation ::= MCDSteps
2   // ---------------------------------------
```

Listing 5.18: Syntactic definition that allows the interoperability of both models

Additionally, since an execution of the MakerDAO system consists of an arbitrary number of transactions, *MCDSteps* is either the empty step or the set of one or more steps. If we reach the empty step, *".MCDSteps"*, then the exit-code is updated to 0, indicating a successful execution. When there are one or more steps the execution is unfolded using the $\sim\rangle$ operator, indicating a sequence of commands in K, specified in K as stacking a computation on top of a continuation.

```
1     syntax MCDSteps ::= ".MCDSteps" | MCDStep MCDSteps
2   // ------------------------------------------------
3     rule <k> .MCDSteps => . ... </k> <exit-code> _ => 0 </exit-code>
4     rule <k> MCD:MCDStep MCDS:MCDSteps => MCD ~> MCDS ... </k>
```

Listing 5.19: System step definition

An *AdminStep* defines state transactions of the complete K high-level model of the system, including initiating transactions, snapshotting of the system, and testing. It is not to be confused with any kind of *system administrator* of the MakerDAO system.

```
1     syntax MCDStep ::= AdminStep
```

Listing 5.20: Syntactic definition of special permission steps

5.1.3.A Transactions

A change to the MakerDAO system's state is initiated by a top-level call by a given user, defined by the terminal symbol *transact*, followed by the user's identifier in the system and the function call desired by the user. The current state is then saved, with the push, drop, pop state commands used for state roll-back, and are given semantics once the entire configuration is present. Then, finally, then the transaction executed. If the execution succeeds then the state is successfully updated. However, if it reverts, the state is rolled back to the previous configuration, and in both cases the transaction is logged.

```
1     syntax AdminStep ::= "transact" Address MCDStep | "#end-transact"
2   // ----------------------------------------------------------------
3     rule <k> transact ADDR:Address MCD:MCDStep => pushState ~> call MCD ~> #end-
        transact ~> assert ~> dropState ... </k>
4     <this> _ => ADDR </this>
5     <msg-sender> _ => ADDR </msg-sender>
6     <mcd-call-stack> _ => .List </mcd-call-stack>
7     <pre-state> _ => .K </pre-state>
8     <tx-log> _ => Transaction(... acct: ADDR, call: MCD, events: .List,
        txException: false) </tx-log>
9     <frame-events> _ => .List </frame-events>
10    <return-value> _ => .K </return-value>
11
12    rule <k> #end-transact => . ... </k>
13    <events> ... (.List => ListItem(TXLOG)) </events>
14    <tx-log> TXLOG => .Transaction </tx-log>
15
16    rule <k> exception MCDSTEP ~> #end-transact => #end-transact ~> exception
        MCDSTEP ... </k>
17    <tx-log> Transaction(... txException: _ => true) </tx-log>
```

Listing 5.21: MCD transaction definition

5.1.3.B Authorization

Authorization happens at the call boundaries, containing both transactions and calls between MCD contracts. Each contract must have defined the authorized function, which returns the set of accounts that are authorized to change the state in regards to a specific account details in the system. By default it is assumed that the special ADMIN account is authorized on all other contracts (for running simulations). The special account ANYONE is not authorized to do anything, so it represents any user in the system.

```
    syntax Set ::= wards ( MCDContract ) [function, functional]
// ------------------------------------------------------------
    rule wards(_) => .Set [owise]

    syntax Address ::= "ADMIN" | "ANYONE"
// -------------------------------------
    syntax Bool ::= isAuthorized ( Address , MCDContract ) [function]
// ------------------------------------------------------------------
    rule isAuthorized( ADDR , MCDCONTRACT ) => ADDR ==K ADMIN orBool ADDR in
     wards(MCDCONTRACT)
```

Listing 5.22: Authorization step definition

5.1.3.C Function Calls

Internal function calls implement the call stack to create call-frames and return values to their caller. On exception, the entire current call is discarded to trigger state roll-back (we assume no error handling on internal exception).

```
     syntax CallFrame ::= frame(prevSender: Address, prevEvents: List,
    continuation: K)
// ------------------------------------------------------------------

    syntax AdminStep ::= "call"     MCDStep
                       | "makecall" MCDStep
// ---------------------------------
    rule <k> call MCD:MCDStep => checkauth MCD ~> checklock MCD ~> makecall MCD ~
     > checkunlock MCD ...  </k>
```

```
8
9       rule <k> makecall MCD:MCDStep ~> CONT => MCD </k>
10          <msg-sender> MSGSENDER => THIS </msg-sender>
11          <this> THIS => contract(MCD) </this>
12          <mcd-call-stack> .List => ListItem(frame(MSGSENDER, EVENTS, CONT)) ... </
          mcd-call-stack>
13          <frame-events> EVENTS => ListItem(LogNote(MSGSENDER, MCD)) </frame-
          events>
14
15      rule <k> . => CONT </k>
16          <msg-sender> MSGSENDER => PREVSENDER </msg-sender>
17          <this> _THIS => MSGSENDER </this>
18          <mcd-call-stack> ListItem(frame(PREVSENDER, PREVEVENTS, CONT)) => .List
          ... </mcd-call-stack>
19          <tx-log> Transaction(... events: L => L EVENTS) </tx-log>
20          <frame-events> EVENTS => PREVEVENTS </frame-events>
```

Listing 5.23: Execution framework function call definition

5.1.3.D Modifiers

Modifiers in Solidity are used to modify the behavior of a function. At the moment, these are typically used in the codebase to check prerequisite conditions when acessing functions in order to prevent unauthorized access and re-entrant calls. In the high-level model of the system, *AuthStep* is used as the modifier to check if a caller belongs to the contract's wards while *LockStep* is used as a non re-entrant check.

```
1       syntax AuthStep
2       syntax AdminStep ::= ModifierStep
3       syntax MCDStep   ::= LockStep | AuthStep
4   // ---------------------------------------
5
6       syntax WardStep ::= "rely" Address
7                         | "deny" Address
8   // ----------------------------------
9
10      syntax ModifierStep ::= "checkauth"   MCDStep
11                            | "checklock"   MCDStep
```

```
12                              | "checkunlock" MCDStep
13                              | "lock"        MCDStep
14                              | "unlock"      MCDStep
15    // ----------------------------------------------
16
17        syntax LockAuthStep
18        syntax LockStep ::= LockAuthStep
19        syntax AuthStep ::= LockAuthStep
20    // --------------------------------
```

Listing 5.24: Function modifiers definition

5.1.3.E Exception Handling

Whenever an exception occurs, the state must be rolled back. During the regular execution of a step, this implies popping the mcd-call-stack and rolling back frame-events.

```
1       syntax Event ::= Exception ( Address , MCDStep ) [klabel(LogException)]
2     // ------------------------------------------------------------------------
3
4       syntax AdminStep ::= "exception" MCDStep
5     // ----------------------------------------
6       rule <k> MCDSTEP:MCDStep => exception MCDSTEP ... </k>
7         requires notBool isAdminStep(MCDSTEP) [owise]
8
9       rule <k> exception E ~> _ => exception E ~> CONT </k>
10           <msg-sender> MSGSENDER => PREVSENDER </msg-sender>
11           <this> _THIS => MSGSENDER </this>
12           <mcd-call-stack> ListItem(frame(PREVSENDER, PREVEVENTS, CONT)) => .List
       ... </mcd-call-stack>
13           <tx-log> Transaction(... events: L => L EVENTS) </tx-log>
14           <frame-events> EVENTS => PREVEVENTS </frame-events>
15
16      rule <k> exception _MCDSTEP ~> dropState => popState ... </k>
17           <mcd-call-stack> .List </mcd-call-stack>
18
19      rule <k> exception _ ~> (assert         => .) ... </k>
20      rule <k> exception _ ~> (_:ModifierStep => .) ... </k>
```

```
21      rule <k> exception _ ~> (makecall _      => .) ... </k>
```

Listing 5.25: Execution exceptions and state rollback definition

5.1.3.F Events

Most operations add entries to the log, which stores the address that made the call and the step that is being logged.

```
1       syntax Event ::= LogNote(Address, MCDStep) [klabel(LogNote), symbol]
2   // ----------------------------------------------------------------------
3       syntax CustomEvent
4       syntax Event ::= CustomEvent
5   // ----------------------------
```

Listing 5.26: Event recording definition

5.1.3.G Time steps

Some methods rely on a timestamp. The time representation within this high-level model of the system is defined below.

```
1       syntax Event ::= TimeStep ( Int , Int ) [klabel(LogTimeStep), symbol]
2   // ----------------------------------------------------------------------
3
4       syntax MCDStep ::= "TimeStep"
5                        | "TimeStep" Int
6   // --------------------------------
7       rule <k> TimeStep => TimeStep 1 ... </k>
8
9       rule <k> TimeStep N => assert ... </k>
10           <current-time> TIME => TIME +Int N </current-time>
11           <events> ... (.List => ListItem(TimeStep(N, TIME +Int N))) </events>
12        requires N >Int 0
```

Listing 5.27: Time representation definition

5.1.3.H Contract Semantics

In the last sections, we explained how the model's execution works, clearly defining how the EVM's execution is abstracted. Alternatively, we will now explain how the Solidity semantics of the Smart Contract implementations of the system are abstracted, using *vat.md* as an example of what is used throughout the system's model.

A function call is modeled as an *MCDStep* and identified by the contract name, *VatContract* and the function, name and arguments, *VatStep*, to be called.

