VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
DEPARTMENT OF SOFTWARE ENGINEERING

Formal Verification of IOTA UTXO Output Type Extensions

Formali IOTA UTXO išvesčių tipų praplėtimų verifikacija

Master's Final Thesis

Author: Edvardas Dlugauskas (signature)
Supervisor: Dr. Karolis Petrauskas (signature)
Reviewer: Dr. Linas Laibinis (signature)

Vilnius – 2024

CONTENTS

INTRODUCTION 3

1. LITERATURE REVIEW 6
   1.1. An introduction to formal methods 6
   1.2. Overview of a formalization process 7
      1.2.1. Translating requirements into formal specifications 9
   1.3. Blockchains 9
      1.3.1. Account model 11
      1.3.2. UTXO model 12
      1.3.3. Smart contracts 14
      1.3.4. Native user-defined tokens 15
      1.3.5. Extended UTXO model 16
   1.4. IOTA EUTXO model 16
   1.5. State of the art 19
      1.5.1. Overview of the use of formal methods in the blockchain industry 19
      1.5.2. Attempts at formalizing the UTXO and EUTXO models 21
      1.5.3. Cardano UTXO model specification in Isabelle/HOL 22
   1.6. Overview of Isabelle/HOL 23
      1.6.1. Comparison with other proof assistants 23
      1.6.2. Isabelle vs. Isabelle/HOL 24
      1.6.3. Development environment and automated proofs 24
      1.6.4. Isabelle/HOL syntax overview 25
      1.6.5. Locales in Isabelle/HOL 26
   1.7. Conclusions 27

2. MODELING UTXO IN ISABELLE/HOL 29
   2.1. Overview of the UTXO model 29
      2.1.1. Essential entities of the UTXO model 29
      2.1.2. Essential properties of the UTXO model 30
   2.2. Hashing 32
      2.2.1. Type of a unique identifier 33
      2.2.2. Identifier uniqueness 34
      2.2.3. Modeling identifier uniqueness using a type class 35
      2.2.4. Modeling identifier uniqueness using a locale 36
   2.3. Subtyping 37
      2.3.1. Producing a modular and extendable model 37
      2.3.2. Modeling subtypes as disjoint subsets of different types 38
      2.3.3. Modeling functions on subtypes as relations 39
      2.3.4. Using quotient types 39
      2.3.5. Using locales 40
   2.4. UTXO model in Isabelle/HOL 40
      2.4.1. Overview 40
      2.4.2. Abstract UTXO model specification 41
      2.4.3. Implementation UTXO model specification 43
      2.4.4. Verification of UTXO model properties 44

3. MODELING IOTA EUTXO IN ISABELLE/HOL 46
   3.1. Scope of modeling 46
   3.2. Formalizing the IOTA UTXO extensions 47
      3.2.1. Entities of the IOTA EUTXO model 47

3.2.2. Properties of the IOTA EUTXO model 51
3.3. IOTA EUTXO model in Isabelle/HOL 53
3.3.1. Modeling UTXO extensions in Isabelle/HOL in a modular way 54
3.3.2. Abstract Alias UTXO 54
3.3.3. Foundry UTXO 56
3.3.4. Implementation model 60
3.3.5. Verification of IOTA EUTXO model's properties 62
3.3.6. Comparison with Cardano UTXO model 64
RESULTS 66
CONCLUSIONS 66
REFERENCES 67
APPENDICES 73
APPENDIX A. ISABELLE/HOL CODEBASE WALKTHROUGH 73

Introduction

Topic relevance

The popularity of proof-of-work cryptocurrencies, such as Bitcoin, is on the rise [HR17]. With this increase in the number of transactions comes the need for efficient, and subsequently, more environmentally friendly [KT18], distributed consensus algorithms. IOTA is a distributed ledger technology that aims to provide fast and efficient transactions which could be used for micropayments in the Internet of Things (IoT) industry [Pop].

IOTA is based on a specialized directed graph referred to as a *tangle*. The transactions are represented as vertices in the tangle. Whenever a new transaction is added to the tangle, it is required to approve two previous unapproved transactions (referred to as *tips*), which adds two new edges to the tangle with the new transaction being the tail in those edges and the new transactions being the heads [Pop]. Furthermore, transactions in IOTA follow the *unspent transaction output* (UTXO) model – the inputs of a transaction consume unspent outputs of previous transactions [Pop]. This is in contrast with an account-based model where the account balances are manipulated directly [But+14]. In IOTA's UTXO model, to use an unspent output, the transaction has to prove ownership of the address the output was sent to; this is done in the *unlock block* part of the transaction [Pop].

Native tokenization and smart contracts are both important functionalities for IoT and are already present in other distributed ledger technologies, such as the Ethereum blockchain [CCM+20a; CD16]. At the time of writing IOTA's UTXO model does not support either of these features. IOTA's RFC 38 "Output Types for Tokenization and Smart Contracts" is a design document that proposes extensions to the IOTA's UTXO model to support native tokenization and smart contracts [Pap21]. To allow IOTA to be a multi-asset network, RFC 38 proposes new output types that would carry user-defined tokens (referred to as *native tokens*) [Pop]. Additionally, new types of unlock blocks are proposed, which would allow validation for different token types, such as *non-fungible token* (NFT) [Pap21]. The proposed *feature block* would allow extra logic to be added to the transactions to achieve functionality similar to a smart-contract [Pap21].

While RFC 38 document describes suggested changes in detail and provides the rationale behind them, it is left unclear whether the proposed changes are sound as no formal verification of the modified system is provided. Formal verification of the proposed changes proving that the modified system is sound and maintains all of the desired properties would increase stakeholders' confidence in the proposal. Alternatively, an example of a modified system's undesired behavior could help expose a vulnerability or unexpected use case or simply identify a bug introduced by the changes.

The possibility of unexpected bugs being found when smart contracts are involved is not unfounded. One of the most famous examples of a known attack being exploited is the exploitation of a vulnerability in the *TheDAO* smart contract, which resulted in more than $50M worth of Ether being stolen [BDF+16b]. Moreover, a recent large-scale analysis of unique contracts used on the Ethereum blockchain found that more than eight hundred of them had at least one

vulnerability [KR18].

There have been numerous attempts at formally verifying distributed ledger technologies. Abstract models of UTXO based ledgers were defined [Zah18a; Zah18b]. The *F** language has been used to verify smart contracts for Ethereum [BDF+16b]. The Agda language and proof assistant have been used to formalize the extended UTXO ledger and the BitML calculus [Mel19]. Agda has also been used to formalize the proposal for multi-asset support in Cardano [CCM+20c]. Moreover, some other attempts at formal verification can be found: a Cardano UTXO model specification written in Isar can be found in Github; a TLA+ specification of the Tendermint consensus protocol is also in Github [Kon].

One possible way to formally verify the changes proposed in RFC 38 is by using the formal specification language Isar and proof assistant Isabelle/HOL. The Isabelle tool collection stands out from other formalization tools by having powerful proof automation instruments such as Sledgehammer while allowing proofs to be expressed using classical logic [NPW02]. Using Isar, a formal specification of the IOTA tangle network modified according to the changes suggested in RFC 38 can be provided. With the help of Isabelle/HOL proof assistant, it can then be shown that the modified system is functionally correct or that one or more of the desired properties of the system are not maintained.

Aim and tasks

The thesis aims to prove the functional correctness of the changes proposed in RFC 38. If, instead, the changes are proven to be functionally incorrect, then the cases where undesired behavior would occur should be demonstrated – for every property that is shown to be no longer maintained, the causes of the undesired behavior will be discussed and suggestions will be provided to the authors of RFC 38.

Subsequently, to achieve the aim of the thesis, the following tasks will be performed:

1. A literature survey – analysis of the state of the art research performed in the field of formalization of distributed ledger technologies;

2. Analysis of the structure of the IOTA network and the changes proposed in RFC 38 “Output Types for Tokenization and Smart Contracts”;

3. Using knowledge gathered during the literature survey and analysis of RFC 38, formalization of the IOTA EUTXO model with the changes described RFC 38 as well as its desired properties in Isar;

4. Evaluation of the adequacy of the model created in the previous task by comparing it to previous attempts discovered while performing the literature survey. This will ensure that the produced model is relevant and complete;

5. Verification of the formal specification produced by the formalization task using Isabelle/HOL;

6. Analysis of the results of formal verification of the model performed in the previous task, identification of undesired scenarios, and formulation of suggestions to the RFC 38 authors.

Research methods

Firstly, a literature survey will be performed to identify research papers, textbooks, and other media to gain a comprehensive view of the research performed in the area of distributed ledger technologies. Formal modeling will then be performed to produce a model of the system under discussion. The model's adequacy will then be evaluated by cross-checking it with the models produced by previous research in the field. Finally, formal verification will be performed on the model to check whether it maintains the correctness properties.

Expected results

The expected results of the thesis are:

1. A formal specification of the IOTA EUTXO model modified according to RFC 38 in Isabelle/HOL;

2. The results of verification of the formal specification using Isabelle/HOL and suggestions to the RFC 38's authors (if any).

1. Literature review

1.1. An introduction to formal methods

As software becomes more intertwined in our daily lives, so rises the responsibility of software to be safe, reliable, and secure. While a failure in a food ordering system might cause a headache, even a rare bug in an airliner's autopilot software can come at the cost of hundreds of human lives [HBM20]. Subsequently, methods of verifying software, such as formal modeling, are becoming increasingly relevant and even mainstream [Rus07].

Validation and verification of software is a standard step in almost all software development processes. For critical software, more than half of the development's cost goes to verification and validation. Furthermore, fixing an issue that has been discovered late in the development process is more costly than correcting it in an earlier stage [Cou+07]. Perhaps the most common software verification and validation method is simulation and testing. In many standard software development models, such as the V-Model, the testing is performed after the related software system or component is already developed [BTG83; Rup10]. While simulation and testing aim to ensure that as many issues as possible are identified and addressed before the software is released, these methods are often not exhaustive and are labor-intensive. Moreover, finding issues after the system has already been designed and developed incurs a large overhead – code would need to be rewritten and a redesign of the related system or component might be required. Formal methods provide an elegant solution to these problems.

Formal methods are not based on the simulation of a system or the execution of code. Instead, they are based on a mathematical model and a mathematical analysis of the code. Formal methods provide a technique to specify and check the properties of a system mathematically [Win90]. In a V-Model, all of this can be done after the design stage and before any of the system's runtime code has been written. Artifacts created early in the lifecycle of a system such as outline designs, requirements, and specifications can all be useful for a formal analysis [Rus07].

A formal method is defined as a combination of a formal notation (syntax) and a formal analysis technique (solver). Two main analysis techniques are often used: model checking and deductive methods [Win90]. Model checking involves exploring the state space of a model's behavior to check whether the system's properties are maintained in every reachable state. An example of a language with a model checker is the TLA$^+$ specification language with the TLC Model Checker [YML99]. Deductive techniques use mathematical reasoning to outright prove a property of the formal model [Win90]. Examples of formalization tools using deductive techniques include Coq, Isabelle, ACL2, and Ada [HM05; Rus07].

More than half a century has passed since formal methods for software have first been investigated [Flo93]. Since then, numerous improvements to the usability and tooling of formalization software have been made [Rus07]. Nowadays, most of the application of formal methods in the industry of software engineering involves fully automatic but very specialized analysis of individual program units. Nevertheless, even the use of interactive theorem provers such as Isabelle and Coq requires a high investment of skill and time, which limits their use to the formalization of

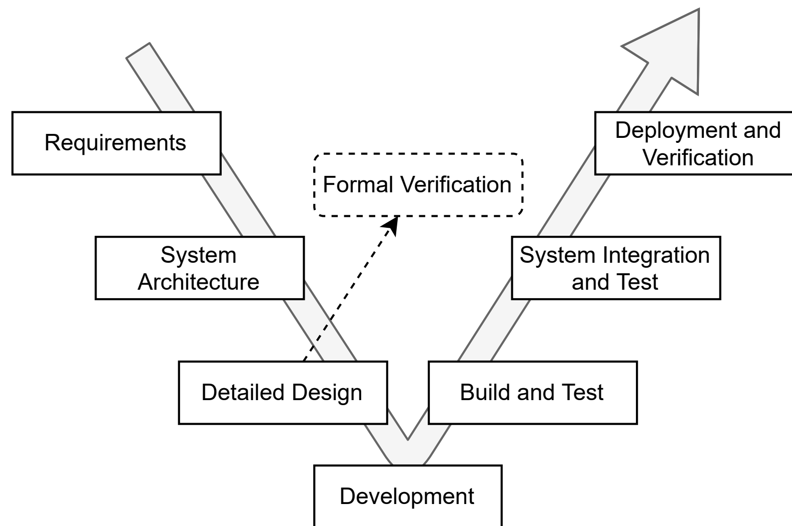

Figure 1. The V-Model can be easily modified to perform formal verification before engaging in development. The results of formal verification aid the detailed design of the system. This way some issues can be found earlier in the lifecycle, and development efforts can be conserved.

critical software or research in academic environments [Rus07].

1.2. Overview of a formalization process

In this section, we discuss a traditional approach to formalizing software. Here we assume that the formalization team has already collected the appropriate requirements and produced any relevant design documents.

The goal of the formalization process is to guarantee the behavior of the formalized system. Unlike with testing and simulation, the aim of formalization is to prove as many relevant properties of the system without actually writing a complete description of the related system. A complete description of a system would equate to the executable code of the system, which would negate the main advantages of using a formal method to abstract away the irrelevant details of a system's codebase. Subsequently, choosing the right level of abstraction is one of the first challenges when creating a formal system specification [AFP+11].

One of the first major steps of the formalization process is obtaining a specification of the related system [AFP+11]. A specification is an expression of a model of the system in a notation of a formal method. A specification contains the definition of the desired behavior of the system. If the specification is entirely abstract, it can be said that it is the description of the intended behavior as seen by an external observer. Otherwise, the specification is more operational and can include lower-level details about the system's behavior, which would only be known to an internal observer [AFP+11]. While an abstract specification requires less effort to produce, it might not be rigorous enough to provide insights into the design of the system. On the other hand, a very detailed specification might be tedious to produce and work with. As such, the right level of abstraction should be chosen.