```
    syntax MCDStep ::= VatContract "." VatStep [klabel(vatStep)]
// ------------------------------------------------------------
```

Listing 5.28: Syntactic definition of Vat function calls

In the following example, we compare how we model the constructor of the vat contract in the K specification against the Solidity implementation of the same function:

```
pragma solidity >=0.5.12;

contract Vat {

    /*
    ...
    */

        // --- Init ---
    constructor() public {
        wards[msg.sender] = 1;
        live = 1;
    }

    /*
    ...
    */
}
```

```
    syntax VatStep ::= "constructor"
// ---------------------------------
    rule <k> Vat . constructor => . ... </k>
         <msg-sender> MSGSENDER </msg-sender>
          ( <vat> _ </vat>
        => <vat>
             <vat-wards> SetItem(MSGSENDER) </vat-wards>
             <vat-live> true </vat-live>
             ...
           </vat>
          )
```

Listing 5.29: Vat Constructor function in K

As we can observe, both implementation use the same function name and number of arguments. The differences are the data types and the execution framework, where Solidity uses maps and integers, the K model uses sets and booleans. Additionally, in Solidity, the Smart Contract implementation executes over bytecode however the high-level model executes in the K semantics execution model described before.

Another example of differences in implementation is the usage of modifiers in the high-level model contrasting to the use of modifiers in Solidity.

```
pragma solidity >=0.5.12;

contract Vat {
    // --- Auth ---
    mapping (address => uint) public wards;
    function rely(address usr) external auth { require(live == 1, "Vat/not-live")
      ; wards[usr] = 1; }
    function deny(address usr) external auth { require(live == 1, "Vat/not-live")
      ; wards[usr] = 0; }
    modifier auth {
        require(wards[msg.sender] == 1, "Vat/not-authorized");
        _;
    }

    /*
    ...
```

```
15      */
16  }
```

Listing 5.30: Modifier definition in Solidity

```
1       syntax VatStep  ::= VatAuthStep
2       syntax AuthStep ::= VatContract "." VatAuthStep [klabel(vatStep)]
3   // ----------------------------------------------------------------
4       rule [[ wards(Vat) => WARDS ]] <vat-wards> WARDS </vat-wards>
5
6       syntax VatAuthStep ::= WardStep
7   // -------------------------------
8       rule <k> Vat . rely ADDR => . ... </k>
9            <vat-wards> ... (.Set => SetItem(ADDR)) </vat-wards>
10           <vat-live> true </vat-live>
11
12      rule <k> Vat . deny ADDR => . ... </k>
13           <vat-wards> WARDS => WARDS -Set SetItem(ADDR) </vat-wards>
14           <vat-live> true </vat-live>
```

Listing 5.31: Modifier usage in K

Again, both implementations use the same function name and arguments, and both use different data types and execution frameworks. In the K implementation, the *rely* and *deny* methods are modeled as an *AuthStep*, which as shown earlier is the *auth* modifier equivalent for the high-level model.

In the Solidity implementation, there are also *requires* clauses in the function calls that revert the transaction, rolling back the state of the system. An example is shown below from the *jug.sol* contract.

```
1   pragma solidity >=0.5.12;
2
3   contract Jug {
4
5       /*
6       ...
7       */
8
9           function drip(bytes32 ilk) external returns (uint rate) {
```

```
10          require(now >= ilks[ilk].rho, "Jug/invalid-now");
11          (, uint prev) = vat.ilks(ilk);
12          rate = rmul(rpow(add(base, ilks[ilk].duty), now - ilks[ilk].rho, ONE),
        prev);
13          vat.fold(ilk, vow, diff(rate, prev));
14          ilks[ilk].rho = now;
15      }
16  }
```

Concerning the high-level K specification, these are implemented with *requires* at the end of the definition of each state transition rules. In the K framework, the boolean conditions that follow the *requires* keyword restrict the possible states that could match and, therefore, apply the specified rule, making it impossible to apply state changes with the restricted rule, much like a rollback with the *require* keyword in Solidity. The counterexample to the one shown above is presented below:

```
1       syntax JugStep ::= "drip" String
2   // --------------------------------
3       rule <k> Jug . drip ILK_ID => call JUG_VAT . fold ILK_ID ADDRESS ( ( (BASE +
          Ray ILKDUTY) ^Ray (TIME -Int ILKRHO) ) *Ray ILKRATE ) -Ray ILKRATE ... </k>
4           <current-time> TIME </current-time>
5           <jug-vat> JUG_VAT </jug-vat>
6           <vat-ilks> ... ILK_ID |-> Ilk ( ... rate: ILKRATE ) ... </vat-ilks>
7           <jug-ilks> ... ILK_ID |-> Ilk ( ... duty: ILKDUTY, rho: ILKRHO => TIME )
          ... </jug-ilks>
8           <jug-vow> ADDRESS </jug-vow>
9           <jug-base> BASE </jug-base>
10        requires TIME >=Int ILKRHO
```

Finally, the semantics of the rest of the contracts are modeled in similar fashion, using the appropriate data types and execution semantics of the high-level model aforementioned.

5.1.4 Model verification

In the previous section we made clear how the formalization of the high-level specification of the Maker system is modeled in the K framework. We will now elaborate how this specification is tested and verified.

The MakerDAO implementation in Solidity is complemented with unit tests that validate it. They are

implemented in solidity syntax with some extra non-traditional annotations that are not part of the regular solidity semantics. These extra semantics are defined by the HEVM [44] "an implementation of the EVM made specifically for symbolic execution, unit testing and debugging of Smart Contracts.". They allow extra operations that manually manipulate the EVM state for the purpose of testing, such as storing debug variables or changing EVM's time. One can easily use these functionalities simply by adding an interface to their Solidity tests and then calling the methods, both using regular Solidity syntax. An example of these tests from the file *vat.t.sol* is shown below:

```
pragma solidity >=0.5.12;

interface Hevm {
    function warp(uint256) external;
    function store(address,bytes32,bytes32) external;
}

    /*
    ...
    */

    function setUp() public {
        hevm = Hevm(0x7109709ECfa91a80626fF3989D68f67F5b1DD12D);
        hevm.warp(604411200);

        gov = new DSToken('GOV');
        gov.mint(100 ether);
        /*
        ...
        */
    }
```

Listing 5.32: Example Hevm test

The MakerDAO high-level specification is verified using randomly generated concrete testing that leverages the use of Finite State Machines at each step of execution to assure that certain properties hold within the system. The code example illustrated above makes use of HEVM's extended semantics, which is used to automatically model all of the high-level concrete tests in Solidity, as shown in example by the file *MkrMcdSpecSolTests.sol*, found on the repository.

As mentioned beforehand the high-level specification of the MakerDAO system updates the state

when a user initiates a transaction. The K code that formalizes a transaction and its explanation can be found in the 5.1.3.A section. Verification of the specification's defined properties happens every time a transaction is executed. These properties are modeled as Finite State Machines and they are checked to track whether certain properties of the system are violated or not, using the *Adminstep assert*. These were modeled to assure that some known vulnerabilities of the MakerDAO protocol, and disclosed attack vectors by the developer known as lucash [45], were in fact patched in updates. If a violation is detected, it is recorded in the state and execution is immediately terminated.

```
1     syntax AdminStep ::= "#assert" | "#assert-failure"
2   // ----------------------------------------------------
3     rule <k> assert => deriveAll(keys_list(VFSMS), #extractAssertEvents(EVENTS
        ListItem(Measure()))) ~> #assert ... </k>
4         <events> EVENTS => .List </events>
5         <properties> VFSMS </properties>
6
7     rule <k> #assert => . ... </k>
8         <properties> VFSMS </properties>
9       requires notBool anyViolation(values(VFSMS))
10
11    rule <k> #assert => #assert-failure ... </k> [owise]
12
13    syntax List ::= #extractAssertEvents ( List ) [function]
```

Listing 5.33: Property verifier definition in K

A violation occurs if any of the properties below holds.

```
1     syntax Map ::= "#violationFSMs" [function]
2   // ----------------------------------------
3     rule #violationFSMs => ( "Zero-Time Pot Interest Accumulation" |->
        zeroTimePotInterest                          )
4                          ( "Pot Interest Accumulation After End" |->
        potEndInteresst                              )
5                          ( "Unauthorized Flip Kick"              |->
        unAuthFlipKick                               )
6                          ( "Unauthorized Flap Kick"              |->
        unAuthFlapKick                               )
```

```
7                               ( "Total Bound on Debt"                 |->
        totalDebtBounded(... dsr: ray(1))             )
8                               ( "PotChi PotPie VatPot"                |->
        potChiPieDai(... offset: rad(0))              )
9                               ( "Fundamental Dai Equation"            |->
        fundamentalDaiEquation                        )
10                              ( "Total Backed Debt Consistency"       |->
        totalBackedDebtConsistency                    )
11                              ( "Debt Constant After Thaw"            |->
        debtConstantAfterThaw                         )
12                              ( "Flap Dai Consistency"                |->
        flapDaiConsistency                            )
13                              ( "Flap MKR Consistency"                |->
        flapMkrConsistency                            )
14                              ( "Flop Block Check"                    |->
        flopBlockCheck(... embers: rad(0), dented: 0) )
```

For each Finite State Machine (FSM), the user must define the derive function, which dictates how that FSM behaves. A default *owise* rule is added which leaves the FSM state unchanged.

```
1       syntax ViolationFSM ::= derive ( ViolationFSM , Event ) [function]
2   // ----------------------------------------------------------------------
3       rule derive(VFSM, _) => VFSM [owise]
```

Listing 5.34: Property violation checker definition

We present below the example of the Finite State Machine named “Debt Constant After Thaw” which constantly tracks the *Vat.debt* in order to guarantee that it didn’t change after *End.thaw* is called, as this implies the creation or destruction of DAI which would ruin *End’s* accounting.

```
1       syntax ViolationFSM ::= "debtConstantAfterThaw"
2   // ----------------------------------------------
3       rule derive(debtConstantAfterThaw, Measure(... debt: DEBT, endDebt: END_DEBT)
          ) => Violated(debtConstantAfterThaw) requires (END_DEBT =/=Rad rad(0))
          andBool (DEBT =/=Rad END_DEBT)
```

Listing 5.35: Example defition of the debtConstantAfterThaw property

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 3.3. 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, using the *vat* and *vow* as examples, in the following manner:

```
    requires "data.md"

module DSS-BIN-RUNTIME
    imports EVM-DATA

    syntax ByteArray ::= "Vat_bin_runtime"
  // ------------------------------------
    rule Vat_bin => #parseByteStack("0x60806040523 (...) 300050c0032") [macro]

    syntax ByteArray ::= "Vow_bin_runtime"
  // ------------------------------------
    rule Vow_bin_runtime => #parseByteStack("0x60806040 (...) 0050c0032") [macro]
```

Listing 5.36: EVM bytecode insertion in K syntax

5.2.1 Model Verification

This model of the MakerDAO protocol is verified with the KEVM that symbolically executes the bytecode against reachability claims. These reachability claims could be manually defined but doing so would require an in-depth knowledge of K by the developer and a time consuming effort. In order to reduce these overheads, the property specification format known as ACT was created to aid Smart Contract developers looking to verify their bytecode with different backends.

ACT specifications are functional descriptions of the behavior of a smart contract. Taking as an example the specification of the behaviour of the *heal* function of the contract *Vat*:

```
behaviour heal of Vat
```

```
2  interface heal(bytes32 u, bytes32 v, int256 rad)
3
4  types
5
6      Can   : uint256
7      Dai_v : uint256
8      Sin_u : uint256
9      Debt  : uint256
10     Vice  : uint256
11
12 storage
13
14     #Vat.wards(CALLER_ID) |-> Can
15     #Vat.dai(v)           |-> Dai_v => Dai_v - rad
16     #Vat.sin(u)           |-> Sin_u => Sin_u - rad
17     #Vat.debt             |-> Debt  => Debt - rad
18     #Vat.vice             |-> Vice  => Vice - rad
19
20 iff
21
22     Can == 1
23
24 iff in range uint256
25
26     Dai_v - rad
27     Sin_u - rad
28     Debt - rad
29     Vice - rad
```

Listing 5.37: Example ACT property specification

Behavior specifications in ACT then generate a series of reachability claims, defining both positive and negative behaviors of the contract, as succeeding and reverting claims, respectively. In the example above, the specification will generate two reachability claims, a positive behavior *Vat_heal_succ_pass_rough.k* and a negative behavior *Vat_heal_fail_rough.k*. Both of these claims will refer to the bytecode of the contract Vat and use the function signature of heal (bytes32,bytes32,int256) as the first 4 bytes of calldata, keeping the rest of the calldata abstract. In the success specification, the conditions under the *iff* headers are postulated, while in the fail specification it is their negation.

The rest of the intended contract behaviors are specified in the same way as we've shown and the

generated KEVM test suite can be found in this repository.

6

High-level Specification Extension

Contents

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

6.1 Liquidations 2.0

6.1.1 System Upgrades

As mentioned in section 4.2.3, the MakerDAO protocol uses liquidations of uncollateralized vaults as a mechanism for maintaining DAI's peg to the dollar [4]. 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 [1] and the complete Improvement Proposal, detailing implementation, can be also found online [2].

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.

6.1.2 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

[1] https://forum.makerdao.com/t/a-liquidation-system-redesign-a-pre-mip-discussion/2790
[2] https://forum.makerdao.com/t/mip45-liquidations-2-0-liq-2-0-liquidation-system-redesign/6352

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/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 High-level model.

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 in the sections 5.1.3.C and 5.1.3.D.

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. The syntax and semantics of the modifier, included in the *clip.md* file, are presented below:

```
1     syntax ClipStop ::= "noBreaker"
2                       | "noNewKick"
3                       | "noNewKickOrRedo"
4                       | "noNewKickOrRedoOrTake"
5     syntax Bool ::= ClipStop "<ClipStop" ClipStop [function]
6   // ----------------------------------------------------
7     rule noBreaker       <ClipStop noBreaker            => false
8     rule noBreaker       <ClipStop _                    => true   [owise]
9     rule noNewKick       <ClipStop noBreaker            => false
10    rule noNewKick       <ClipStop noNewKick            => false
```

```
11    rule noNewKick       <ClipStop _                      => true   [owise]
12    rule noNewKickOrRedo <ClipStop noNewKickOrRedoOrTake => true
13    rule noNewKickOrRedo <ClipStop _                      => false  [owise]
14    rule _               <ClipStop _                      => false  [owise]
```

Listing 6.1: Circuit breaker modifier definition

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. To satisfy this requirement the file *kmcd-data.md* was extended with:

```
1         syntax Rad ::= Rad "*RadWad2Rad" Wad [function]
2                      | Wad "*WadRay2Rad" Ray [function]
3   // ----------------------------------------------
4       rule FInt(R, RAD) *RadWad2Rad FInt(W, WAD) => FInt(R *Int W /Int WAD, RAD)
5       rule FInt(W, WAD) *WadRay2Rad FInt(R, RAY) => FInt(W *Int R, RAD)
6
7       syntax Wad ::= Rad "/RadRay2Wad" Ray [function]
8   // ----------------------------------------------
9       rule FInt(R1, RAD) /RadRay2Wad FInt(R2, RAY) => FInt(R1 /Int R2, WAD)
```

Listing 6.2: Additional data conversion definition

Besides the new additions to the codebase, this update to the high-level specification also refactored how temporary variables are defined in the K framework. In the imperative programming language that is Solidity, it is regular to use temporary variables that are only available in memory and never saved in storage. However, since K is a declarative language these temporary variables need to be formalized. Previously, the codebase used the *#fun* construct to model these variables but the syntax declarations to use multiple temporary variables required nesting of the construct, resulting in almost illegible code when the use of these variables exceeded an acceptable threshold. During the modeling of the *clip.md* file it became clear that these would need to be replaced by a new, easier to read and more efficient *#let* variable binding that besides making the code cleaner also reduced the compile time of the project by an hour.

6.2 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 4.1.

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.

```
1     syntax Rad ::= calcSumOfDais(Map) [function]
2                  | calcSumOfDaisAux(List, Rad) [function]
3   // ----------------------------------------------------------
4     rule calcSumOfDais(VAT_DAIS) => calcSumOfDaisAux(values(VAT_DAIS), rad(0))
5     rule calcSumOfDaisAux( .List, TOTAL) => TOTAL
6     rule calcSumOfDaisAux( ListItem(AMOUNT) REST, SUM) => calcSumOfDaisAux(REST,
        SUM +Rad AMOUNT)
```

Listing 6.3: Extended measured system events

Finally, the *Fundamental Equation of DAI* is defined as a Finite State Machine in the following manner:

```
1     syntax ViolationFSM ::= "fundamentalDaiEquation"
2   // ------------------------------------------------
3     rule derive(fundamentalDaiEquation, Measure(... debt: DEBT, vice: VICE,
        sumOfDais: SUM))
4     => Violated(fundamentalDaiEquation) requires SUM =/=Rad (DEBT +Rad VICE)
```

Listing 6.4: Fundamental DAI equation definition

As it is shown, 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.

7

Refinement Proofs

Contents

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.

Throughout this thesis we characterized and documented the many components that enable refinements in the scope of Decentralized Finance. As mentioned, refinement proofs are useful to verify that a system model accurately captures the behavior of the implementation. By definition, in order to properly define a refinement, we must have a system, a model, and a system implementation. In the context of this document the system is MakerDAO, which we decomposed into intent, mechanism design, implementation, and the connection between them. In addition, this new documentation which can be found in the chapters Chapter 4, Chapter 5, and Chapter 6, aims to facilitate reasoning about refinements over the protocol. We introduced our documentation of the existing, and further extended, high-level model of the MakerDAO protocol, and also described the KEVM and bytecode implementation of the protocol on the EVM.

After defining these components we are now ready to reason about refinement proofs over the MakerDAO system.

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 [46], 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 Related Work

The concept of refinement and refinement proofs methods and techniques are a widely studied field in Computer Science for guaranteeing system equivalence [47, 48] and correctness [49, 50].

In the area of Smart Contracts and Decentralized Finance some known refinement methods have been applied and studied.

Particularly tools such as Helmholtz [51] and Solid [52] use refinement type systems as intermediary representations for proving smart contract properties correct.

HEVM [44] has also recently released a feature that attempts to prove equivalence of smart contracts by symbolically executing both and comparing the execution results.

Other tools and projects also aim to ensure smart contract properties correct and can be found in section 2.5.1.

7.3 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. [53]. 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.

A bisimulation over the cut states, where the cut denotes a collection of relevant states at which two programs may be synchronized, is known as cut-bisimulation. An example of these cut states in the context of Decentralized Finance protocols is the states at the beginning and end of an EVM transaction. This allows for a simple method of demonstrating equivalence, in which one may examine whether the two programs synchronize at the cut states, which we refer to as synchronization points. The cut may also be tweaked to determine the precision of synchronization points, which are used to indicate the verifiable behaviors of two programs that should be taken into account while determining their equivalence. This helps dealing with intermediate states that aren't crucial to determining program equivalence.

Assume two cut transition systems, one of which replicates the other but not the other. If implemented appropriately, an abstract model cut-simulates its concrete implementation, but the opposite may not be true since the model may omit certain behaviors, leaving them implementation dependent, allowing the implementation to pick any behavior. In this scenario, determining if a model property is also maintained in the implementation is not straightforward. Intuitively, the model's set of all possible cut-states is a superset of the implementation's set. As a result, if a cut-state isn't accessible in the model, it won't be reachable in the implementation either. This means that the model's safety properties are maintained in the implementation, since a safety property may be expressed as "nothing wrong occurs". Inductive invariants are maintained in the refined system in general.

For a thorough formal explanation of the logic behind it the reader can consult the origin cut-bisimulation paper [53].

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.

7.3.1 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.

7.3.1.A 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.