Once the abstraction level of the specification is known, the model of the system can be produced [Pel19]. The purpose of this model is to be a transitional step between the requirements and the specification. This transitional model can be automatically generated from design documents, written down as an *English specification*, verbally discussed, or even just implied by the team performing the formalization. After the model with the desired abstraction level is established, a matching specification can be produced.

A specification defines the model using mathematical formalisms provided by the selected formal method's formal notation (syntax). While different formal methods can use considerably different ways of expressing the model, some concepts are common among them. Usually, the system is expressed as a state machine, and writing a specification can involve describing the following [AFP+11]:

1. The states or the data of the system;
2. The valid transitions between states of the system or predicates specifying when such transitions can occur;
3. The transformation of data when a transition between two states occurs or how the data evolves;
4. The predicates used inside the system.

When the specification is complete, it can be verified. A wide range of techniques can be used to verify a specification [Pel19], the main ones being model checking and formal proofs. In the case of model checking, some extra effort might be required to configure the parameters of the model checker, such as constraining the model so that the possible states of the system are not infinite and can be enumerated. Formal proofs usually require more work to verify, as a formal proof needs to be constructed. Many formal method tools offer computer-assisted and interactive ways to produce a proof and help check the proof's correctness [AFP+11; Pel19]. The incompleteness of the specification or other mistakes can also be found during this step, requiring a return to one of the previous steps before continuing.

Finally, a complete and verified specification can be used to guide the implementation. However, because of the stark differences between the specification and executable code of the system, it is often not trivial to obtain an implementation with the behavior of the formal specification. Different methods can offer some solutions to this *formal relation between specifications and implementations* problem [AFP+11]. Some formal method specifications can be directly executed, eliminating the problem altogether; others provide opportunities to derive an implementation from the specification and show the correctness of the derivation [AFP+11]. It is also possible to trust the judgment of the writers of the implementation to follow the specification closely, which is a flexible solution but leaves a lot of room for errors.

1.2.1. Translating requirements into formal specifications

The creation of a formal specification requires the translation of a system's requirements into a formal language. There is no standard requirements language: the system's requirements can be communicated verbally, presented as quick sketches "on a napkin", modeled with UML or BPMN diagrams, or otherwise stored in an ad-hoc design document [BKC+17; KKU13]. As a result, the requirements are usually written in natural language, which raises the issue of potential ambiguity, misunderstanding, or simply overload of information [IO05; KKU13].

There have been numerous attempts at tackling the problem of accurately translating requirements into a formal specification, such as an introduction of a strict requirements language with a controlled grammar [Sch02], or parsing of the natural language [IO05]. Despite that, there seems to be no industry standard. Furthermore, it is unclear whether the introduction of a transitional grammar or automatic translation would solve the issue at hand, as the translation would need to support the many types of formal notations as output to be useful.

While the automatic translation of requirements is usually not a viable choice, there are benefits to constructing a transitional model by hand. The transitional model can be a representation of the main expected properties of the model in a mathematical notation. If the formal method has already been decided on, the transitional model can attempt to express the system's properties in the logic of the formal method, such as classical logic. This model can also serve as a higher level, abstract, "as seen by an external observer" representation of the system's intended behavior, which can be useful to reason about the system or share it with a wider audience because of a mathematical notation's ubiquity. Moreover, the creation of such a model forces one to reason about the minutiae of the specified system so that any potential ambiguities can be addressed at an earlier stage[AFP+11]. Finally, the transitional model can serve as a guideline for the main specification.

When choosing a formal method that uses deductive techniques, extra attention should be paid to the class of logic that is being used. There are two main classes of deductive logic used in formal methods: classical logic and constructive logic. To put simply, classical logic is the standard logic that is most widely used; constructive logic is a subset of classical logic that focuses on proving by *constructing* a proof [Avi00]. In the context of formal methods for software, it should be noted that constructive logic comes with the upside of possible extraction of efficient code from the specification [Sas86]. However, constructive logic does not include the law of the excluded middle and double negation elimination, which are one of the central rules in classical logic [Avi00]. As a result, a formal method using constructive logic should be chosen if code extraction from the specification is desired, but otherwise, a formal method using classical logic might require less effort to work with.

1.3. Blockchains

Blockchain technology is the foundation of IOTA. To understand the motivations behind the extension of IOTA UTXO, this section describes the possibilities and limits of blockchain technology and relevant technical decisions made by IOTA that resulted in the current state of the

IOTA technology.

A blockchain is a mostly immutable ledger of blocks, which is implemented in a distributed fashion and usually without a central authority [YMR+19; ZXD+17]. Blockchains can be used in many fields, such as financial services, smart contracts, public services, security services, and, as claimed by IOTA, as a base for an Internet of Things service [Pop; ZXD+17]. The unique combinations of properties that a blockchain maintains and the technical challenges it poses have resulted in blockchain technology being in the spotlight of both researchers and entrepreneurs in recent years.

Serguei Popov of the IOTA Foundation claims that the blockchain technology was selected as the underlying base of IOTA to achieve a free and feeless Internet of Things infrastructure, where both value and responsibilities are shared between its users [PL19]. According to Popov, a distributed ledger technology can be a decentralized trusted backbone that allows its users to "trade data, manage access to them, and track responsibilities" to "billions of IoT devices and data" [PL19].

While nowadays there are many different blockchain solutions, we can still identify some core components that they have in common: transactions, a distributed ledger, a consensus mechanism, and asymmetric key cryptography [PMM+18]. Blockchain nodes communicate with each other by using transactions – descriptions of atomic actions that change the state. The current state of the blockchain is represented by these transactions [PMM+18]. The blockchain groups transactions into validated blocks. To regulate the addition of blocks and maintain a consistent state, a consensus mechanism is used. Besides exchanging information between different nodes of the system, a consensus mechanism provides security properties that help prevent fraud [PMM+18]. To provide a form of authentication, asymmetric keys are used – users sign their transactions with a private key which they keep secret, while their public key acts as a public address known to all participants of the network [PMM+18].

One of the most difficult challenges that a consensus mechanism has to solve when creating a distributed network is creating an asynchronous, Byzantine Fault-Tolerant system [FLP85]. In simple terms, this means that the network has to remain functional in an asynchronous environment where messages can get lost or delayed, all while some actors might be irrational or intentionally malicious. Solutions to this problem are often defined in the terms of the minimum amount of voting power that is required to break or exploit the system. The proposed solutions to this problem include approaches such as Proof-of-Work (PoW) and Proof-of-Stake (PoS). Both of these solutions are based on the idea of having a limited resource act as voting power in the system. In the case of PoW, processing power ("work") is the limited resource; in the case of PoS – it is the amount of currency ("stake").

One of the most famous examples of a PoW consensus is the Satoshi Nakamoto consensus model, which is used in Bitcoin. To prove that a large amount of work has been performed, a mathematical problem is formulated – a hash matching a very specific condition has to be found [Ren19]. To put it simply, for a new block to be added to the blockchain, a very large amount of guesses need to be performed until a satisfactory hash is found. Once a new block

is added, the mathematical problem is then formulated anew. The act of guessing the new hash is often referred to as *mining*, and the special type of entity that performs the mining is called a *miner*. The miner who first finds the new hash creates a new block and is rewarded with some amount of the currency for their work [Ren19].

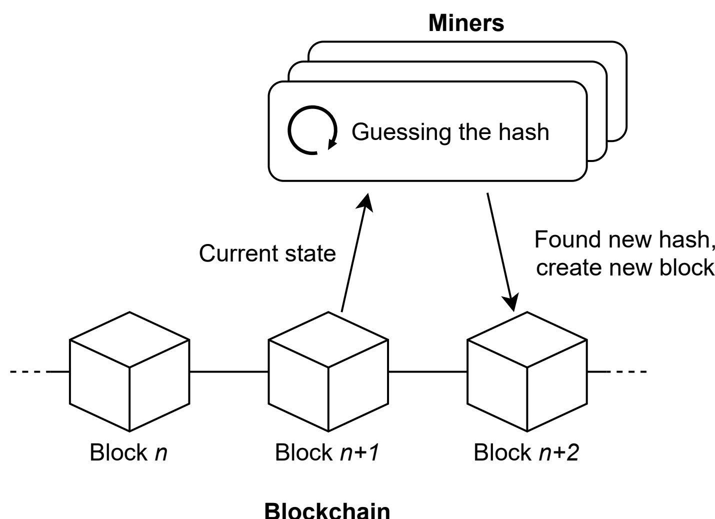

Figure 2. In a Nakamoto Proof-of-Work consensus model, the miners solve a difficult problem of finding a specific hash. Once a solution is found, the miner creates a new block for the blockchain. Upon receiving information about the new block, nodes can easily verify that the block is a correct extension of the blockchain and append it to their ledgers.

Since the blockchain network is distributed and asynchronous, any new information might propagate slowly. It is possible for two or more miners to find a new hash at around the same time. What happens, in this case, is that different partitions of nodes have a different state of the blockchain, because their last blocks have diverged. In a Nakamoto consensus model, this issue is resolved by one of these branches proving that it has the most combined computational power [Ren19]. Miners maintain information on all current branches but prioritize creating new blocks based on the first valid block that they have received. The idea here is that, once a new hash is found and a new block is created, one of the branches will unambiguously contain proof of containing the most amount of work – the nodes can then simply accept it as the source of truth [Ren19].

1.3.1. Account model

The account model is a simple and intuitive way to keep a ledger that contains an account identifier and the account's balance. The idea behind the account model is to keep a mapping of account identifiers to the related balance. A transaction in this model is an operation that simply reduces one account's balance while increasing another account's balance by the same amount [Woo+14]. Furthermore, a single atomic transaction can include any number of these reductions and increases, as long as the total balance of all accounts remains the same. A log of transactions can be

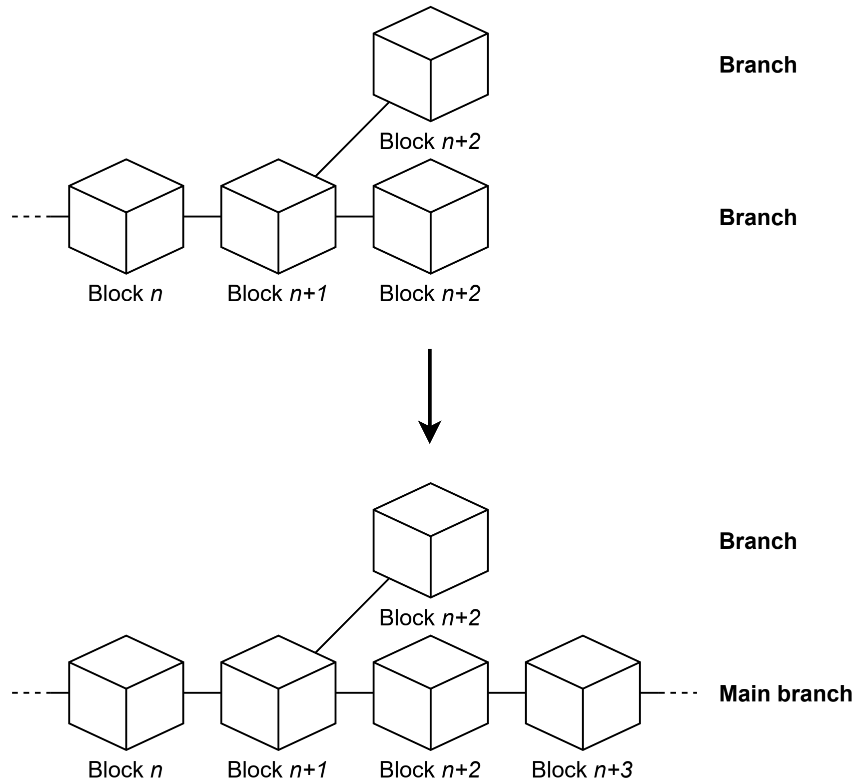

Figure 3. In a Nakamoto Proof-of-Work consensus model, when there is a potential conflict between two branches with the same amount of work performed (the same length), it is resolved by waiting for a new block to be created. Once one of the branches is longer than all others, it is considered to be the source of truth.

kept separately from the ledger and, to find out the current balance of an account, only a simple lookup is required.

While it is hard to contend with the account model's elegance in a centralized ledger, some issues become apparent if we try to apply the account model to a distributed ledger technology. The account model's ledger is, by design, a centralized resource. As such, it is important to keep the ledger's state updated when making a transaction. And, in the case of distributed ledger technologies such as blockchain, that can be a slow and expensive process [LFL19]. In general, the fact that transactions in the account model depend on the global state as input means that, usually, transactions involving the same account need to be processed sequentially.

1.3.2. UTXO model

The UTXO model is an alternative to the account model for keeping a ledger with account balances. Unlike the account model, the UTXO model does not store the total balance of an account. Instead, the ledger is a set of *Unspent Transaction Outputs*, or *UTXO*s for short [DPN+18; MDH+20]. A UTXO has a value amount and an owner. The transaction is then defined as a set of input UTXOs and output UTXOs, such that the total amount of value does not change. To

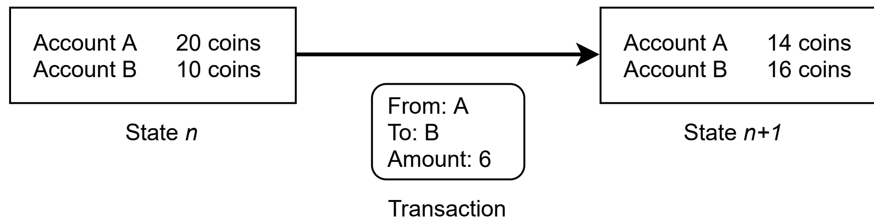

Figure 4. In the account model, the main components of the system state are the account identifiers and their respective balances. Note that the balances of accounts do not necessarily change with a state transition. In general, it can be assumed that the total value of all balances remains the same as transactions are applied.

make sure that such a transaction is valid, it is required to know that the input UTXOs have not been spent before – the total amount of value owned by the participants is of no concern.

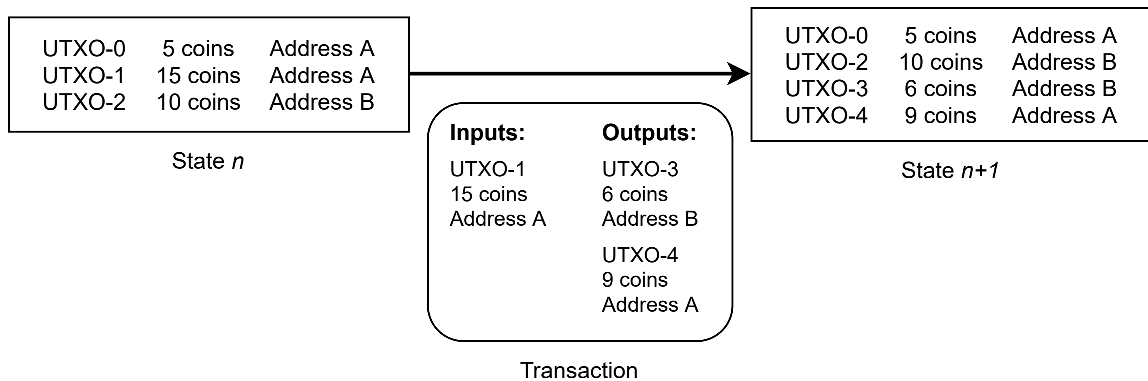

Figure 5. In the UTXO model, the current state is a set of all of the unspent transaction outputs. In other words, the UTXOs form a directed acyclic graph, and the current state is the set of all of the leaves of this graph. Note that this current set of UTXOs includes not only the UTXOs that were created during the latest state transition but also the ones from previous states which have not been spent.

While the UTXO model might seem confusing at first, we have a real-life analogy that can help better understand the motivation behind this ledger model. To put it simply, one can imagine an account model being akin to a bank balance, and a UTXO model as being physical cash. If we were to go to the market and receive some coins, we can spend them right away without having to put them into our bank beforehand. When a transaction occurs, we provide several coins (our input) and request several coins as change (output). Everyone in the market would observe the transactions and know that our coins are not fake – we were trading with coins that originated in the market this whole time. There is no need to consult a central authority, such as a bank, as long as we remain in this closed community. When we would go to a grocery

store and try to use the coins we received earlier, the store would then need to be aware of the market's records to confirm the coins as authentic.

More formally, the blockchain ledger's state in a UTXO model contains a set of UTXOs that have not been spent at some given moment. Transactions consume UTXOs as their inputs and produce UTXOs as their outputs, thus modifying the UTXO set. It is then enough to inspect only the current UTXO set to validate a transaction – no inspection of the full blockchain is required [DPN+18]. For a transaction to be valid, some conditions have to be satisfied. To avoid having to inspect the whole blockchain, UTXOs store all required information for validating a transaction. Besides the transferred amounts, they contain a locking script: each UTXO can have some individual conditions that allow it to be used as an input in the transaction, or, in other words, *unlocked*. These conditions are referred to as *unlock conditions* [DPN+18]. One of such conditions that is often included is the validation of the digital signature of the owner – it is obvious that only the owner of the UTXO can spend it. In this case, the proof – the digital signature – is part of the *redeemer* object provided in the transaction [CCM+20b; DPN+18].

The main benefit of using a UTXO model over the account model is that transactions in the former can be processed in parallel and independent transactions can be processed in any order [MPP+22]. This provides a positive effect on the system's scalability.

1.3.3. Smart contracts

In the UTXO model, the UTXOs can have a condition for spending a UTXO. Usually, this condition is simply the validation of the digital signature of the owner – one can use a UTXO if they can prove that they own it. However, there is a need for more programmable blockchain logic. The UTXO model supports some validation logic, but it's mostly limited to a set of basic operations [BG20].

In general terms, a smart contract is a protocol for verifying and enforcing contracts on a blockchain. A smart contract is stored on the distributed ledger technology, it inspects the state of the ledger, maintains and modifies its internal state, and performs actions such as creating new transactions in a blockchain. Storing a smart contract as a part of the blockchain comes with the benefits of a distributed and transparent execution, but also comes with some challenges, such as security concerns, required amounts of processing power, and the lack of regulation [BG20; WOY+19]. To address these problems of smart contracts, the solution of using a Layer 2 has been proposed [WOY+19]. The Layer 2 approach suggests the use of an off-the-chain execution environment for smart contracts. The blockchain is then only used as a consensus mechanism and so the execution of smart contracts does not impact the blockchain, providing a high level of performance and privacy [WOY+19].

In general, the steps for executing a smart contract are [MA19]:

1. Coding the contract – specifying the conditions and the outcomes;

2. Adding the smart contract to the blockchain;

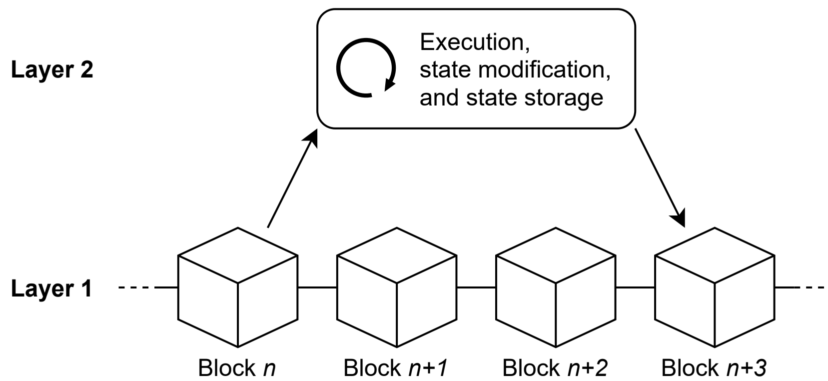

Figure 6. Layer 2 provides an execution environment that is separate from the main chain. It interacts with the main chain by reading and creating transactions. The main chain, which can also be referred to as Layer 1, acts as the consensus mechanism. The Layer 2 is the execution environment for logic based on the main chain's consensus mechanism. Layer 2 is often used to execute smart contracts or create other blockchains (sidechains).

3. When the predefined conditions of the contract are met, it executes and produces the outcomes.

Because of the complexities of implementing a smart contract, especially during step 1, the use of the account model is often preferred. As the UTXO model is stateless, a smart contract's transactions are forced to include any state information in the UTXOs themselves, further complicating the process and introducing overhead. Thus, a hybrid model which uses UTXOs for balances and accounts for smart contracts is sometimes used [BG20; ZTC+21].

There is a tradeoff between simple scripting systems and more sophisticated, Turing-complete ones. While a simpler scripting system might not allow for complex logic, it is easier to use when writing programs and verifying them. Feature-rich languages allow for more sophisticated smart contracts, but they are more error-prone and harder to secure [ZTC+21].

1.3.4. Native user-defined tokens

Native tokens are types of currency that are natively supported by the blockchain – every cryptocurrency has at least one main native token. However, there is a demand for blockchain technologies, such as IOTA, to support user-defined native tokens that can be used in user workflows or managed by smart contracts. While the EUTXO model defines a datum on the UTXO that can contain arbitrary information including user-defined tokens [CCM+20b], this approach has several downsides compared to native tokens. These downsides include extra complexity, processing cost, and inefficiency – the currency management code has to be replicated instead of reusing the existing currency management code of the blockchain's native token [CCM+20a].

A ledger that natively supports user-defined tokens would need to manage the construction

of these new user-defined tokens, the creation of some amounts of these tokens, and the transfer of ownership of user-defined tokens [CCM+20a].

1.3.5. Extended UTXO model

The *Extended Unspent Transaction Output* (*EUTXO*) model is an attempt to introduce expressive smart contracts while maintaining the semantic simplicity of the UTXO model [CCM+20b]. In short, the EUTXO model proposes changes to the UTXO model that allow a certain type of state machine to be implemented. This increases the expressiveness of the smart contracts while preserving the main properties of the UTXO model. To achieve this, arbitrary data is added to UTXO outputs. It has been shown that using this data together with changes to the validation logic is enough to enable state machine behavior [CCM+20b].

To allow the UTXO model to persist state between transactions, the EUTXO model proposes the addition of contract data to the UTXOs and a modification of the validation logic. Recall that in a regular UTXO model the UTXO consists of two core parts – the amount of value that is being transferred and the validator script [CCM+20b; DPN+18]. The EUTXO model proposes to extend this model with the addition of *datum* – arbitrary contract-specific data that the validator can inspect. This data can serve as the state of the contract. Another addition is the extension of the validator. Instead of validating only the value and the redeemer of the transaction, the value, redeemer, the newly added datum, and the whole transaction are inspected. Thus, the validator can impose arbitrary validity constraints on a transaction [CCM+20b]. It has been shown that this extended UTXO model enables the use of state machine behavior in a blockchain [CCM+20b].

1.4. IOTA EUTXO model

The EUTXO model provides the ability for arbitrary validation logic while maintaining the core UTXO workflow. While having arbitrary validation logic provides a great deal of flexibility, it can also result in arbitrary transaction validation execution times and arbitrary amounts of used validation processing power. Most blockchain solutions can solve this problem by providing monetary incentives (bounties) for processing a transaction requiring expensive validation or penalties to scripts that take a lot of CPU time. However, this ambiguity is an issue for a system that has no explicit incentives for processing a transaction. IOTA aims to have free transactions by requiring every node to perform validation to have its own transaction submitted, and as such, lacks the monetary tools to implement the complete EUTXO model supporting arbitrary state machine logic. As a result, IOTA chose to support flexible yet hard-coded scripts for output and transaction validation on Layer 1. While this still increases the complexity of validation logic, it remains bounded.

In essence, IOTA proposes a schema for the EUTXO datum and an extension to the validator to support this schema[1]. The schema describes new types of information recorded in the UTXOs, or, in other words, different subschemas intended for different workflows. For easier comprehension, these types of subschemas and their associated validation rules can be grouped

[1] https://github.com/lzpap/tips/blob/master/tips/TIP-0018/tip-0018.md

by their use case: UTXOs that have vastly different transition rules and are referred to as *output types*; unlocking conditions for these UTXOs called *unlock types*; finally, *feature types* are extra constraints and functionality not related to unlocking. Each output contains at most one unlock condition and feature of each type and not all unlock condition and feature types are supported for each output type. Additionally, to ensure that the state machine data is carried across transactions, the new *chain constraint* is introduced.

The new output types proposed by IOTA are:

1. Basic Output: a UTXO used for general transfer of funds with some attached metadata with optional spending restrictions. The main use cases are on-ledger smart contract invocation requests, native token transfers, and indexed data storage in the UTXO ledger with the use of the tag feature.

2. Alias Output: a UTXO representing smart contract invocation chain accounts on Layer 1 that can process requests and transfer funds.

3. Foundry Output: a UTXO that contains the state of and manages user-defined native tokens. Use cases include cross-chain asset transfers and asset wrapping (basically, representing some asset that belongs to one chain on another chain).

4. NFT Output: a UTXO that represents a non-fungible token (NFT) with attached metadata and proof of origin.

The new unlock types proposed by IOTA are:

1. Address Unlock Condition: an unlock condition that was previously widely used, generalized to fit into the *unlock type* abstraction. This unlock condition does not introduce new functionality. To satisfy this condition, in simple terms, the transaction has to be signed by the indicated address.

2. Storage Deposit Return Unlock Condition: an unlock condition that requires a certain transaction structure to be maintained. A UTXO with Storage Deposit Return Unlock Condition specified can only be consumed if the transaction deposits a certain amount of IOTA tokens into the specified address.

3. Timelock Unlock Condition: an unlock condition that requires the system to be in a certain state. A UTXO that contains a Timelock Unlock Condition can not be consumed before the specified timelock has expired.

4. Expiration Unlock Condition: an unlock condition that enables the optional transfer of assets. The sender specifies a time window during which the output can be consumed. Once the time expires, the sender can regain control of the assets.

5. State Controller Address Unlock Condition: an unlock condition used only by the Alias Output to control the state of its assets. It is almost equivalent to the Address Unlock

Condition. Note that the Alias Output not only has to be unlocked but the UTXO's state machine transition constraints have to be satisfied – the state has to be advanced.

6. Governor Address Unlock Condition: an unlock condition used only by the Alias Output to control the state of its governing, such as the owner's address. It is almost equivalent to the Address Unlock. Note that, even when changing a governing property, such as the owner address, constraints of the UTXO's state machine transition logic have to be satisfied.

7. Immutable Alias Address Unlock Condition: an unlock condition used by chain-constrained UTXOs. Similar to the Address Unlock Condition, but requires an Alias Output address.

The new feature types proposed by IOTA are:

1. Sender Feature: a feature that holds the address of the validated sender. A UTXO with this feature is validated to ensure that it is unlocked by the specified sender.

2. Issuer Feature: a feature analogous to the Sender Feature, but used by the Alias Output and NFT Output to specify the issuer (creator) address.

3. Metadata Feature: a feature used for storing arbitrary binary data. One use case is storing smart contract invocation request parameters.

4. Tag Feature: a feature used for providing an index for an output. The tags are expected to be used as an index when retrieving data.

For a UTXO to function as a state machine, the state of the UTXO must be moved forward when it is consumed as an input. In the UTXO model, the input UTXOs are essentially burned and only the value amount is distributed among the output UTXOs. Recall that in the EUTXO model, the UTXOs have to carry the state machine information inside their datum. Subsequently, IOTA proposes an extension to the validator called a *chain constraint*. To put it briefly, the chain constraint allows the transfer of the UTXO state machine state encoded in the datum across transactions. The Alias output, Foundry output, and NFT output utilize this chain constraint to transfer the states of the alias state machine, foundry state machine, and native token amounts respectively. When a UTXO with a chain constraint is consumed as an input, a respective UTXO with the subsequent state must be created as an output. The subsequent state is a single valid state transition of the state in the input UTXO, where the state transition rules are defined for each output type. Consequently, as each output with a chain constraint produces a successor, a path, or, in other words, a *chain* is created in the UTXO graph. Each such chain can be identified by a unique global identifier.

To conclude, the aim of these proposed changes is to have an affordable and scalable decentralized application platform, which supports native tokens and smart contracts. While the proposed changes do not enable the support of a generic state machine in the IOTA EUTXO model, they provide the needed framework to implement native token support and the most common smart contract conditions.