```
1          configuration
2          <kmcd-driver>
3            <return-value> .K </return-value>
4            <msg-sender> 0:Address </msg-sender>
5            <this> 0:Address </this>
6            <current-time> 0:Int </current-time>
7            <mcd-call-stack> .List </mcd-call-stack>
8            <pre-state> .K </pre-state>
9            <events> .List </events>
10           <tx-log> .Transaction </tx-log>
```

```
11            <frame-events> .List </frame-events>
12            <kevm/>
13            <mcd-accounts/>
14          </kmcd-driver>
```

Listing 7.1: Execution framework extension

7.3.1.B 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.

```
1     configuration
2       <mcd-accounts>
3         <mcd-account multiplicity="*" type="Map">
4             <mcd-id> 0:Address </mcd-id>
5             <address> 0 </address>
6         </mcd-account>
7       </mcd-accounts>
```

Listing 7.2: Address translation between models definition

7.3.1.C 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. With modularity in mind we adjust the high-level specification to be amenable to transaction translation in the following manners.

Firstly, we begin by decomposing individual transactions on the high-level specification into appropriate function name and arguments, which will allow us to properly encode the function signature in EVM as well as pack its arguments into the call. Each individual contract function call, *CallStep*, is characterized by its function name, *Op*, and its arguments, *Args*.

```
    syntax Op ::= String
    syntax Arg ::= Bln | Wad | Ray | Rad | Int | String | Address
    syntax Args ::= List{Arg, ""}
    syntax CallStep ::= Op | Op Args
```

Listing 7.3: Transaction decomposition syntactic definition

An example of this function subsorting in the Vat cage function.

```
    syntax CallStep ::= VatStep
    syntax Op       ::= VatOp
    syntax Args     ::= VatArgs
// --------------------------

    syntax VatCageOp ::= "cage"
    syntax VatOp ::= VatCageOp
    syntax VatAuthStep ::= VatCageOp [klabel(#VatCage), symbol]
```

Listing 7.4: Example of function decomposition

Subsequently, it is necessary to encode every high-level argument into the proper EVM bytecode data type. Below we find the translation table for some of the types, omitting the full implementation due to size constraints:

```
  syntax TypedArg ::= #encodeEVM( String, FInt    ) [function]
                    | #encodeEVM( String, String  ) [function]
                    | #encodeEVM( String, Address ) [function]
// ------------------------------------------------
    rule #encodeEVM ( "uint160", FINT:FInt) => #uint160 (value(FINT))
    rule #encodeEVM ( "uint256", FINT:FInt) => #uint256 (value(FINT))

    rule #encodeEVM ( "bytes32", FINT:FInt  ) => #bytes32 (value(FINT))
    rule #encodeEVM ( "bool"   , FINT:FInt  ) => #bool    (value(FINT))
    rule #encodeEVM ( "string" , STR:String) => #string (STR)
```

Listing 7.5: EVM argument types conversion definition

Now that we have decomposed the functions to be refined into the suitable structures the following syntax and semantics are used to encode whole function calls to the proper bytecode translation.

```
    syntax ByteArray ::= #abiEncode(CallStep, List) [function]
                       | #abiEncode(CallStep)       [function]
// ----------------------------------------------------------
    rule #abiEncode ((OP ARGS):CallStep, TYPES ) => #abiCallData(OP, #MCDtoEVM(
      ARGS, TYPES))
    rule #abiEncode ((OP     ):CallStep) => #abiCallData(OP, .TypedArgs)
    syntax TypedArgs ::= #MCDtoEVM    ( Args, List            ) [function]
                       | #MCDtoEVMAux ( Args, List, TypedArgs ) [function]
// ----------------------------------------------------------
    rule #MCDtoEVM(ARGS, TYPES) => #MCDtoEVMAux(ARGS, TYPES, .TypedArgs)
```

Listing 7.6: Translate full transaction between models definition

Finally, the full transactions, defined by the initiating user, function name, and arguments, are serialized and inserted into the correct state configuration cells and ready to be executed by theKEVM in the following step:

```
    syntax KItem ::= #serializeTransaction ( Address, MCDStep )
// ----------------------------------------------------------
    rule <k> #serializeTransaction ( ADDR, (CONTRACT:MCDContract . CALL:CallStep)
      :MCDStep) => #execute  ... </k>
        <account>
            <acctID> CONTRACT_ID </acctID>
            <code> CONTRACT_BIN_RUNTIME </code>
            ...
        </account>
        ( <callState> _ </callState> =>
        <callState>
            <program> CONTRACT_BIN_RUNTIME </program>
            <jumpDests> #computeValidJumpDests(CONTRACT_BIN_RUNTIME) </jumpDests>
            <id> CONTRACT_ID </id>
            <caller> CALLER_ID </caller>
            <callData> #abiEncode(CALL) </callData>
            ...
        </callState> )
        <mcd-account>
            <mcd-id> ADDR </mcd-id>
            <address> CALLER_ID </address>
```

```
21          </mcd-account>
22          <mcd-account>
23              <mcd-id> CONTRACT </mcd-id>
24              <address> CONTRACT_ID </address>
25          </mcd-account>
```

Listing 7.7: Translate function call by user between models definition

In the refinement method, the steps 1 and 2, initiating the transaction from the high-level specification and executing the equivalent transaction over the bytecode are declared below:

```
1       syntax AdminStep ::= "transact" Address MCDStep | "#end-transact"
2   // ----------------------------------------------------------------
3       rule <k> transact ADDR:Address MCD:MCDStep => #runKEVM(ADDR, MCD) ... </k>
4
5       syntax KItem ::= #runKEVM ( Address, MCDStep )
6                      | "#executeKEVM"
7   // ------------------------------------------------
8       rule <k> #runKEVM ( ADDR:Address, MCD:MCDStep ) =>
9          #serializeTransaction ( ADDR, MCD ) ~>
10         #executeKEVM
11         ... </k>
12
13      rule <k> #executeKEVM => #execute ... </k>
14              <evm>
15                  <callData> CALL_DATA </callData>
16                  ...
17              </evm>
18        requires CALL_DATA =/=K .K
```

Listing 7.8: Transaction initation and translation definition

Now that the execution semantics have been refined between specifications we elaborate on how storage is refined and how it is proven equally symbolically updated.

7.3.2 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.

There are two ways this storage refinement can be achieved:

The initial approach would be to manually define the full mapping between the high-level configuration and the low-level EVM bytecode storage. This method is viable and works without having to redefine how the low-level implementation handles bytecode execution and storage manipulation. However, it is unpractical for large protocols that are composed by many contracts, requiring manually mapping and translating values between bytecode and K high-level specification cell configurations.

The second option of refinement, the one we implement, uses 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 [54]. 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 [55–57] 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.

Below we analyze how this refinement is implemented.

7.3.2.A Abstract Storage

This refinement technique works by abstracting the storage for an account on the KEVM implementation. By 'abstracting' we imply subsorting the regular KEVM storage cell, which by default uses a map, to a new sort called *ContractStorage*, which allows using a different data structure for the storage cell. The concept of using an abstract storage for accounts on the KEVM can be easily reused by other high-level K specifications and further research on EVM's storage.

```
configuration
  <kevm>
    <k> $PGM:EthereumSimulation </k>
    <exit-code exit=""> 1 </exit-code>
    <mode> $MODE:Mode </mode>
    <schedule> $SCHEDULE:Schedule </schedule>
    ...
        <accounts>
          <account multiplicity="*" type="Map">
            <acctID>      0                      </acctID>
            <balance>     0                      </balance>
            <code>        .ByteArray:AccountCode </code>
            <storage>     .Map:ContractStorage   </storage>
            <origStorage> .Map:ContractStorage   </origStorage>
            <nonce>       0                      </nonce>
          </account>
        </accounts>
    ...
```

Listing 7.9: Storage abstraction definition in EVM

This new storage abstraction allows using any of the high-level specification contract K cell configurations directly into the KEVM specification, demonstrated below with the vat configuration:

```
  syntax ContractStorage ::= MCDStorage
// -------------------------------------

  syntax MCDStorage ::= #storageVat(VatCell)
```

Listing 7.10: Example Vat abstract storage

The syntax *VatCell* is a direct reference to the Vat configuration declaration, which includes all of the K cells that constitute it. The same mechanisms that K offers for normal cell configuration manipulation are available due to K allowing cell nesting.

This application of the high-level specification configuration as storage in EVM, enabled by K's flexibility when adapting implementations, requires only to define the behavior of reads and writes performed by the KEVM for this type of abstract storage. In the following subsections we exemplify the syntax definitions and semantic behavior required for this particular implementation.

7.3.2.B Storage Reads

The EVM has two opcodes that manipulate the storage: one for reading and loading a word from storage into the stack and another for writing a word to storage.

Below we find the semantic rule that implements the behavior of the load opcode, *SLOAD* in EVM. This rule uses the *#lookup* operator to retrieve a value from storage. The implementation below is unchanged from the original KEVM specification.

```
    syntax UnStackOp ::= "SLOAD"
// ----------------------------
    rule <k> SLOAD INDEX => #lookup(STORAGE, INDEX) ~> #push ... </k>
         <id> ACCT </id>
         <account>
           <acctID> ACCT </acctID>
           <storage> STORAGE </storage>
           ...
         </account>
```

Listing 7.11: Abstract storage load definition override in EVM

In order to lookup values in the new abstract storage we need to introduce a new syntax declaration that allows lookups on arbitrary abstract storage.