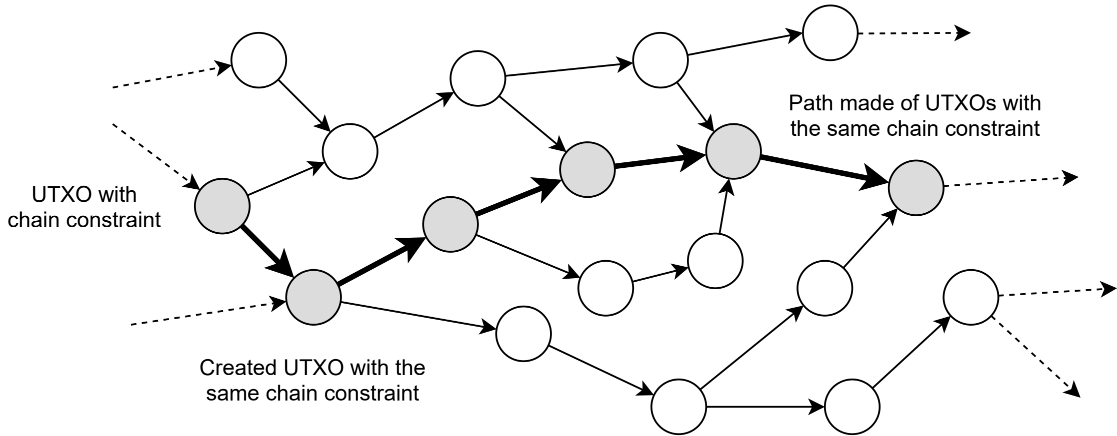

Figure 7. A path of consumed UTXOs with the same chain constraint forms a chain. In the figure, arrows indicate the creation of an output and the circles are the transactions; gray circles are transactions that contain a chain constraint. The chain starts when a UTXO with the chain constraint is created, and it ends when the UTXO with the chain constraint is destroyed; these actions are usually part of the UTXO's state transition logic.

1.5. State of the art

In this section, an overview of state-of-the-art software formalization methods in the context of blockchains is provided. Firstly, we provide an overview of how formal methods are used in the blockchain industry. Then we move on to discuss some specific examples of UTXO and EUTXO models being formalized. Finally, we discuss in greater detail an attempt to specify the Cardano UTXO model in Isabelle/HOL and provide our rationale on whether this work can be used as a base that can be extended with the proposed IOTA UTXO extensions.

1.5.1. Overview of the use of formal methods in the blockchain industry

Due to the immutable nature of a blockchain, it is vital that any bugs or errors are found as soon as possible, as otherwise they can become permanent once published to a live network. As such, there has been considerable interest in using formal verification in the distributed ledger technology industry, particularly in the verification of blockchain consensus mechanisms and smart contracts. In general, there are many components of a blockchain system that can benefit from formal verification, such as cryptographic algorithms, key management mechanisms, cryptocurrency wallet clients, and smart contracts [MA19]. In some cases, such as the analysis of Nakamoto consensus by Ling Ren [Ren19], only a formal analysis is performed without any verification; no automated proof tools are used. In other cases, such as the InnoChain distributed ledger system, the whole system stack – the underlying operating system, the consensus mechanism, the local node logic, the smart contract execution, and the smart contract language – is claimed to be formally verified [MRV20].

A smart contract is intended to be a flexible solution – a contract that can be programmed in a free form, allowing its authors to express the conditions of the contract with arbitrary complexity.

While the smart contract code is often available for inspection, manual review of the smart contract code is tedious and error-prone; the infamous DAO smart contract was written by experts in the field of blockchain, and yet it had a critical vulnerability [DMH17]. Subsequently, there have been numerous efforts to formally verify smart contracts.

One of the ways of formally verifying a smart contract is by automatically translating the contract code into a formally verifiable language. Bhargavan, Delignat-Lavaud, Fournet, Gollamudi, et al. have shown that the smart contracts written in Solidity can be partially translated to the F* language and verified automatically [BDF+16a]. After the translation, vulnerable patterns are detected in the code using the F* type checking system and other properties of the translated system can be verified. Due to syntax limitations, only 46 out of 396 smart contracts were successfully verified. In the end, this approach has only been shown to be suitable for simple smart contracts.

Amani, Bégel, Bortin, and Staples took a slightly different approach by creating a framework that allows verification of Ethereum smart contract bytecode in Isabelle/HOL [ABB+18]. In essence, they augmented an existing, thoroughly validated, formal Ethereum Virtual Machine (EVM) model in Isabelle/HOL; they then split the EVM code into so-called basic blocks. This allowed them to show that sound program logic proceeds from such blocks down to the level of instructions. They have also successfully applied their approach to a case study. However, the produced framework does not support all of the Solidity syntax, such as loops and message calls to other contracts. Consequently, the verification of bytecode has been shown to have a lot of promise but is only suitable for verifying simple smart contracts at the moment.

Another way of verifying smart contracts is through the application of model checking. Nehai, Piriou, and Daumas have demonstrated a way of applying model checking to an application based on smart contracts and have carried out a case study to illustrate the approach [NPD18]. In their study, the system was represented by three layers: the Ethereum blockchain, the application execution environment, and the smart contract. The model of the Ethereum blockchain was heavily simplified, and the contract code was translated from Solidity to the NuSMV symbolic model checker input language. The NuSMV input language is claimed to be suitable only for simple smart contracts, but it is not specified what set of rules is missing to capture the behavior of complex contracts. Overall, model checking has been shown to be sufficient for formally verifying simple smart contracts.

Finally, a specific smart contract can be formally verified by manually translating it to a formal method's notation. To avoid the *formal relation between specifications and implementations* problem, the formal specification can be written in a formal method supporting constructive logic, allowing the specification code to be extracted to smart contract code. Annenkov, Milo, Nielsen, and Spitters have shown that smart contract specifications written in the Coq formal verification language can be extracted to Liquidity and Midlang functional smart contract languages; the technique was applied to well-known examples of complex smart contracts, such as the DAO contract and the escrow contract [AMN+21]. The proposed method allows us to write programs in Coq, test them semi-automatically, verify, and then extract the Coq code to functional smart

contract languages. In total, while this approach requires some formalization skills and manual labor, it has been shown that writing specifications manually can be a flexible choice in cases when the contracts are relatively complex.

1.5.2. Attempts at formalizing the UTXO and EUTXO models

In blockchains based on the UTXO model, any native token or smart contract implementations, in the end, rely on the correct behavior of the base UTXO ledger. Due to the UTXO model's relative simplicity, there might be little value from formal verification. On the other hand, it is an important step, or even a prerequisite, to formalizing any modifications to the UTXO model, such as the Extended UTXO model or the IOTA UTXO extensions. During our research, we have observed more intention of formalizing the EUTXO model than the plain UTXO model. In this section, some of these attempts are presented and discussed.

Gabbay present an abstract, "idealized", high-level, but succinct mathematical formalization of the UTXO and EUTXO models. In their work, they build on the definitions provided by Chakravarty, Chapman, MacKenzie, Melkonian, Jones, and Wadler and focus on specifying the blockchain UTXO model as an algebraic structure. It is shown that blockchain segments, or the so-called chunks of the blockchain, form a partially ordered partial monoid [Gab22]. While some high-level properties are manually proven, the specifics of blockchains, such as tokens and monetary policies, are left out. However, the authors claim that the semantics introduced in the paper can be successfully used for concrete examples of UTXO-based blockchains. The paper provides an inspiring example of how mathematical analysis can be applied during the formalization process: the authors construct three categories – the *abstract chunk system*, the idealized UTXO, and the idealized EUTXO – and provide functors between them showing that they form a cycle of categorical embeddings. Many abstract algebraic constructs are used in the process, for instance, the abstract chunk system is defined as oriented atomic monoids of chunks that form a category with objects and arrows. All in all, while inspiring, this mathematical model can be difficult to comprehend without a deep prior knowledge of category theory. The model is also quite abstract, which means that some implementation details were dropped in favor of conciseness – while generic, this model might not be specific enough to offer insights into the properties of a real system.

Atzei, Bartoletti, Lande, and Zunino focus on specifying the Bitcoin transaction model in a mathematical notation as well as formally proving some specific properties of Bitcoin, such as no double spending and constant supply [ABL+18]. Unlike the formalization provided by Gabbay, this model focuses on providing an abstract model that does not require a high level of mathematical skill to understand and use – it is aimed at programmers. The definitions are more straightforward, for example, the blockchain is defined as a sequence of timestamped transactions, and the UTXOs are unspent transactions in the blockchain. No extra layers of abstraction, like blockchain chunks, are used. The data types are defined loosely and the properties of these types are defined and proved separately. The theorems are proven manually, no automatic tools seem to be used. The authors also present an open-source domain-specific language Balzac that can

be used to write Bitcoin transactions and translate them into standard Bitcoin transactions. To conclude, we believe that this paper is a good resource for understanding the UTXO model, even for researchers with a minimal background in mathematics.

The EUTXO model is formalized in Agda by Chakravarty, Chapman, MacKenzie, Melkonian, Jones, and Wadler. Unlike in the previously presented works, formal verification is performed in the Agda proof assistant tool which is based on constructive logic; numerous properties of the EUTXO are proved [CCM+20b]. It is shown that the proposed EUTXO model and Constraint Emitting Machines – a certain type of state machine that is suitable for execution on a blockchain ledger – have a weak bisimulation. To put simply, bisimulation means that one system effectively simulates the states and transitions of the other, and vice versa; *weak* in this case refers to the fact that the systems can still have some number of internal actions that are not visible to an external observer. Moreover, in a following work, the authors build on the Agda specification to prove that a ledger with custom tokens is strictly more expressive than the original EUTXO ledger [CCM+20a]. These works demonstrate the power of using a formal method with a proof assistant, as it allowed the authors to build on previous work and reuse the specification in future works: the code is available in a ubiquitous text format in an open source repository and can be easily imported; previous proofs are available and come "for free", as, using the proof assistant, it is usually relatively easy to check whether these previously defined proofs are still valid. In the end, the EUTXO formalization in Agda shows that formalization using a formal method with a proof assistant such as Agda promises to deliver continuous value in the research of the blockchain industry.

In the open source community, we have found one UTXO model formalization attempt without an accompanying paper. The Cardano ledger UTXO specification is an attempt to formalize the Cardano UTXO model in Isabelle/HOL by Javier Díaz. The primary motivation of the model seems to be, judging by its internal name *cardano-ledger-high-assurance*, to increase confidence in the Cardano UTXO model's correctness. The main properties of the system are proven, for example, the constant supply property and the no double spending property.

1.5.3. Cardano UTXO model specification in Isabelle/HOL

The attempt to formalize the Cardano UTXO model in Isabelle/HOL by Javier Díaz[2] is one of the works that is closest to the aims of this paper. As the plain UTXO model implementation usually does not significantly differ across different blockchain technologies, perhaps the results of the Cardano UTXO specification can be built upon and extended to include the IOTA UTXO extensions.

One of the main issues of effectively using this formalization is the lack of an accompanying paper – we were not able to find any related documents that would help shed light on some of the decisions taken when writing the specification. The specification seems to be heavily based on the finite map structure. There are more than 7 custom abbreviations and notations defined related to the finite map, which makes the reading of the following code cumbersome. Almost a

[2] https://github.com/input-output-hk/cardano-ledger-high-assurance/blob/master/Isabelle/UTxO/UTxO.thy

dozen helper lemmas are defined for the generic finite map, putting into question whether the use of the finite map was justified for the UTXO model. Because there is no accompanying paper, it is unclear whether there is a need for better understanding on our side or it is an actual concern.

In the specification, the transaction identifier, the address, and the transaction itself are defined as abstract types, for which the properties are then defined separately. For instance, it is defined that an abstract identifier is a type that is countable and has a linear order. It is not clear why an existing type, such as the natural number type, was not used. The current UTXO set data type is defined as a map from input UTXOs to output UTXOs. Three main properties of the UTXO model are proven – no double spending, the money supply is constant, and that applying a UTXO is the same as adding outputs and subtracting inputs. The proofs are quite long, with the proof for the money supply property spanning about 300 lines.

In the end, it is not evident to us whether the use of the finite map structure was justified. Perhaps some extensions to this model were envisioned which would benefit from the use of a map structure. Overall, we believe that it is beneficial to instead provide a new formalization of the UTXO model for use in future formalization efforts.

1.6. Overview of Isabelle/HOL

Isabelle/HOL is a tool used for formal proof and interactive theorem proving. It is built upon higher-order logic (HOL) and belongs to the Isabelle family of proof assistants [NPW02]. Isabelle/HOL allows for the development of machine-verifiable, human-readable formal proofs and specifications.

The main objective of Isabelle/HOL is to establish a mathematical foundation that is rigorous and capable of verifying systems, properties, and proofs. Isabelle/HOL assists in modeling complex systems and reasoning about their behavior, correctness, and security. Isabelle/HOL has applications in disciplines like mathematics, software engineering, and computer science.

Isabelle/HOL utilizes higher-order logic, which is a type of formal logic that extends first-order logic by providing the ability to quantify functions and predicates [NPW02]. Thus, it can represent complex mathematical structures and relationships. To implement higher-order logic, Isabelle/HOL uses simply-typed lambda calculus, providing a solid foundation for reasoning about types, functions, and predicates [NPW02].

1.6.1. Comparison with other proof assistants

There are several other proof assistants that can be considered as alternatives to Isabelle/HOL. Here we provide a short comparison of Isabelle/HOL with Coq, Agda, and Lean.

Coq: Coq is a proof assistant that combines higher-order logic with dependent types, called the Calculus of Inductive Constructions (CIC). Compared to Isabelle/HOL, which uses classical higher-order logic, Coq follows intuitionistic logic, resulting in a more constructive approach to proofs [BBC+97; Chl13]. While Coq's expressive type system allows for concise representation of complex mathematical structures, it can be challenging to learn and use [Tav21]. On the

other hand, Isabelle/HOL offers a more straightforward approach that allows users to use familiar mathematical notation and concepts.

Agda: Agda is a proof assistant and dependently-typed functional programming language [BDN09; Nor09]. Like Coq, Agda also relies on intuitionistic logic and has a powerful type system. However, it emphasizes the Curry-Howard correspondence, which connects proofs to programs, making it more programming-oriented. In contrast, Isabelle/HOL focuses on higher-order logic, making it more mathematically oriented, similar to traditional pen-and-paper proofs.

Lean: Lean is a proof assistant and functional programming language based on dependent type theory [dMKA+15; dMU21]. It supports both classical and intuitionistic logic, making it flexible for users depending on their preferences. Lean's syntax and features are designed to make it easier to automate, making it a promising tool for large formalization efforts [dMU21].

In summary, Isabelle/HOL differs from other proof assistants by using classical higher-order logic and providing a more accessible environment for mathematicians and researchers. In contrast, other proof assistants, such as Coq and Agda, focus on dependent types and intuitionistic logic.

1.6.2. Isabelle vs. Isabelle/HOL

Isabelle is a general proof assistant framework that can accommodate various logics, while Isabelle/HOL is a specific instance of Isabelle that uses higher-order logic for formalization and verification [NPW02; Vil21].

Isabelle is a tool that helps to prove theorems and verify formal systems. It can work with different formalisms and logics, so users can choose the one that best suits their needs [NPW02]. Isabelle/HOL is an instance of Isabelle that uses higher-order logic (HOL) as its underlying logic. Higher-order logic extends first-order logic by enabling quantification over functions and predicates, which makes it more powerful in capturing complex mathematical structures and relationships [NPW02]. Isabelle/HOL provides a user-friendly environment for formalizing and verifying systems, properties, and proofs within the context of higher-order logic by leveraging the Isabelle infrastructure [NPW02; Vil21].

1.6.3. Development environment and automated proofs

Isabelle/HOL provides an integrated development environment (IDE) called Isabelle/jEdit, which offers a user-friendly interface for writing and verifying proofs [NPW02]. It includes syntax highlighting, code completion, error checking, and various tools for proof development and management. This environment allows researchers to focus on their work while simplifying the proof construction process.

Automation plays a crucial role in Isabelle/HOL, as it helps users complete proofs with minimal manual effort. One notable automation tool in Isabelle/HOL is Sledgehammer, an automatic theorem prover (ATP) integration system. Sledgehammer connects external ATPs and satisfiability modulo theories (SMT) solvers with Isabelle/HOL, allowing for automatic proof discovery. Sledgehammer's integration allows users to focus on higher-level reasoning [Bla16].

However, it is important to note that the automatic search for proof can still be time-consuming, affecting the productivity of researchers as they may need to wait for results before continuing with their proof development. Choosing the right levels of abstraction and providing concise models can help mitigate this issue.

1.6.4. Isabelle/HOL syntax overview

Isabelle/HOL's syntax is designed to resemble traditional mathematical notation, making it easier for users to express and understand formalizations. Here, we provide a short overview of the main Isabelle/HOL keywords and constructs.

```
1  theory IotaUtxo
2    imports Main HOL.Finite_Set
3  begin
4  ...
5  end
```

In the above code block, the *theory* keyword is used to define a new theory named $IotaUtxo$. The imports keyword specifies the dependencies, in this case, $Main$ and $HOL.Finite_Set$. The scope of the construct is defined by the *begin* and *end* keywords.

```
1  datatype     addr = Addr nat
2  type_synonym iota = nat
```

Here, the *datatype* keyword is used to define a new data type *addr*, which is a simple wrapper around the *nat* (natural numbers) type. The *type_synonym* keyword defines a type alias, *iota*, which is an alias for nat.

```
1  datatype utxo = UTXO utxoID addr iota
2                | ALIAS utxoID addr iota aliasID
```

In this code block, the *datatype* keyword is used to define a new algebraic data type *utxo*, which has two constructors: $UTXO$ and $ALIAS$. Each constructor is followed by its respective arguments. The $UTXO$ constructor takes three arguments: $utxoID$, $addr$, and $iota$. The $ALIAS$ constructor takes four arguments: $utxoID$, $addr$, $iota$, and $aliasID$.

Constructor matching is a powerful technique for defining functions and proofs in a pattern-matching style. When defining a function or a proof, one can provide separate cases for each constructor of the data type. In the following *utxo_id* function definition, constructor matching is used to define the function's behavior for both $UTXO$ and $ALIAS$ constructors.

```
1  text ‹Simple getter.›
2  primrec utxo_id :: utxo ⇒ utxoID where
3  utxo_id (UTXO oid _ _) = oid |
4  utxo_id (ALIAS oid _ _ _) = oid
```

The *text* keyword is used to add comments in the theory file. In this case, it describes the purpose of the following function definition. The *primrec* keyword defines a primitive recursive function *utxo_id*, which takes an input of type *utxo* and returns a value of type $utxoID$.

```
1  lemma tx_input_not_in_outputs:
2    fixes tx :: tx
3      and i :: utxo
4    assumes a1: tx_valid tx
5      and a2: i ∈ tx_inp tx
6    shows i ∉ tx_out tx
7    using a1 a2 disjoint_iff tx.exhaust tx_inp.simps tx_out.simps tx_valid.
       ↪ simps
8    by metis
```

In this code block, the *lemma* keyword introduces a new lemma named *tx_input_not_in_outputs*. The *fixes* keyword declares two variables, tx of type tx and i of type $utxo$. The *assumes* keyword specifies two assumptions, $a1$ and $a2$, which are required for the lemma to hold. The *shows* keyword indicates the goal of the lemma, which is to prove that i is not an element of *tx_outtx*. The *using* keyword lists the facts used in the proof, and the *by* keyword indicates the proof method, in this case, *metis*.

1.6.5. Locales in Isabelle/HOL

A locale in Isabelle/HOL is a collection of parameters and assumptions that provide a context for proving theorems [Bal03]. To be more precise, locales are a way to define abstract contexts and structures, which can be instantiated later with specific types, functions, or relations. They provide a mechanism to reason about abstract properties and assumptions, allowing to prove theorems in a generic context and reuse the results in specific instances. Locales are used in the Isabelle/HOL libraries, such as the algebra library.

Mathematically, a locale can be expressed as follows [Bal03]:

$$\forall x_1, \ldots, x_n.\ [[A_1; \ldots; A_m] \implies C]$$

Here parameters x_1 to x_n are fixed, assumptions A_1 to A_m are made, and the conclusions C are implied. When writing Isabelle/HOL code, C would correspond to the proofs for lemmas and theorems that can be proven inside the context of the locale.

A locale can be instantiated by satisfying its parameters and assumptions using a specific type. Here is an example of a simple locale and its instantiation:

```
1  locale add_n_locale =
2      fixes n :: nat
3      assumes n_pos: n > 0
4  begin
5
6  lemma add_n_comm: x + n = n + x by ...
7
8  end
9
10 interpretation add_three_locale: add_n_locale 3 by ...
```

In this example, the *locale* keyword is used to define a new locale named *add_n_locale*. The $fixes$ keyword declares a constant n of type nat. The $assumes$ keyword specifies an assumption, *n_pos*, stating that n is greater than 0. The lemma *add_n_comm* is proved within the context of *add_n_locale*, showing the commutativity of addition with n. The $interpretation$ keyword is then used to instantiate the locale *add_n_locale* with a specific value, in this case, 3, creating an instance called *add_three_locale*.

The number 3 satisfies the assumptions of the *add_n_locale* because it is a natural number greater than 0, fulfilling the *n_pos* assumption. Using the *add_three_locale* instance, we can use the lemmas of the parent locale (*add_n_locale*) instantiated with the specific value of 3. For example, we can use the *add_n_comm* lemma from the parent locale to prove the commutativity of addition with 3, i.e., $x + 3 = 3 + x$. This allows for the reuse of the proven properties of the parent locale, making it easier to work with similar concepts and properties without the need to reprove the same lemmas for different values.

We can further reuse locales by subtyping with $+$ or *sublocale* commands [Bal03]. Consider the following continuation of the previous example:

```
1  locale add_n_m_locale = add_n_locale +
2      fixes m :: nat
3      assumes m_pos: m > 0
4  begin
5
6  lemma add_n_m_comm: x + n + m = m + n + x by ...
7
8  end
```

In this example, the *add_n_m_locale* extends the *add_n_locale* using the $+$ symbol. This creates a new locale that includes all the elements of the *add_n_locale* and adds additional elements, such as the constant m and the assumption *m_pos*. By extending the *add_n_locale*, the *add_n_m_locale* automatically inherits all the lemmas from its parent locale and can define new lemmas that depend on both n and m.

The *sublocale* keyword is used to establish a relationship between the locales by explicitly stating that *add_n_m_locale* is a subtype of *add_n_locale*:

```
1  sublocale add_n_m_locale ⊆ add_n_locale
2    using n_pos by ...
```

Using the *sublocale* command, we can prove that an instance of the *add_n_m_locale* also satisfies the assumptions of the *add_n_locale*. In this case, we prove the *n_pos* assumption from the parent locale to show that the sublocale relationship holds.

1.7. Conclusions

Formal methods offer a way to formally verify the properties of a system before any executable code has been written. While specifying a system in a formal method's notation requires a degree

of skill and knowledge, the advantages include confidence in the design, insights into the system, and the detection of possible defects early in the software lifecycle. Nowadays there are a plethora of tools that assist the formalization and verification of a system, be it by model checking or proof-assisted theorem proving.

In the blockchain industry, having native tokens and smart contract support is in high demand. IOTA proposes a collection of extensions to its UTXO model to support these workflows. While the extended UTXO model proposed by IOTA does not enable the use of a generic state machine, the introduced output types, unlock conditions, features, and constraints are generic building blocks that can be used to create native tokens and program common smart contracts on the blockchain. As with other blockchains, any defects or errors must be found as early as possible, as they might become permanent once published to a live network. Because the IOTA blockchain intends to be a critical backbone for other applications and devices, formally verifying the proposed extensions is worthwhile.

There have been numerous attempts to formalize both the UTXO and EUTXO models. The formalizations vary in the level of abstraction, intended audience, the complexity of notation, and the goal of formalization. We found significant evidence of extensive analysis of the generic UTXO and EUTXO models, but most of such works did not focus on the verification of the models, used abstract constructs, or did not offer a specification that can be used with a proof assistant or another tool. We have also observed some attempts to specify the UTXO models, but they either did not have an accompanying paper with explanations and rationale or were using constructive proofs, which would complicate the proving of properties of the extended specification. Subsequently, no suitable base for the IOTA UTXO extension formalization was found. In the end, we believe that it is beneficial to create a formal specification of the UTXO model using the notation of a proof assistant based on classical logic, such as Isabelle/HOL. This specification can be partially based on the discussed works, but would still be different enough and would need to be written from scratch. This specification can then be used as a base and updated with the UTXO extensions proposed by IOTA and, finally, formally verified.

2. Modeling UTXO in Isabelle/HOL

2.1. Overview of the UTXO model

In this section, we provide an overview of the UTXO model, focusing on the core components, including types such as Output, Input, Transaction, and Ledger.

The models in this section assume that there is a single node and subsequently are not concerned with the consensus mechanism that would exist in a real network. Thus, all of the actions in such a node, such as application of transactions, are performed sequentially.

The details in the following subsections are generally applicable to many different implementations. However, it is important to note that specific implementations, such as IOTA, may refine or extend some of these rules due to their specific requirements or design choices. Readers should be aware that certain aspects may differ in these individual implementations.

2.1.1. Essential entities of the UTXO model

Output. An output represents a portion of tokens that can be spent by an entity:

$$\text{Output}: \begin{cases} id : UniqueId \\ unlockConditions : UnlockConditions \\ amount : \ \mathbb{N} \end{cases}$$

In the UTXO model, outputs are created as a result of transactions. Each output has a unique identifier, ensuring that it can be distinctly referenced in future transactions. The unlock conditions determine the conditions under which the tokens can be spent, such as the owner entity's address who has the authority to spend the tokens held in the output. The amount field is equal to the number of tokens stored in the output.

There can be several types of unlock conditions which require specific types of data to be present in the output – a reference unlock condition requires an unlock block containing the index of the previous unlock block, while a signature unlock conditions uses an unlock block with the hash of the appropriate public key. Here we abstract away the data and logic requirements of unlock conditions under $UnlockConditions : Output \to \mathbb{B}$ – a predicate which returns *true* iff the unlock block conditions are satisfied.

Transaction. A transaction contains a list of inputs, outputs, and unlock blocks:

$$Transaction : \begin{cases} id : UniqueId \\ inputs : Set(Output) \\ outputs : Set(Output) \\ unlockBlocks : Set(UnlockBlock) \end{cases}$$

Inputs represent unspent outputs that are being consumed in the transaction, while outputs rep-

resent outputs that are expected to be created. Unlock blocks provide the necessary signatures to authorize the spending of inputs. All of the elements in these collections are unique and do not depend on any ordering, thus they can be represented by a set of the relevant type. Each transaction also has a unique identifier to ensure it is traceable within the ledger.

Ledger. The ledger, which we define as a UTXO set, is a set of outputs representing unspent outputs. This set contains outputs that have not yet been consumed by any transaction:

$$Ledger : Set(Output)$$

We define a function $ApplyTransaction$ that computes the next ledger state from the current state and a transaction. This function updates the ledger by removing consumed outputs from the UTXO set and adding newly created outputs:

$$ApplyTransaction : Ledger \rightarrow Transaction \rightarrow Ledger$$

The presented model operates sequentially, simulating a ledger within a single node. As a result, it abstracts away aspects such as consensus. Thus, behaviors such as consensus are not in the scope of this framework.

2.1.2. Essential properties of the UTXO model

In this section we provide an overview of the essential properties of the UTXO model. These properties serve as a basis to derive other properties of the model.

1. **Constant Supply:** The sum of unspent outputs in the ledger must be constant. This ensures that tokens are neither created nor destroyed in any transaction:

   $$\Box \sum_{output \in Ledger} output.amount = TotalSupply \tag{1}$$

   Here $TotalSupply$ is the constant total supply of tokens in the system. The temporal operator $\Box$ indicates the fact that the formula holds throughout the $Ledger$'s lifecycle.

   It is worth noting that some UTXO model implementations require mechanisms to change the total supply of tokens in the system. Inflation or deflation in the UTXO model can be achieved by using specially designated owner entities. The entities can either release tokens that were unavailable earlier or accept tokens never to release them back in circulation. Even so, some implementations choose to allow special kinds of transactions that directly violate the constant supply property by creating (minting) new tokens or destroying (burning) tokens in the transaction.

2. **Unspent Output Consumption:** An output can be consumed by a transaction only if it is a part of the current ledger state. This ensures that transactions cannot spend outputs that are not in the ledger:

$$\forall t \in Transaction.\ \forall o \in t.inputs \Rightarrow\ o \in Ledger \tag{2}$$

Here $Transaction$ is a valid transaction that is to be applied to the ledger. For a transaction to be valid all of the outputs that are consumed in the transaction have to be inside of the ledger.