```
    syntax Int ::= #lookup ( ContractStorage , Int ) [function, functional,
      smtlib(lookupContractStorage)]
//
      ----------------------------------------------------------------------------------
```

Listing 7.12: Lookup on abstract storage definition

Subsequently it is necessary to provide semantic meaning to the definition of the lookup in the abstract storage. These rules vary depending on the contract configuration declaration. Different types of configurations and particular records on the high-level specification require different rules. As an example, below we can find the rules that allow accessing the different data in an urn in the vat for a particular combination of ilk and user.

```
module VAT-LEMMAS
```

```
2       imports VAT
3
4       syntax Map ::= #lookupMap (Map, Int)    [function, functional]
5     // ----------------------------------------------------------
6       rule #lookupMap( (KEY |-> MAP:Map ) _M, KEY ) => MAP
7       rule #lookupMap(                      M, KEY ) => .Map   requires notBool KEY
          in_keys(M)
8       rule #lookupMap( (KEY |-> VAL     ) _M, KEY ) => .Map   requires notBool
          isMap(VAL)
9
10      syntax VatUrn ::= #lookupUrns (Map, Int) [function, functional]
11      syntax VatUrn ::= "EmptyUrn"
12    // ----------------------------------------------------------------
13      rule  #lookupUrns( (KEY |-> URN:VatUrn ) _M, KEY ) => URN
14      rule  #lookupUrns(                        M, KEY ) => EmptyUrn
15      requires notBool KEY in_keys(M)
16      rule  #lookupUrns( (KEY |-> VAL        ) _M, KEY ) => EmptyUrn
17      requires notBool isVatUrn(VAL)
18
19      syntax Int ::= #lookupUrn (VatUrn, Int) [function, functional]
20    // ----------------------------------------------------------
21      rule #lookupUrn(Urn(... ink: INK), 0) => value(INK)
22      rule #lookupUrn(Urn(... art: ART), 1) => value(ART)
23      rule #lookupUrn(EmptyUrn, _) => 0
24      rule #lookup(#storageVat(<vat> ... <vat-urns> VAT_URNS </vat-urns> ... </vat>
          ), #Vat.urns[ILK][USR].ink ) => #lookupUrn(#lookupUrns(#lookupMap(VAT_URNS,
           ILK), USR), 0) [simplification]
25      rule #lookup(#storageVat(<vat> ... <vat-urns> VAT_URNS </vat-urns> ... </vat>
          ), #Vat.urns[ILK][USR].art ) => #lookupUrn(#lookupUrns(#lookupMap(VAT_URNS,
           ILK), USR), 1) [simplification]
26
27    endmodule
```

Listing 7.13: Example Vat contract storage lookup definition

This process is repeated throughout the high-level specification contracts until the rules fully implement the desired storage lookup behavior.

7.3.2.C Storage Writes

The EVM opcode that writes a word to storage is *SSTORE*. Below we find the modified KEVM rule that allows the definition of particular write operations over an abstract storage, using the new *#write* operator on the storage cell.

```
    syntax BinStackOp ::= "SSTORE"
// ------------------------------
    rule <k> SSTORE INDEX NEW => . ... </k>
         <id> ACCT </id>
         <account>
           <acctID> ACCT                                    </acctID>
           <storage> STORAGE => #write(STORAGE, INDEX, NEW) </storage>
           ...
         </account>
```

Listing 7.14: Abstract storage write definition override in EVM

Similarly to the read operator we need to define the syntax for this *#write* operator so it can then be implemented for each particular contract configuration.

```
    syntax ContractStorage ::= #write ( ContractStorage , Int, Int ) [function,
      functional]
// ------------------------------------------------------------------------------
```

Listing 7.15: Write on abstract storage definition

Afterwards we define the semantics for this *#write* operator. The example below shows the write rules for an user in the Vat wards cell configuration.

```
module VAT-LEMMAS
    imports VAT

    rule #write(#storageVat(<vat> ... <vat-wards> VAT_WARDS </vat-wards> ... </
      vat>), #Vat.wards[A] , 1) => #storageVat(<vat> ... <vat-wards> VAT_WARDS |
      Set SetItem(A) </vat-wards> ... </vat>) [simplification]
```

```
5       rule #write(#storageVat(<vat> ... <vat-wards> VAT_WARDS </vat-wards> ... </
          vat>), #Vat.wards[A] , 0) => #storageVat(<vat> ... <vat-wards> VAT_WARDS -
          Set SetItem(A) </vat-wards> ... </vat>) [simplification]
6
7       ...
8   endmodule
```

Listing 7.16: Example Vat contract storage write definition

7.3.2.D Storage Equivalence

Finally, with this method of storage abstraction it becomes trivial to verify that storage updates performed by the bytecode execution match the intended high-level behavior claims, step 4 of the refinement method. Since the high-level specification configuration for each contract is the same configuration bytecode uses to read and write from, all that is required to verify equivalence of storage is to change all of the high-level specification semantic rules to reachability claims and let the haskell backend of the K framework verify that the symbolic manipulation of storage is equal.

Using Vat Wards as an example, the keyword *rule* is simply replaced with *claim* producing the reachability claims used for proving the refinement:

```
1       claim <k> Vat . rely ADDR => . ... </k>
2             <vat-wards> ... (.Set => SetItem(ADDR)) </vat-wards>
3             <vat-live> true </vat-live>
4
5       claim <k> Vat . deny ADDR => . ... </k>
6             <vat-wards> WARDS => WARDS -Set SetItem(ADDR) </vat-wards>
7             <vat-live> true </vat-live>
```

Listing 7.17: Example Vat reachability claim rewrite

In the subsequent analysis section we also present an example of the usage of this abstract storage refinement method.

This concludes the entire implementation of the refinement proofs. It is noted once more that this refinement method can be easily adapted to other protocols on Ethereum, requiring only to re-define each of the contracts' storage lookup tables, which could be automatized in future work.

With these refinement proofs it is possible to determine exactly what behavior of the implementation of a protocol is captured by the high-level specification. Furthermore, with symbolic execution of bytecode on the KEVM it is possible to know the full behavior set of a protocol on EVM. Theoretically the set

of total possible behaviors of the bytecode minus behaviors captured by the high-level specification is where most of the undefined and possibly faulty behavior of a protocol resides. Behaviors in this space should lead to reverting the transaction execution and therefore not update storage.

8

Analysis

Contents

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 [1], 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 Chapter 6 we executed the already defined randomized concrete tests present in the specification [2], explained in Chapter 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 this section we present an example of a proven claim that uses the abstract storage technique used for refinement presented in the Chapter 7.

We start by formalizing a high-level specification of the flip contract, using a simplified version of the abstract model used in the high-level MakerDAO K specification. This specification configuration declares two variables that are part of the flip storage. It is then declared the abstract storage of this specification to be used by the KEVM.

```
1 module ABSTRACT-FLIP
2     imports INT
3
4 // Flip Configuration
5
6     configuration
7       <flip>
```

[1] https://github.com/makerdao/mkr-mcd-spec/pull/250
[2] https://github.com/makerdao/mkr-mcd-spec/tree/master/tests

```
8             <flip-ttl> 10800:Int  </flip-ttl>
9             <flip-tau> 172800:Int </flip-tau>
10          </flip>
11
12  // Abstract Contract Storage
13
14      syntax ContractStorage ::= #storageFlip(FlipCell)
15   // ----------------------------------------------
16
17  endmodule
```

Listing 8.1: Abstract flip configuration

EVM's storage and stack use 256 bit words. Due to this implementation detail some Smart Contracts have their storage layout designed in a way where for example two *uint48*, unsigned integers represented with 48 bits, are packed together into a single 256 bit EVM word. In the flip contract the *tau* and *ttl* variables are packed together into a single word. This Smart Contract design decision saves on gas costs, since these values are frequently manipulated simultaneously, using only one *SLOAD* opcode instead of two.

As explained beforehand with this storage abstraction technique it is necessary to define the syntax and semantic rules that allow the KEVM to operate over it. In this special case of word packing in the EVM the rules that allow the *ttl* and *tau* values to be looked up individually are:

```
1  module WORD-PACK-HASKELL [kore]
2      imports WORD-PACK-COMMON
3
4      rule maxUInt48 &Int #lookup(#storageFlip(<flip> ... <flip-ttl> TTL </flip-ttl
         > ... </flip>), #Flipper.ttl_tau) => TTL
5      [simplification]
6
7      rule maxUInt48 &Int (#lookup(#storageFlip(<flip> ... <flip-tau> TAU </flip-
         tau> ... </flip>), #Flipper.ttl_tau) /Int pow48) => TAU
8      [simplification]
```

Listing 8.2: Concrete flip reads on abstract storage

Finally, we prove that the KEVM can effortlessly lookup the desired storage values over the abstract storage model. This is achieved using an already proven claim in the KEVM codebase, but with only

one minor change, the account storage for flip uses the abstract storage refinement. The claim is shown below, with the abstract storage in the appropriate storage cell.