3. **No Double Spending:** An output can only be consumed by a single transaction.

   The UTXO model is designed to prevent double spending, which is a critical security concern in digital currency systems. Double spending occurs when a single output is consumed more than once. In the UTXO model, this is prevented by ensuring that an output can only be consumed by a single transaction.

$$\forall o \in Output.\ \exists! t \in Transaction : o \in t.inputs \tag{3}$$

Here $Output$ refers to the set of outputs that were at some point added to the $Ledger$ and $Transaction$ is a set of transactions that were previously applied to the $Ledger$. The formula states that for each output, there exists exactly a single unique transaction that consumed it as an input.

From this definition, we can also derive a different expression of this property. Assume that there are two transactions, t_1 and t_2 that have a common input o:

$$o \in t_1.inputs \cap t_2.inputs \tag{4}$$

From (3) we know that for each output o, there exists exactly one unique transaction that consumes it as an input. Let us call the unique transaction t_u:

$$o \in t_u.inputs \tag{5}$$

From (4) and (5) we can conclude that t_1 and t_2 are both equal to t_u:

$$t_1 = t_u, t_2 = t_u \Rightarrow t_1 = t_2 \tag{6}$$

Thus, we have deduced an alternative representation of the property:

$$\forall t_1, t_2 \in Transaction.\ \forall o \in t_1.inputs \cap t_2.inputs \Rightarrow\ t_1 = t_2 \tag{7}$$

This formula states that for any two transactions t_1 and t_2, if they have a common input o, then the transactions must be identical. In other words, an output can only be used as an input for a single transaction, which prevents double-spending. The expression in (7)

can be useful when proving properties related to multiple transactions, such as creating an equivalence relation.

4. **Signature Verification:** A transaction must have valid unlock blocks (signatures) for all inputs. This ensures that only authorized parties can spend the corresponding outputs:

$$\forall t \in Transaction.VerifySignatures(t.inputs, t.unlockBlocks) \tag{8}$$

Here $Transaction$ is a set of transactions that are to be applied to the $Ledger$. $VerifySignatures$ is a function that checks the validity of signatures for a given set of inputs and unlock blocks and ensures that all signatures are valid.

5. **Conservation of Value:** The sum of input amounts must equal the sum of output amounts for each transaction. This ensures that the total value of inputs is redistributed among the outputs, preserving the total supply of tokens:

$$\forall t \in Transaction. \sum_{i \in t.inputs} i.amount = \sum_{o \in t.outputs} o.amount \tag{9}$$

Here $Transaction$ is a set of valid transactions.

6. **Progress:** The UTXO model should always allow for the possibility of creating new transactions. This ensures that the system remains operational and that tokens can be transferred between parties:

$$\forall l \in Ledger, \exists t \in Transaction.\ IsValidTransaction(l, t) \tag{10}$$

$IsValidTransaction$ is a function that checks whether a given transaction is valid for the current ledger state. The progress property asserts that, for any given ledger state, there exists at least one valid transaction that can be applied. This allows the system to progress and be functional. In simpler words, it can be said that the system should never get stuck.

2.2. Hashing

Hashing is the application of a hash function to some data. In general terms, a hash function is a function that maps input data of arbitrary size to an output of fixed size [Con+19; CZ17]. Moreover, the most commonly used hash algorithms, such as Secure Hash Algorithm-256 (SHA-256), have the following properties [Con+19]:

- Hashing the same data always produces the same output.
- Hashes of two very similar, but different, data sets can be very different.
- While two different data sets can theoretically have the same hash, it is extremely unlikely. In practice, it is often assumed that no collisions can occur and a hash is unique.

In a blockchain ledger, the blocks are chained together using an approach of recursive hashing [Con+19; CZ17]. In this approach, a block's hash is calculated using both its content and metadata, which incorporates the hash of the preceding block. Consequently, the hash not only identifies the block but also connects it to previous blocks. With recursive hashing, even if two blocks have identical content, their respective hashes will differ due to the inclusion of distinct previous block hashes in their content.

In the UTXO model, every UTXO has a hash that is assumed to be unique. The hash is generated when a UTXO is added to the ledger. This hash acts as the UTXO's identifier [Zah18a].

In this formalization of the UTXO ledger and its extensions, we abstract from the hashing algorithms and only rely on the properties the hashes provide. The main use of the hashes in this model is to identify various objects, e.g. transactions, transaction outputs, aliases, and others. With this in mind, we consider them only as identifiers. Assuming the collisions of the hashes are impossible, these identifiers are unique. The following subsections describe various ways of modeling the UTXO model's unique identifiers.

2.2.1. Type of a unique identifier

There are two main ways to approach modeling the types of unique hashes which act as UTXO identifiers:

1. Build on existing types. A natural candidate for modeling a unique identifier is *Nat* – the type of natural numbers.

2. Defining a new type, e.g. *Hash*, and assume the needed properties for it.

The first approach is simpler to intuitively grasp. If we assume that the identifier (ID) is a natural number, then it is trivial to prove that a new, unique value always exists – we can simply take the successor of the largest natural number that has been used up to this point.

```
1 type_synonym  utxo_id = nat
```

On the other hand, using Nat as an identifier is not an accurate abstraction. The type of natural numbers carries with itself the many properties and operations of natural numbers (e.g. addition, ordering) which are not relevant in the modeling of unique identifiers. This means that this approach not only does not accurately express the intent of the model but that the irrelevant properties might be accidentally used in future proofs.

Another consequence of using Nat is that it might also result in a negatively impacted speed of automated theorem proving in Isabelle.

The second approach to modeling the type of a unique identifier is defining a new type. Let us define a new type *Hash* and assume that it is infinite. Assuming that only a finite number of hashes were used before, we can always acquire a new Hash that we have not yet encountered.

```
1 typedecl hash
2 type_synonym  utxo_id = hash
```

While both of these approaches are sufficient for modeling a type of unique hash, we prefer the latter approach. We believe that defining a new type is a more accurate representation of the model, as it explicitly defines the properties required by the model and is does not risk accidentally relying on any extraneous properties of the Nat type.

2.2.2. Identifier uniqueness

In this model, we abstracted away the hashes to arbitrary identifiers. We don't derive them from the actual data of an object, thus we have to guarantee their uniqueness explicitly. The general approach taken in this model is to always take an identifier that is not used by any object in the model.

We observed two primary approaches to modeling the uniqueness of an identifier in Isabelle. The difference lies in how the collection of used identifiers is acquired:

1. The set of already used identifiers can be maintained explicitly as a part of the defined model. As such a construct would not exist in a real system, it can be considered to be an artifact of the modeling process.

2. The set of already used identifiers can be modeled as a function of the model state. Such a function would return a set of all identifiers used in the model state.

The main difference between these approaches is that by tracking the set of already used identifiers as an explicit set we would consider all the previously used identifiers as still in use, even if they are not part of the current data in the model. In the latter approach, the function would return only the identifiers currently in use. Such a set can decrease if objects containing some identifiers are removed from the model.

In our abstract model, we represent the set of used hashes as a finite subset of identifiers, which includes the already used ones. We can express this relationship as:

$$\begin{aligned} &usedHashes : FiniteSet(Hash) \\ &usedInLedger \subseteq usedHashes \end{aligned} \tag{11}$$

In (11) $usedHashes$ is a finite set of hash values, and $usedInLedger$ denotes the set of hashes used in the ledger. The second line of the equation shows that $usedInLedger$ is a subset of $usedHashes$, which means that all hashes used in the ledger are part of the finite set of used hashes.

To avoid having to explicitly specify the set of used identifiers as an explicit parameter to definitions inside the model, the set of used identifiers and their uniqueness can be considered as an implicit part of the model.

We considered two main ways to provide implicit assumptions to a certain scope of Isabelle expressions:

1. Defining the assumptions in a type class.

2. Defining the assumptions as a part of a locale.

2.2.3. Modeling identifier uniqueness using a type class

Using a type class, we can define an assumption for a data structure that we want to add inside the class. The relevant data types can then become instances of this type class, ensuring that the assumption is always present. When working with identifiers, we can rely on the fact that the data types have the type class, which in turn guarantees that the corresponding assumptions are present and satisfied.

```
1 class hashed =
2   fixes used_hashes :: 'a ⇒ hash set
```

In this code block, we define a type class called *hashed* which requires a function *used_hashes* to be implemented for any type $'a$ that becomes an instance of this class. The function *used_hashes* takes an instance of type $'a$ and returns a set of hashes.

As an example, let's define a concrete data type $MyData$ with two hash parameters to demonstrate how structured data can be mapped to a set of hashes:

```
1 datatype MyData = MyData (MyData_h1 : hash) (MyData_h2 : hash)
```

Now, we will show how $MyData$ can be an instance of the hashed class. To do this, we need to satisfy the assumptions of the *used_hashes* function for the MyData type by defining *used_hashes_MyData*.

```
1 instantiation MyData :: hashed
2 begin
3 primrec used_hashes_MyData :: MyData ⇒ hash set
4     where used_hashes_MyData (MyData h1 h2) = {h1, h2}
5 instance ...
6 end
```

In this code block, we use the *instantiation* keyword to show how the new data type $MyData$ implements the *hashed* type class. By convention, the function required by the type class is named $\{FunctionName\}$_$\{DataType\}$ with the corresponding parameters. In this case, we define *used_hashes_MyData* as a function that takes an instance of $MyData$ and returns a set containing both hashes $h1$ and $h2$. In general, such a function could collect hashes recursively. The *instance* keyword is used to prove that $MyData$ implements the *hashed* type class.

We can now define a function that accepts a parameter of type *hashed* instead of $MyData$ or any other specific type.

First, let's define a function called *get_unused_hash* that takes a parameter of type $'a$ :: *hashed* and returns a hash:

```
1 definition get_unused_hash :: ('a :: hashed) ⇒ hash
2     where get_unused_hash x = (SOME h. h ∉ used_hashes x)
```

This function definition uses Isabelle's $SOME$ keyword to find a hash h that is not in the set of used hashes for the given hashed object x. The type signature ($'a$ :: *hashed*) means that the function takes a parameter of type $'a$ that is an instance of the hashed class.

We can apply the get_unused_hash function to an instance of $MyData$:

```
1  value get_unused_hash (MyData h1 h2)
```

The function will return a hash that is not in the set of used hashes $h1, h2$ for the given $MyData$ object. To summarize, we have shown how to define and use a function that accepts a parameter of type $hashed$ and how it can be applied to a specific data type that implements the $hashed$ class, such as $MyData$.

2.2.4. Modeling identifier uniqueness using a locale

The essence of using a locale to define these implicit assumptions is similar to using the type class. We define the required assumptions, and they are available inside the locale without having to include them as explicit parameters in definitions. Later, we can show that a particular data type satisfies the locale (its assumptions) by providing an interpretation – demonstrating that the concrete type satisfies the assumptions defined in the locale. For a more detailed overview of locales in Isabelle/HOL, see Section 1.6.5 Locales in Isabelle/HOL.

As an example, let us define a locale $hashes$ with the assumption that it contains an infinite set of hashes (H) and a finite one (used hashes, U).

```
1  locale hashes =
2    fixes H :: hash set
3      and U :: hash set
4  assumes H_inf: infinite H
5      and U_fin: finite U
6  begin
7
8  ...
9
10 end
```

Here, the locale fixes two variables H and U, both being sets of hashes. The set H stands for a set of all possible hashes, and U stands for the set of hashes already used. We define the assumption for the set of hashes to be infinite by H_inf and for the set of used hashes to be finite by U_fin.

Note how the locale's assumption U_fin serves as an alternative to the type class assumption $used_hashes$.

In the locale, we can define $take_hash$:

```
1  locale hashes =
2  ...
3  begin
4
5  definition take_hash :: hash set ⇒ hash
6  where take_hash excl ≡ (SOME h :: hash. h ∈ H ∧ h ∉ U ∧ h ∉ excl)
7
8  end
```

The $take_hash$ function is designed to find and return an unused hash that is also not in the exclusion set ($excl$). The exclusion set allows us to specify additional hashes that should be avoided, even if they are not yet part of the used hashes set (U).

Now let us provide an example of how the locale can be implemented using interpretation and demonstrate the usage of the $take_hash$ function. Suppose we have a concrete data type that satisfies the assumptions of the $hashes$ locale. We can then use interpretation to show that this data type is an instance of the locale. In our case, this is done by providing the required infinite set of all possible hashes and the finite set of used hashes.

```
1 interpretation my_hashes: hashes all_hashes used_hashes
2 by ...
```

We can then use the $take_hash$ function to find an unused hash for our specific data type. For example, we can retrieve two unique hashes at the same time:

```
1 definition find_two_unused_hashes :: (hash * hash)
2     where find_two_unused_hashes ≡
3     let
4         (first_hash = my_hashes.take_hash {};
5         second_hash = my_hashes.take_hash {first_hash})
6     in
7         (first_hash, second_hash)
```

In this definition, the $find_two_unused_hashes$ function returns a pair of unique hashes that are not part of the used hashes set (U) and are guaranteed not to intersect with each other. We first call the $take_hash$ function with an empty exclusion set to obtain the first hash. Then, for the second call, we pass an exclusion set containing only the first hash to ensure that the second hash is different from the first one. This way, we can guarantee that the two hashes are unique and not part of the used hashes (U).

We have observed that the Isabelle automatic proof system required fewer steps to prove theorems defined using the locale approach than identical theorems written using type classes. The authors of Isabelle also prefer using locales instead of type classes, which is apparent in the Isabelle/HOL abstract algebra library source code.

2.3. Subtyping

2.3.1. Producing a modular and extendable model

The extended ledger we model involves different types of outputs. Specific types of outputs convey additional restrictions for the ledger and allow the enforcement of the desired behavior. Examples of such specific output types are Alias and Foundry outputs. The former is introduced to simulate a sub-chain in a ledger, and the latter is used to implement a specific token with defined supply rules. We assume such specific outputs conform to all the rules for the regular UTXO ledger outputs. E.g., apart from the additional features, they have to convey a non-zero amount of tokens, etc. Thus considering type as a set we have:

$$AliasUTXO_{\text{set}} \subseteq UTXO_{\text{set}}, \quad FoundryUTXO_{\text{set}} \subseteq UTXO_{\text{set}},$$
$$AliasUTXO_{\text{set}} \cap FoundryUTXO_{\text{set}} = \emptyset \quad (12)$$

We build our model based on the Higher Order Logic (HOL). The set of all UTXOs $UTXO_{\text{set}}$ is modeled as a type $UTXO$. Then we can define functions over these types, e.g., to get the amount of a particular UTXO, we define a function $amount : UTXO \to \mathbb{N}$.

Subsets of values of a particular type can be considered subtypes in general. However, HOL doesn't support subtyping directly.

The UTXO model can be defined using several different Isabelle/HOL constructs. But, besides the technical details, when producing a UTXO model in Isabelle/HOL, we aimed to make it easy to interpret and extend by a human. Subsequently, the following subjective metrics were considered:

1. Separation of model properties and implementation specifics. This separation would ensure a clean and modular design, making it easier to understand, maintain, and extend the model.
2. Extensibility. An extensible design will allow us to easily build upon the base UTXO model to incorporate additional features without altering the core definitions.

To address both the technical challenges and subjective concerns mentioned earlier, we considered the alternatives for defining the UTXO model in Isabelle/HOL that are defined further in this section.

2.3.2. Modeling subtypes as disjoint subsets of different types

This approach defines functions to map subtype values to general types and functions for each subtype independently.

For example, consider two subtypes, A and B, which are part of a general type, T. Functions are created to map A and B to T, such as $a_to_t : A \to T$ and $b_to_t : B \to T$, and handle operations on A and B separately. In our model, we consider objects can be of a single subclass only, meaning they cannot be of type A and B simultaneously. Thus, the subsets are disjoint. We can express this behavior as:

$$\forall a \in A, b \in B . a_to_t(a) \neq b_to_t(b) \quad (13)$$

In the context of the UTXO model, this approach could be applied to manage different output types (e.g., Alias and Foundry outputs) and their corresponding operations.

However, while offering modularity, managing multiple mappings can increase complexity. Maintaining the relationship between subtypes and the general UTXO type might require a complex mapping system, which increases the effort required to maintain and extend it.

2.3.3. Modeling functions on subtypes as relations

In this approach, functions on subtypes are modeled as relations. Instead of explicitly defining a function that maps elements of a subtype to their corresponding results, we define a relation that pairs these elements with their results.

For instance, given a subtype A and a function R, we define a relation R_f that associates elements of A with their corresponding results.

$$R_f \subseteq A \times B \tag{14}$$

In the UTXO model, this approach could be applied to model the relationship between transactions and their corresponding outputs. Let T be the set of transactions and O be the set of outputs. We can define a relation R_{tx_out} that associates each transaction with its corresponding outputs:

$$R_{tx_out} \subseteq Transaction \times \mathcal{P}(Output) \tag{15}$$

For a transaction $t \in Transaction$ and a set of outputs $O_t \in \mathcal{P}(Output)$, we have $(t, O_t) \in R_{\text{tx_out}}$ if the outputs in O_t are created by the transaction t. This relation shows the link between transactions and their outputs in the UTXO model.

However, maintaining the consistency of these relations might require a lot of effort. For example, it might be necessary to introduce extra checks and constraints to ensure that each element in the domain is related to exactly one element in the codomain.

2.3.4. Using quotient types

Quotient types partition a type into equivalence classes and modeling subtypes. It is a method for organizing elements into distinct groups based on their shared properties or relations.

Suppose we have a type T and an equivalence relation $\sim$. We can create a quotient type $T/\!\sim$ consisting of equivalence classes, effectively representing subtypes.

$$T/\sim = \{[t]_\sim \mid t \in T\} \tag{16}$$

Here a new group $T/\sim$ is created by taking all the elements from the original type T and placing them into equivalence classes $[t] \sim$ based on the equivalence relation $\sim$. Each of those equivalence classes represents a subtype, and the elements within the equivalence class share the properties defined by the equivalence relation.

In the UTXO model, this approach could be used to group different output types based on specific properties. For instance, we could define an equivalence relation that groups outputs according to their restrictions. Overall, this approach may simplify the handling of specific output types but might require extra effort to define the equivalence relations.

2.3.5. Using locales

Locales are an Isabelle/HOL construct that provide a way to model subtypes.

Locales allow the separation of essential model properties from the specifics of an implementation. A locale is a collection of parameters and assumptions that provide a context for proving theorems. Locales also offer ways to easily extend one another, such as inheriting one locale into another locale using the + operator or proving that one locale is an extension of another locale using the *sublocale* command [NPW02].

$$\forall x_1, \ldots, x_n. \ [[A_1; \ldots; A_m] \implies Theorems] \tag{17}$$

This formula states that for all variables $x_1 ... x_n$, in the context of assumptions $A_1 ... A_m$, we can prove the given theorems. For a more detailed overview of locales in Isabelle/HOL, including specific examples of subtyping, see Section 1.6.5 Locales in Isabelle/HOL.

By using locales, we receive the benefits of subtyping while keeping separate the essential properties of the UTXO model from the specifics of implementation. This promotes the model's modularity, extensibility, and maintainability, allowing us to focus on the core concepts and assumptions.

In summary, Isabelle/HOL locales offer a proper balance of modularity, extensibility, and separation of concerns for defining the UTXO model while utilizing subtyping, making it a more suitable choice than the alternatives.

2.4. UTXO model in Isabelle/HOL

2.4.1. Overview

In our approach to formalizing the UTXO model in Isabelle/HOL, we divide the specification into the abstract and implementation layers.

The abstract layer, as described in the *AbstractBasicUtxoLedger* theory, contains the core properties of UTXOs, transactions, and the ledger, but does not commit to concrete data types. This allows us to prove properties of the UTXO model in a generic way, which means that any valid interpretation of this abstraction will also have these properties.

The implementation layer, as described in the *BasicUtxoLedger* theory, provides a concrete, specific representation of the UTXO ledger, including UTXOs, transactions, and other entities. We use specific data types to define the ledger.

We show that the implementation model is an interpretation of the abstract model by mapping the constructs of the implementation model to their respective abstract representations. By proving that the assumptions defined in the abstract model hold, we can ensure that the implementation inherits all the properties of the abstract model. This approach allows us to apply the proven properties from the abstract model to our implementation, which ensures that it maintains the desired properties of the UTXO model, such as output uniqueness and conservation of the total token amount.

This layered approach of abstract and concrete specifications provides us with a foundation which is solid, yet modular and flexible. Several smaller specifications can also be easier to comprehend and analyze than one big specification.

Abstract model. The abstract model is designed to have the fundamental aspects of the UTXO model. We define the abstract UTXO model using several locales which represent the basic entities such as outputs, transactions, and the ledger. Each locale has the essential properties and operations of these entities. For example, the $basic_output$ locale ensures that each output has a non-zero amount, and the $basic_transaction$ locale enforces the conservation of total token amount in a transaction.

Implementation model. In the implementation model, we provide concrete realizations of the UTXO, transaction, and ledger. We define specific datatypes like $UTXO_type$ and TX_type to represent unspent outputs and transactions, respectively. Their types are backed by basic types that might be used in a real-world application. For instance, a TX_type has as set of $UTXO_type$ representing the input UTXOs, and $UTXO_type$ represents the token value it carries with the nat type.

The implementation also includes definitions of various functions and predicates, such as $apply_tx$ for applying transactions to the ledger and tx_valid for checking whether transactions are valid. These functions are defined in a manner consistent with the abstract model's properties, ensuring that the implementation maintains the UTXO model's integrity.

Note that elements of the implementation model can contain properties of several abstract elements, thus implementing several abstractions. This allows to consider the properties of the abstractions separately but then merge them into a fewer, relevant concrete implementations.

Implementation model as an interpretation of the abstract model. By using the Isabelle/HOL interpretation mechanism (see Section 1.6.5 Locales in Isabelle/HOL), we map the concrete entities to the abstract model's locales and assumptions to the and prove that the implementation satisfies all the of the abstract model's assumptions. This process not only validates the implementation against the abstract specification but also allows us to inherit all the proven properties, thus also guaranteeing of correctness for the UTXO implementation model.

2.4.2. Abstract UTXO model specification

The $AbstractBasicUtxoLedger$ Isabelle/HOL theory file provides a formalization of the abstract UTXO model for a ledger system's execution inside a single node. The formalization is performed in a modular way, where each component is encapsulated in a locale. This allows us to construct the abstract UTXO model step by step.

The theory file defines several locales which model outputs, transactions, the current UTXO set, and the transaction execution function.

The $basic_output$ locale encapsulates the main properties of an individual output, including the output itself and the associated amount. It imposes the condition that the amount must be

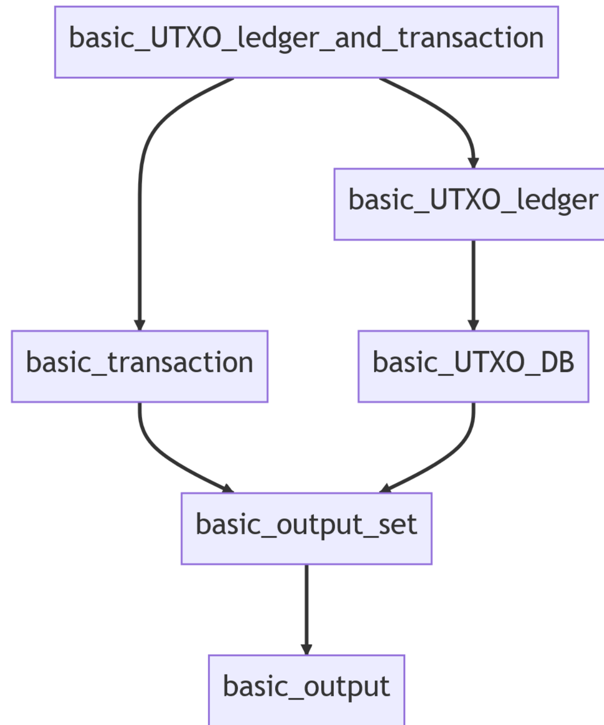

Figure 8. In the given graph diagram, six distinct locales are represented as nodes, and the relationships between them are depicted with directed edges. These locales model various aspects of the UTXO model. The locales extend one another – for example, locale $basic_output_set$ models a set of outputs and thus relies on the definition of $basic_output$.

greater than zero. The $basic_output_set$ locale extends $basic_output$ and represents a finite set of transaction outputs. It ensures that the output set is not empty, all elements within the set satisfy the assumptions of $basic_output$, and that the output identifiers are unique for different elements of the set.

The $basic_UTXO_DB$ locale represents an unspent transaction output database, which relies on a $basic_output_set$. It enforces the uniqueness of the UTXOs within the database and requires the sum of amounts in the database to be greater than zero.

The $basic_UTXO_ledger$ locale depends on $basic_UTXO_DB$, modeling the properties of a ledger system operating under the UTXO model. It ensures that the total amount of assets remains constant, the UTXO database remains unique after applying transactions, and no double-spending occurs.

Lastly, the $basic_UTXO_ledger_and_transaction$ locale combines the $basic_UTXO_ledger$ and $basic_transaction$ locales, capturing the properties of a ledger system and a transaction operating together under the UTXO model. It assumes the transaction to be valid within the context of the ledger, e.g., the transaction inputs are a subset of the unspent outputs in the ledger.

The diagram in 8 highlights the relationships between these locales. The $basic_output_set$ locale extends the $basic_output$ locale, while the $basic_transaction$ and $basic_UTXO_DB$ locales rely on the $basic_output_set$ locale. The $basic_UTXO_ledger$ locale depends on the $basic_UTXO_DB$ locale. The $basic_UTXO_ledger_and_transaction$ locale incorporates

both the $basic_transaction$ and $basic_UTXO_ledger$ locales.

As can be seen in Figure 8, the locales build upon each other to produce the final specification. Technically, such reuse of locales is achieved by defining assumptions requiring the fixed variables of one locale to belong to another specific locale. Consider the following simplified example of the $basic_UTXO_ledger$ locale:

```
1  locale basic_UTXO_ledger =
2    fixes DB :: 'o set
3      and utxo_id :: 'o ⇒ 'oid
4      and utxo_amt   :: 'o ⇒ nat
5      and apply_tx :: 'o set ⇒ 't ⇒ 'o set
6    assumes basic_UTXO_DB DB utxo_id utxo_amt
7      and ...
8  begin
9  ...
10 end
```

In this example, the $basic_UTXO_ledger$ locale is built on top of the $basic_UTXO_DB$ locale through the assumption $basic_UTXO_DB\ DB\ utxo_id\ utxo_amt$. This assumption requires that the fixed variables DB, $utxo_id$, and $utxo_amt$ satisfy the assumptions and constraints defined in the $basic_UTXO_DB$ locale. This approach can be considered as an alternative to extending the locale using $+$, as in $basic_UTXO_ledger = basic_UTXO_DB + \dots$.

In conclusion, the hierarchical structure and modular design of locales in the $AbstractBasicUtxoLedger$ Isabelle/HOL theory file allowed us to create a concise representation of the UTXO ledger system.

2.4.3. Implementation UTXO model specification

An implementation model provides a formally verified implementation of one or more abstract models. By proving the necessary properties within a specific context, we make sure that our model is not only theoretically sound but also practically viable. The $BasicUtxoLedger$ theory in Isabelle/HOL represents the concrete implementation of the abstract UTXO model. This section details how the abstract entities like outputs, transactions, and the ledger are instantiated with specific data types and functions to reflect a realistic UTXO ledger system.

```
1  type_synonym UTXO_id_type = hash
2  type_synonym addr_type = nat
3  type_synonym amount_type = nat
4
5  datatype UTXO_type = UTXO (UTXO_id: UTXO_id_type) (UTXO_addr: addr_type) (
     ↪ UTXO_amount: amount_type)
6
7  datatype TX_type = TX (inp: UTXO_type set) (out: UTXO_type set)
```

The implementation introduces concrete data types to represent the key components of the UTXO ledger:

- $UTXO_id_type$: A unique identifier for each UTXO, represented as a $hash$ type.
- $addr_type$: Represents the address associated with a UTXO, which is a natural number in this model.
- $amount_type$: Represents the token amount of a UTXO, also a natural number.