```
requires "verification.k"

module ABSTRACT-FLIPPER-TAU-PASS-SPEC
    imports VERIFICATION

    // ABSTRACT Flipper_tau
    claim [ABSTRACT.Flipper.tau.pass]:
      <k> #execute ~> CONTINUATION => #halt ~> CONTINUATION </k>
      <exit-code> 1 </exit-code>
      <mode> NORMAL </mode>
      <schedule> ISTANBUL </schedule>
      <ethereum>
        <evm>
          <output> _ => #buf(32, Tau) </output>
          <statusCode> _ => EVMC_SUCCESS </statusCode>
          <endPC> _ => ?_ </endPC>
          <callStack> _VCallStack </callStack>
          <interimStates> _ </interimStates>
          <touchedAccounts> _ => ?_ </touchedAccounts>
          <callState>
            <program> Flipper_bin_runtime </program>
            <jumpDests> #computeValidJumpDests(Flipper_bin_runtime) </jumpDests>
            <id> ACCT_ID </id>
            <caller> CALLER_ID </caller>
            <callData> #abiCallData("tau", .TypedArgs) ++ CD => ?_ </callData>
            ...
          <accounts>
            <account>
              <acctID> ACCT_ID </acctID>
              <balance> ACCT_ID_balance </balance>
              <code> Flipper_bin_runtime </code>
              <storage> #storageFlip( <flip> FLIP </flip> ) </storage>
              <origStorage> _ACCT_ID_ORIG_STORAGE </origStorage>
              <nonce> _Nonce_Flipper => ?_ </nonce>
            </account>
```

```
36                ...
37              </accounts>
```

Listing 8.3: Example of using abstract storage in EVM and proving a reachability claim

This example exhibits how straightforward it is to use the abstract storage when proving claims.

9

Conclusions

Contents

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 are also correct over the bytecode implementation of the system.

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.

Bibliography

[1] R. Verification, https://runtimeverification.com/, 2021.

[2] 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.

[3] Vaibhav, https://github.com/vasa-develop/defi-diagrams/, 2020.

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

[5] 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

[6] 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.

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

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

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

[10] 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

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

[12] MakerDAO, https://makerdao.com/en/, 2021.

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

[14] 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.

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

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

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

[18] H. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, pp. 358–366, 1953.

[19] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," 2020.

[20] A. Singh, R. Parizi, Q. Zhang, K.-K. R. Choo, and A. Dehghantanha, "Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities," *Computers Security*, vol. 88, p. 101654, 10 2019.

[21] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "Verification of smart contracts: A survey," *Pervasive and Mobile Computing*, vol. 67, p. 101227, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574119220300821

[22] N. Sánchez-Gómez, J. Torres-Valderrama, J. A. García-García, J. J. Gutiérrez, and M. J. Escalona, "Model-based software design and testing in blockchain smart contracts: A systematic literature review," *IEEE Access*, vol. 8, pp. 164 556–164 569, 2020.

[23] Z. Nehaï, P. Piriou, and F. Daumas, "Model-checking of smart contracts," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 980–987.

[24] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," 01 2018.

[25] K. Nelaturu, A. Mavridoul, A. Veneris, and A. Laszka, "Verified development and deployment of multiple interacting smart contracts with verisolid," in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2020, pp. 1–9.

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

[27] W. Ahrendt and R. Bubel, “Functional verification of smart contracts via strong data integrity,” in *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2020, pp. 9–24.

[28] Z. Yang, H. Lei, and W. Qian, “A hybrid formal verification system in coq for ensuring the reliability and security of ethereum-based service smart contracts,” *IEEE Access*, vol. PP, pp. 1–1, 01 2020.

[29] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, “Semantics-based program verifiers for all languages,” *SIGPLAN Not.*, vol. 51, no. 10, p. 74–91, Oct. 2016. [Online]. Available: https://doi.org/10.1145/3022671.2984027

[30] M. Mandrykin, J. O’Shannessy, J. Payne, and I. Shchepetkov, “Formal specification of a security framework for smart contracts,” 2020.

[31] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding the greedy, prodigal, and suicidal contracts at scale,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 653–663. [Online]. Available: https://doi.org/10.1145/3274694.3274743

[32] D. Annenkov, M. Milo, J. B. Nielsen, and B. Spitters, “Verifying, testing and running smart contracts in concert.”

[33] A. Li, J. A. Choi, and F. Long, “Securing smart contract with runtime validation,” *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 2020. [Online]. Available: http://dx.doi.org/10.1145/3385412.3385982

[34] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, “Verx: Safety verification of smart contracts,” 05 2020, pp. 1661–1677.

[35] D. Park, Y. Zhang, and G. Rosu, “End-to-end formal verification of ethereum 2.0 deposit smart contract,” in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 151–164.

[36] G. Rosu, “K framework - an overview,” https://runtimeverification.com/blog/k-framework-an-overview/, 2018.

[37] 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

[38] 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.

[39] R. Christensen, “Makerdao has come full circle,” https://blog.makerdao.com/makerdao-has-come-full-circle//, 2021.

[40] ——, “Introducing edollar, the ultimate stablecoin built on ethereum,” https://www.reddit.com/r/ethereum/comments/30f98i/introducing_edollar_the_ultimate_stablecoin_built/, 2014.

[41] MakerDAO, https://github.com/makerdao, 2021.

[42] ——, https://vote.makerdao.com/, 2021.

[43] D. D. McCracken and E. D. Reilly, “Backus-naur form (bnf),” 2003.

[44] DappHub, https://github.com/dapphub/dapptools/tree/master/src/hevm, 2021.

[45] Lucash, https://hackerone.com/lucash-dev, 2019.

[46] 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.

[47] V. Cheval, H. Comon-Lundh, and S. Delaune, “Automating security analysis: Symbolic equivalence of constraint systems,” in *Proceedings of the 5th International Conference on Automated Reasoning*, ser. IJCAR’10, 2010, p. 412–426. [Online]. Available: https://doi.org/10.1007/978-3-642-14203-1_35

[48] T. Matsumoto, H. Saito, and M. Fujita, “Equivalence checking of c programs by locally performing symbolic simulation on dependence graphs,” 04 2006, pp. 6 pp.–.

[49] J. von Wright, “Program refinement by theorem prover,” 1994.

[50] M. Mumme and G. Ciardo, “A fully symbolic bisimulation algorithm,” vol. 6945, 09 2011, pp. 218–230.

[51] Y. Nishida, H. Saito, R. Chen, A. Kawata, J. Furuse, K. Suenaga, and A. Igarashi, *Helmholtz: A Verifier for Tezos Smart Contracts Based on Refinement Types*, 03 2021, pp. 262–280.

[52] B. Tan, B. Mariano, S. Lahiri, I. Dillig, and Y. Feng, “Soltype: Refinement types for solidity,” 2021.

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

[54] E. Foundation, https://docs.soliditylang.org/en/v0.8.9/internals/layout_in_storage.html, 2021.

[55] D. Park, https://github.com/ethereum/deposit_contract/issues/27, 2019.

[56] ——, https://github.com/ethereum/deposit_contract/issues/28, 2019.

[57] ——, https://github.com/ethereum/deposit_contract/issues/38, 2019.

Most prominent DeFi hacks of 2020

DeFi Hacks 2020

Date	Protocol	Damage (in $)
Feb. 14/18	bZx	954000$
Apr. 19	Lendf.me	25 million $
Mar.12	Maker	8 million $
Aug. 11	YAM	750000$
Sept. 13	bZx	8 million $
Sept. 28	Eminence	15 million $
Oct. 26	Harvest	24 million $
Nov. 9	SharkTron	260 million $
Nov. 21	Pickle	19 million $
Dec. 28	Cover	5 million $

Figure A.1: Most prominent DeFi hacks of 2020.

B

Overview of Pickle Finance exploit

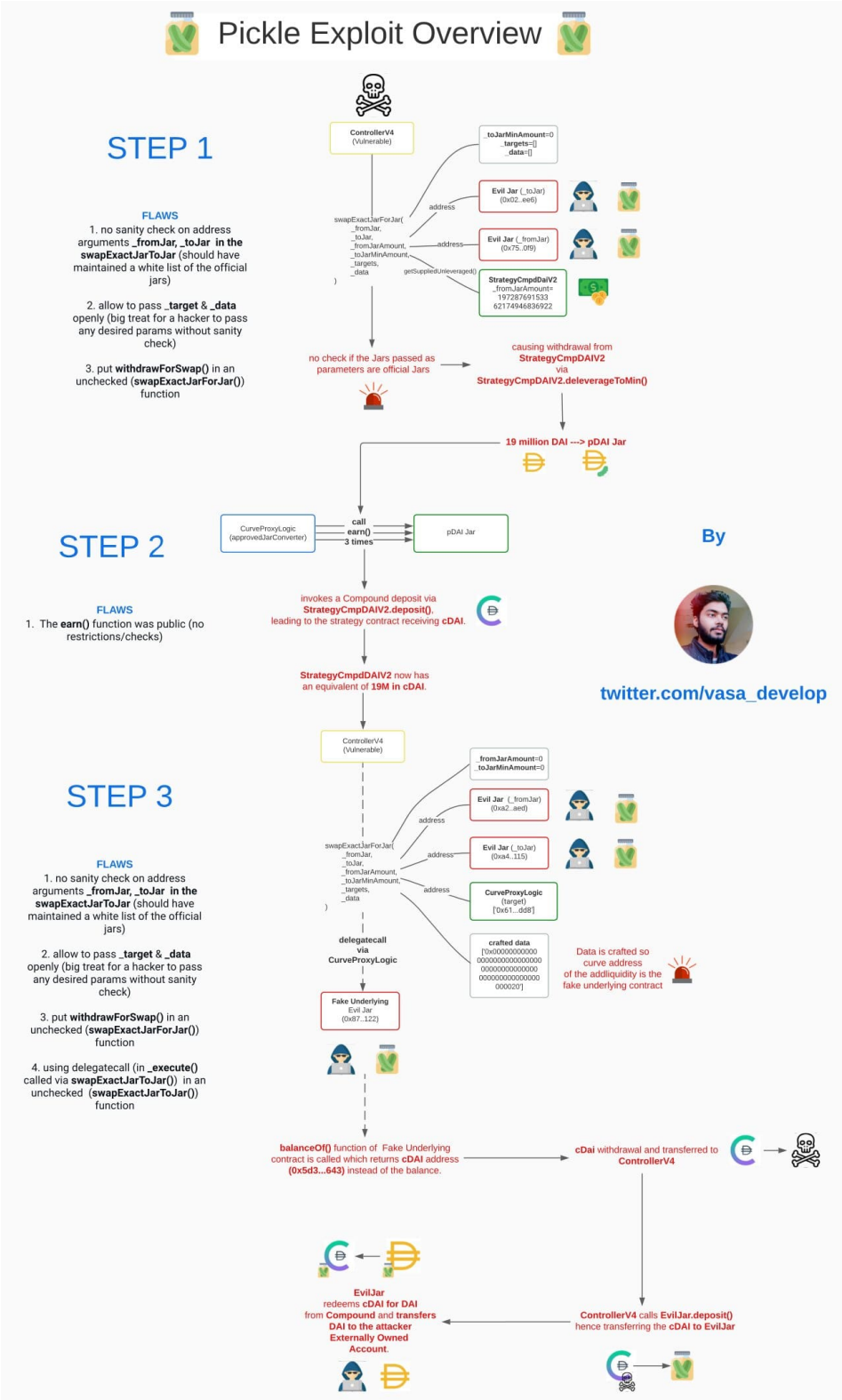

Figure B.1: Overview of Pickle Finance exploit, 19.7 million dollars where stolen. [3]

C

KEVM configuration

```
configuration
  <kevm>
    <k> $PGM:EthereumSimulation </k>
    <exit-code exit=""> 1 </exit-code>
    <mode> $MODE:Mode </mode>
    <schedule> $SCHEDULE:Schedule </schedule>

    <ethereum>

      // EVM Specific
      // ============

      <evm>

        // Mutable during a single transaction
```

```
16              // ------------------------------------
17
18              <output>           .ByteArray  </output>           // H_RETURN
19              <statusCode>       .StatusCode </statusCode>
20              <endPC>            0           </endPC>
21              <callStack>        .List       </callStack>
22              <interimStates>    .List       </interimStates>
23              <touchedAccounts>  .Set        </touchedAccounts>
24
25              <callState>
26                <program> .ByteArray </program>
27                <jumpDests> .Set </jumpDests>
28
29                // I_*
30                <id>         .Account   </id>                   // I_a
31                <caller>     .Account   </caller>               // I_s
32                <callData>   .ByteArray </callData>             // I_d
33                <callValue>  0          </callValue>            // I_v
34
35                // \mu_*
36                <wordStack>    .WordStack </wordStack>          // \mu_s
37                <localMem>     .Memory    </localMem>           // \mu_m
38                <pc>           0          </pc>                 // \mu_pc
39                <gas>          0          </gas>                // \mu_g
40                <memoryUsed>   0          </memoryUsed>         // \mu_i
41                <callGas>      0          </callGas>
42
43                <static>     false </static>
44                <callDepth>  0     </callDepth>
45              </callState>
46
47              // A_* (execution substate)
48              <substate>
49                <selfDestruct>      .Set  </selfDestruct>          // A_s
50                <log>               .List </log>                   // A_l
51                <refund>            0     </refund>                // A_r
52                <accessedAccounts>  .Set   </accessedAccounts>
53                <accessedStorage>   .Map   </accessedStorage>
```

```
54              </substate>
55
56              // Immutable during a single transaction
57              // ------------------------------------
58
59              <gasPrice> 0        </gasPrice>                    // I_p
60              <origin>   .Account </origin>                      // I_o
61
62              // I_H* (block information)
63              <blockhashes> .List </blockhashes>
64              <block>
65                <previousHash>     0          </previousHash>     // I_Hp
66                <ommersHash>       0          </ommersHash>       // I_Ho
67                <coinbase>         0          </coinbase>         // I_Hc
68                <stateRoot>        0          </stateRoot>        // I_Hr
69                <transactionsRoot> 0          </transactionsRoot> // I_Ht
70                <receiptsRoot>     0          </receiptsRoot>     // I_He
71                <logsBloom>        .ByteArray </logsBloom>        // I_Hb
72                <difficulty>       0          </difficulty>       // I_Hd
73                <number>           0          </number>           // I_Hi
74                <gasLimit>         0          </gasLimit>         // I_Hl
75                <gasUsed>          0          </gasUsed>          // I_Hg
76                <timestamp>        0          </timestamp>        // I_Hs
77                <extraData>        .ByteArray </extraData>        // I_Hx
78                <mixHash>          0          </mixHash>          // I_Hm
79                <blockNonce>       0          </blockNonce>       // I_Hn
80
81                <ommerBlockHeaders> [ .JSONs ] </ommerBlockHeaders>
82              </block>
83
84            </evm>
85
86            // Ethereum Network
87            // ================
88
89            <network>
90
91              // Chain identifier
```

```
92              // -----------------
93              <chainID> $CHAINID:Int </chainID>
94
95              // Accounts Record
96              // ---------------
97
98              <activeAccounts> .Set </activeAccounts>
99              <accounts>
100               <account multiplicity="*" type="Map">
101                 <acctID>      0                     </acctID>
102                 <balance>     0                     </balance>
103                 <code>        .ByteArray:AccountCode </code>
104                 <storage>     .Map                  </storage>
105                 <origStorage> .Map                  </origStorage>
106                 <nonce>       0                     </nonce>
107               </account>
108             </accounts>
109
110             // Transactions Record
111             // -------------------
112
113             <txOrder>   .List </txOrder>
114             <txPending> .List </txPending>
115
116             <messages>
117               <message multiplicity="*" type="Map">
118                 <msgID>      0          </msgID>
119                 <txNonce>    0          </txNonce>            // T_n
120                 <txGasPrice> 0          </txGasPrice>         // T_p
121                 <txGasLimit> 0          </txGasLimit>         // T_g
122                 <to>         .Account   </to>                 // T_t
123                 <value>      0          </value>              // T_v
124                 <sigV>       0          </sigV>               // T_w
125                 <sigR>       .ByteArray </sigR>               // T_r
126                 <sigS>       .ByteArray </sigS>               // T_s
127                 <data>       .ByteArray </data>               // T_i/T_e
128                 <txType>     0          </txType>
129                 <txAccess>   [ .JSONs ] </txAccess>
```

```
130                 <txChainID>  0          </txChainID>
131               </message>
132             </messages>
133
134           </network>
135
136         </ethereum>
137       </kevm>
138
139     syntax EthereumSimulation
140     syntax AccountCode ::= ByteArray
141   // ---------------------------------
```

D

MakerDAO Actors and their Goals, Obligations, Punishments, Incentives, Required Knowledge, Risk and Interaction with the MakerDAO Protocol

Table D.1: Subsets of Actors and their required knowledge.

ID	Actors			*Required Knowledge*
A	***System Users***		Dai Holders	Basic, should know DAI is stable
B			Vault Owners	Intermediate, must know vault dynamics
C	***MKR Holders***	***Examples of MKR Holders responsibilities & categories***		Advanced, should know all system dynamics
D		Core Unit Teams	Risk	Advanced, must know all system dynamics
E			Oracles	**Oracle Updates: Medium, must know how to update collateral price** **Team: Advanced, must know all system dynamics**
F			Real World Finance	Advanced, must know all system dynamics
G			Growth	Advanced, must know all system dynamics
H			Protocol Engineering	Advanced, must know all system dynamics
I			GovAlpha	Advanced, must know all system dynamics
J			Content Production	Medium, must know most system dynamics
L			Sustainable Ecosystem Scaling	Medium, must know most system dynamics
M			Governance Communications	Medium, must know most system dynamics
N			Dai Foundation	Medium, must know most system dynamics
O	**Keepers**	Entities that permissionlessly execute on-chain actions necessary for the protocol to function. In the event of Keeper incentive failure, MKR holders must be ready to step up and divert system funds to ensure that necessary actions are taken.		Advanced, must know most system dynamics

Table D.2: Actors and their goals, obligations and punishments.

ID	*Goals*	*Obligations*	*Punishment*
A	Own decentralized stablecoin	/	/
B	Speculate	Repay Debt	Incapable of retrieving collateral
C	Monitor and change system	- Vote on Spells - Trigger ESM	/
D	Ensure Maker Protocol's risk profile is mitigated at all times	- Calculations and proposed adjustments of parameters - Risk metrics of cypto collateral debt exposure	Removed from team
E	Minimize external dependencies to reduce the attack surface and maximize resiliency of the system	Bring off-chain information to on-chain system	Removed from team
F	Pursue Real-World Asset work and more generally helps MakerDAO to take over the traditional finance world.	- Investing in Real-World Assets - Yield curve implications in DeFi - Accounting, solvency, potential regulatory reporting	Removed from team
G	Grow the available distribution channels for the Maker protocol	- Give support and education to drive Dai adoption and integrations. - Generate expansion and adoption of Dai. - Develop an integration strategy and give continuous advice to the newest distribution channels.	Removed from team
H	Engineering, security, research and smart contract development experience to ensure that the Maker protocol can safely continue to grow	- Extending the functionality of the protocol - Assisting in the maintenance and operation of existing smart contracts - Ensuring the safety and correctness of the protocol	Removed from team
I	Consistent and well-run governance process	- Remain neutral and objective on issues outside of the governance domain and focus on the facilitation of governance processes - Voice opinions on issues related to the structure and processes of governance within MakerDAO	Removed from team
J	Enhance MakerDAO's position as a reputable authority on topics like decentralized governance, token engineering, and DeFi	- Produce entertaining and educational content that promotes engagement with Dai and the Maker Protocol. - Produce promotional content for other Core Units and provide resources to educate the ecosystem on best practices in content production and distribution.	Removed from team
L	Grow the Maker Protocol's moats by removing barriers between decentralized workforce, capital, and work.	- Identify scaling bottlenecks - R&D on removing these bottlenecks - Fund possible solutions	Removed from team
M	Focus on improving MakerDAO as a public organization, not just as a technology	- Aggregate and simplify information available on resources and services to stakeholders	Removed from team
N	Protect Maker's intangible assets and prepare to mitigate problems in worst case scenarios	- Safeguarding of new intellectual property that is added to the Protocol - Create transparency around how the usage rights of the Maker Assets are allocated	Removed from team
		Permissionless possible actions	
O	- Execute proposals - Provide balance to system discrepancies - Provide liquidity to collateral auctions	- Auctions - Ilk Rates Updates - Arbitrage - Execute spells	/

Table D.3: Actors and their benefits and incentives, their risks and risk assessment, and interactions with the MakerDAO protocol.

ID	*Benefits/Incentives*	*Risk*	*Risk Assessment*	*Interacts With*
A	Stability	DAI falling off peg	LOW	Money Markets
B	Borrow	- Collateral Price dump liquidates collateral - DAI rising over the peg and becoming extra expensive to repay debt	MEDIUM	Vaults
C	- Better functioning of the system will eventually accrue value to MKR holders	- Decisions might also have negative influence on the system which will devalue the MKR token, system cash-flows or lead to insolvency of DAI	MEDIUM	Governance Module
D	Paid by the system on schedule	Not meeting deadlines/proposed goals		Governance Module
E	**Team: Paid by the system on schedule** **Oracle Updates: Paid by the system per oracle update**	Not meeting deadlines/proposed goals		- Oracle Module - Governance Module
F	Paid by the system on schedule	Not meeting deadlines/proposed goals		Governance Module
G	Paid by the system on schedule	Not meeting deadlines/proposed goals		Governance Module
H	Paid by the system on schedule	Not meeting deadlines/proposed goals		Whole System (mainly Vat and Gov)
I	Paid by the system on schedule	Not meeting deadlines/proposed goals		- Governance Module - Community
J	Paid by the system on schedule	Not meeting deadlines/proposed goals		- Community - Media
L	Paid by the system on schedule	Not meeting deadlines/proposed goals		- Governance Module - Community
M	Paid by the system on schedule	Not meeting deadlines/proposed goals		- Community
N	Paid by the system on schedule	Not meeting deadlines/proposed goals		- Governance Module - Community
O	- Bounties - Discounted DAI - Discounted Collateral - Discounted MKR	/	VERY LOW	- Governance Module - System Stabilizer Module - Rates Module

E

MakerDAO Smart Contracts Implementation Diagram

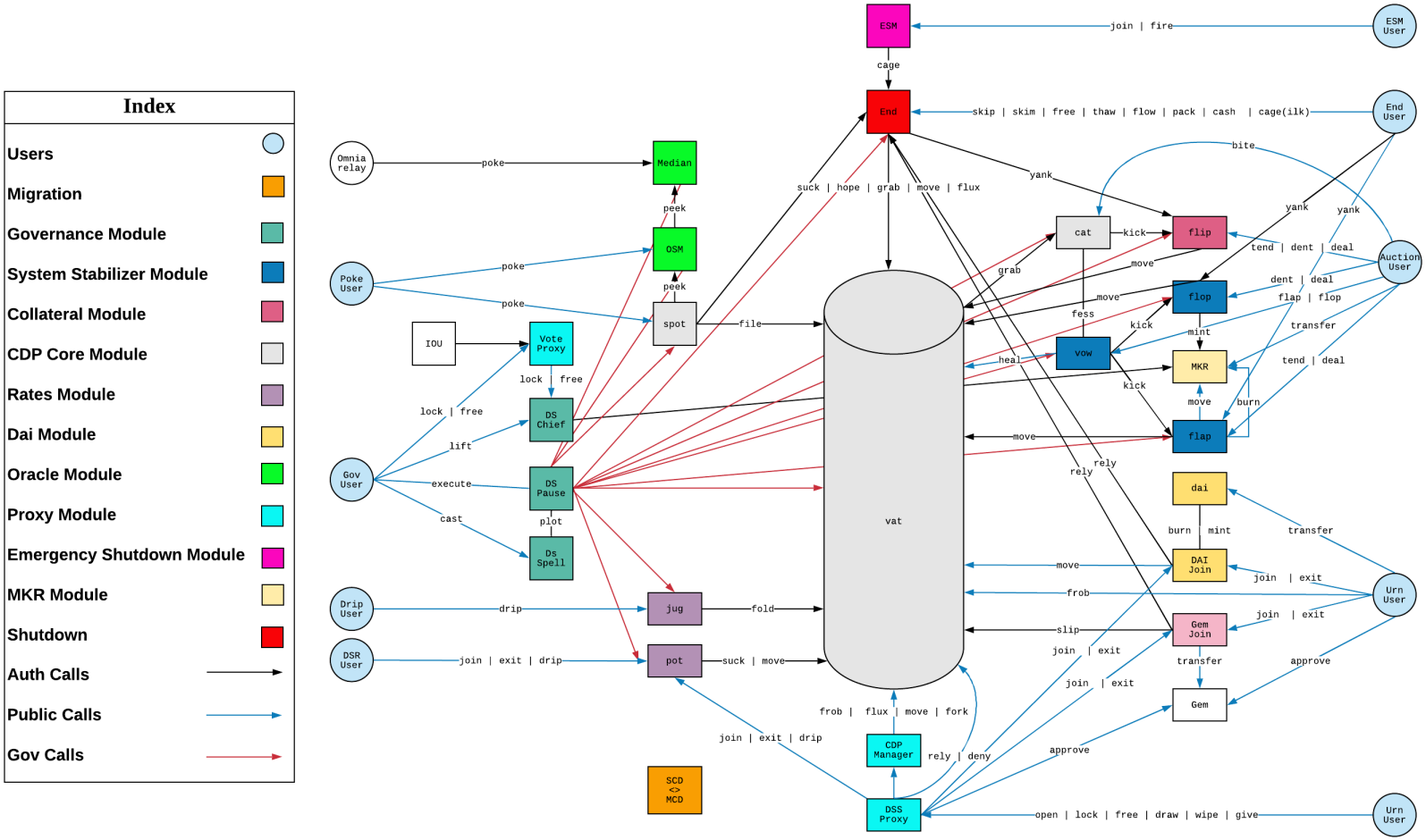

Figure E.1: Smart contracts implementation diagram. [4]

F

Related Work Taxonomy

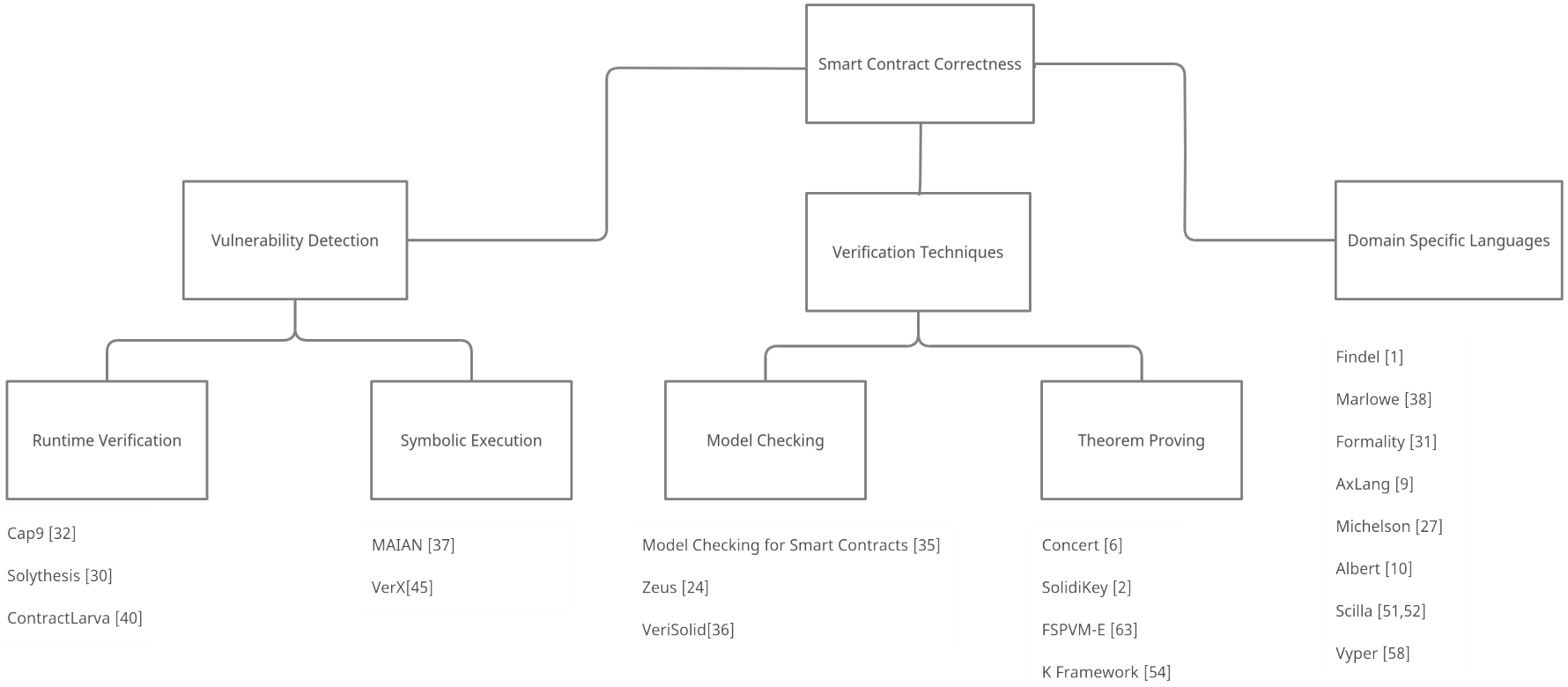

Figure F.1: Taxonomy of Frameworks and Tools on Formal Verification of Smart Contracts.