The $UTXO_type$ datatype encapsulates these elements to define a UTXO with an identifier, address, and amount. Similarly, the TX_type datatype represents transactions with sets of input and output UTXOs.

```
1 definition tx_valid :: UTXO_type set ⇒ TX_type ⇒ bool where
2   tx_valid db tx =
3     (UTXO_set_valid (inp tx)
4     ∧ UTXO_set_valid (out tx)
5     ∧ nonzero_outputs tx
6     ∧ inputs_in_db db tx
7     ∧ inp_out_same_amount_sum tx
8     ∧ inputs_not_in_outputs tx
9     ∧ outputs_not_in_db db tx)
```

Several functions and predicates are defined. Notably, tx_valid determines the validity of a transaction based on several criteria, such as input presence, output uniqueness, and the conservation of token amounts.

To validate the implementation, we interpret the abstract locales in the context of the concrete data types and functions. This involves proving the assumptions hold and that properties defined in the abstract model hold true when applied to the implementation. For example, the $concrete_basic_UTXO$ interpretation asserts that ledger in the implementation preserves the total token amount and prevent double-spending, as specified in the abstract model.

```
1 interpretation concrete_basic_UTXO:
2   basic_UTXO_ledger DB UTXO_id UTXO_amount apply_tx
3 proof ...
```

To summarize, the $BasicUtxoLedger$ theory provides a formally verified implementation of the UTXO ledger model. By proving the necessary properties within a specific, concrete context, we ensure that our overall UTXO model is not only theoretically sound but also practically viable.

2.4.4. Verification of UTXO model properties

In this section, we show that three main properties of the UTXO model hold for our formalization: constant supply, unspent output consumption, and no double spending. We use a series of theorems within the locale $using_basic_utxo_ledger$ to demonstrate that these properties hold in the context of the implementation UTXO model that we have created.

Here we demonstrate only the high-level definitions of these proofs, which rely on other lower-level definitions. We do not provide the full definitions of the properties as they would be too verbose. For instructions on accessing the full codebase, see Appendix A.

Constant Supply. The constant supply property ensures that the total amount of tokens in the system remains unchanged before and after transactions are applied. This property is formally verified in Isabelle/HOL as follows:

```
1 theorem constant_supply:
2   shows sum_amount DB = sum_amount (apply_valid_transaction)
3   by (simp add: apply_tx_amt_constant tx_valid)
```

This theorem shows that the sum of amounts in the ledger database DB remains constant when a valid transaction is applied, adhering to the property outlined in Equation 1.

Unspent Output Consumption. This property ensures that only outputs present in the ledger can be consumed by transactions:

```
1 theorem unspent_outputs_consumption:
2   shows ∀u ∈ inp tx. u ∈ DB ∧ u ∉ apply_valid_transaction
3   using apply_valid_transaction_def apply_transaction_to_db_def
    ↪ inputs_not_in_outputs_def tx_valid tx_valid_def inputs_in_db_def
4   by auto
```

This theorem states that all inputs in a valid transaction are in the ledger and are no longer present in the ledger after applying the transaction, which corresponds to Equation 2.

No Double Spending. This property prevents the same outputs from being spent more than once:

```
1 theorem no_double_spending:
2   assumes tx_valid DB tx1
3       and tx_valid (apply_transaction_to_db DB tx1 inp out) tx2
4   shows inp tx1 ∩ inp tx2 = {}
5   using apply_transaction_to_db_def assms(1) assms(2) inputs_in_db_def
    ↪ outputs_not_in_db_def tx_valid_def
6   by fastforce
```

This theorem ensures that no two distinct transactions can share an input. It first assumes that two valid transactions can be applied in sequence and then shows that these two transactions cannot share any inputs, thus maintaining the no double spending property as described in Equation 7.

Signature Verification, Conservation of Value, Progress. These properties were not explicitly proven in our work. Signature verification lies outside of this work's scope, as we were not aiming to model cryptographic protocols of distributed ledgers. Conservation of value was

used as an assumption of the model, so proving it in this work's context would be a tautology. The proof of the progress property was deemed not essential and is left as an opportunity for future work.

In summary, we have successfully verified three essential properties of the UTXO model using our formalization in Isabelle/HOL. The results show that our implementation model adheres to the rules of the UTXO model. Below, we provide a table summarizing the outcomes of each property's verification. The following table summarizes the results of our verification, highlighting both the results of our research as well as opportunities for future work to build upon it.

No.	Property	Verification Result
1	Constant Supply	Successfully Verified
2	Unspent Output Consumption	Successfully Verified
3	No Double Spending	Successfully Verified
4	Signature Verification	Out of Scope
5	Conservation of Value	N/A
6	Progress	Not Verified

Table 1. Summary of verification results for UTXO model properties

3. Modeling IOTA EUTXO in Isabelle/HOL

3.1. Scope of modeling

In the IOTA UTXO model, several extensions are introduced to the standard UTXO model. These extensions are designed to enhance the functionality of UTXOs, allowing for more complex types of transactions and stateful contracts. In our work we focus on two main extensions – Alias and Foundry outputs. The proposed NFT output type was not formalized, as it was deemed to be similar to the Foundry output. All of the proposed unlock conditions, as well as all of the proposed feature types are not formalized in our work due to the fact that they are not required to prove the essential properties of Alias and Foundry.

The table below summarizes the scope of our modeling.

No.	Component	Modeling Status
1	Basic Output	Modeled
2	Alias Output	Modeled
3	Foundry Output	Modeled
4	NFT Output	Not Modeled
5	Unlock Types	Not Modeled
6	Feature Types	Not Modeled

Table 2. Scope of the modeling of the proposed IOTA EUTXO extensions

3.2. Formalizing the IOTA UTXO extensions

3.2.1. Entities of the IOTA EUTXO model

Alias Output. The Alias Output has several fields that control its behavior and describe its state. Formally, an Alias Output, $AliasOutput$, can be represented as:

$$AliasOutput : \begin{cases} id : UniqueId \\ unlockConditions : UnlockConditions \\ amount : \mathbb{N} \\ aliasId : UniqueId \\ governor : Address \end{cases}$$

Where $aliasId : UniqueId$ is a hash uniquely representing the Alias during the stateful transformations and $governor : Address$ is the identifier of the entity that has the authority to govern the Alias Output.

Note that an $AliasOutput$ inherits all of the properties of a regular UTXO $Output$, as described in Section 2 Modeling UTXO in Isabelle/HOL. Subsequently, this is an equivalent representation of the above:

$$AliasOutput : Output \times \begin{cases} aliasId : UniqueId \\ governor : Address \end{cases}$$

Moreover, we can generalize even further:

$$AliasPart : \begin{cases} aliasId : UniqueId \\ governor : Address \end{cases}$$

$$AliasOutput : Output \times AliasPart;$$

Given these definitions, we can modify the $Transaction$ definition:

$$TransactionWithAlias : \begin{cases} id : UniqueId \\ inputs : Set(AliasOutput) \\ outputs : Set(AliasOutput) \\ unlockBlocks : Set(UnlockBlock) \end{cases}$$

Given that the properties that we analyze in this work rely only on the fields in $AliasPart$, we can apply logic analogous to the $AliasOutput$ generalization we have performed above. Having defined $AliasOutput : Output \times AliasPart$, we can further split the definition into components that we can reason about independently:

$$TransactionAliasParts : \begin{cases} aliasInputParts : Set(AliasPart) \\ aliasOutputParts : Set(AliasPart) \end{cases}$$

$$TransactionWithAlias : Transaction \times TransactionAliasParts$$

Notably, we do not retain the relationship from an $AliasPart$ to the respective $Output$, as they are independent in terms of operations and properties of the UTXO model – an operation or a property definition will only reference either $AliasPart$ or the $Output$ part of the transaction, but not both.

Similarly, a $LedgerWithAlias$ would contains the $AliasParts$:

$$\begin{aligned} LedgerWithAlias : & Set(AliasOutput) \\ & \equiv Set(Output) \times Set(AliasPart) \\ & \equiv Ledger \times Set(AliasPart) \end{aligned}$$

The ledger's state transition function, $ApplyTransactionWithAlias$, is, respectively:

$$\begin{aligned} & ApplyTransactionWithAlias : \\ & \quad LedgerWithAlias \to TransactionWithAlias \to LedgerWithAlias \end{aligned}$$

Foundry Output. The Foundry Output is used to manage the state and supply of user-defined native tokens within the IOTA EUTXO model. It allows the creation, transfer, and burning of native tokens on the ledger. Each Foundry Output inherits all properties of a basic UTXO Output and additionally manages token balances.

The native tokens are expected allowed to be persisted on any Output, thus we need to augment the existing Output definition:

$$NativeTokenPart : \begin{cases} id : UniqueId \\ tokenBalances : Map(TokenId \to \mathbb{N}) \end{cases}$$

$$Output : \begin{cases} id : UniqueId \\ unlockConditions : UnlockConditions \\ amount : \ \mathbb{N} \end{cases} \times NativeTokenPart;$$

Here $tokenBalances$ is a map where each $TokenId$ is associated with a quantity ($\mathbb{N}$), representing the balance of native tokens that are bound to this Output. Note that the $NativeTokenPart$ contains the id property, as the $NativeTokenPart$ is not uniquely identified by the amounts of tokens in the $tokenBalances$.

The new Output definition is expected to be used for all other UTXO types that extend the basic Output, such as the Alias Output.

A Foundry Output, denoted as $FoundryOutput$, is an extension of the basic UTXO Output

model. We define it formally as:

$$FoundryOutput : \begin{cases} id : UniqueId \\ unlockConditions : UnlockConditions \\ amount : \mathbb{N} \\ tokenBalances : Map(TokenId \to \mathbb{N}) \\ foundryId : UniqueId \end{cases}$$

In this model $foundryId$ represents a unique identifier for the Foundry Output. This is a stateful property that is maintained during ledger transformations by following the chain constraint rules. This identifier is equal to the native token type this Foundry is managing.

Due to the inheritance of the Output properties, we can simplify the above model to:

$$FoundryPart : \begin{cases} foundryId : UniqueId \end{cases}$$

$$FoundryOutput : Output \times FoundryPart;$$

Note that the new Output definition includes the $tokenBalances$ property. Subsequently, $FoundryOutput$ also has $tokenBalances$, which can include both the token type that is managed by this Foundry and other token types.

Transactions that involve Foundry Outputs are called $TransactionWithFoundry$. These transactions allow minting, transferring, and burning native tokens and are defined as follows:

$$TransactionWithFoundry : \begin{cases} id : UniqueId \\ inputs : Set(FoundryOutput) \\ outputs : Set(FoundryOutput) \\ unlockBlocks : Set(UnlockBlock) \end{cases}$$

We can further generalize this definition and define a way to extend an existing Transaction with Foundry rules:

$$TransactionFoundryParts : \begin{cases} foundryInputParts : Set(FoundryPart) \\ foundryOutputParts : Set(FoundryPart) \\ nativeInputs : Set(NativeTokenPart) \\ nativeOutputs : Set(NativeTokenPart) \end{cases}$$

$$TransactionWithFoundry : Transaction \times TransactionFoundryParts$$

Here we opt to use the $FoundryPart$ and $NativeTokenPart$ to represent the Foundries and native token sets participating in the transaction, respectively. The validity of a $FoundryTransaction$ relies on the proper management of token quantities. Token amounts

must remain constant unless the transaction explicitly includes a Foundry Output for the token type being minted or burned.

In a similar structure to the $LedgerWithAlias$, a $LedgerWithFoundry$ contains $FoundryParts$ and integrates native tokens across various outputs:

$$
\begin{aligned}
LedgerWithFoundry : &Set(FoundryOutput)\\
&\equiv Set(Output) \times Set(FoundryPart) \times Set(NativeTokenPart)\\
&\equiv Ledger \times Set(FoundryPart) \times Set(NativeTokenPart)
\end{aligned}
$$

The state transition function for a ledger that includes Foundry Outputs, $ApplyTransactionWithFoundry$, is defined as follows:

$$
\begin{aligned}
&ApplyTransactionWithFoundry :\\
&\quad LedgerWithFoundry \to TransactionWithFoundry \to LedgerWithFoundry
\end{aligned}
$$

Combining Alias and Foundry Outputs. The Alias and Foundry models can be effectively used together. In this section we demonstrate how these models can be combined in a single model.

We first define the $BasicOutput$ type, which is an extension of the $Output$ type with the :

$$
BasicOutput : \begin{cases}
id : UniqueId\\
unlockConditions : UnlockConditions\\
amount : \mathbb{N}\\
tokenBalances : Map(TokenId \to \mathbb{N})
\end{cases}
$$

To define $IotaExtendedOutput$, we then allow three distinct types of outputs:

$$
\begin{aligned}
IotaExtendedOutput : &BasicOutput +\\
&(BasicOutput \times AliasPart) +\\
&(BasicOutput \times FoundryPart);
\end{aligned}
$$

Which we further simplify using the definitions of $AliasOutput$ and $FoundryOutput$:

$$
\begin{aligned}
IotaExtendedOutput : &BasicOutput +\\
&AliasOutput +\\
&FoundryOutput;
\end{aligned}
$$

Here $IotaExtendedOutput$ is a sum type – an instance of the $IotaExtendedOutput$ can either be a $BasicOutput$, an $AliasOutput$, or a $FoundryOutput$.

Using $IotaExtendedOutput$ for transactions allows the model to support transactions that simultaneously handle Alias and Foundry outputs:

$$TransactionIotaExtended : \begin{cases} id : UniqueId \\ inputs : Set(IotaExtendedOutput) \\ outputs : Set(IotaExtendedOutput) \\ unlockBlocks : Set(UnlockBlock) \end{cases}$$

The ledger in this combined model is simply a set of the extended outputs:

$$LedgerIotaExtended : Set(IotaExtendedOutput);$$

Finally, the state transition function is then be defined as:

$$\begin{aligned} &ApplyTransactionIotaExtended : \\ &\quad LedgerIotaExtended \rightarrow TransactionIotaExtended \rightarrow LedgerIotaExtended \end{aligned}$$

3.2.2. Properties of the IOTA EUTXO model

The focus of IOTA UTXO extensions is to introduce statefulness to the UTXO model. Thus, the main new properties that the IOTA EUTXO defines are related to the transition of state throughout the lifetime of a ledger. Specifically, new transaction validity rules are introduced to allow for new data continuity and governance rules.

Continuity of State (Chain Constraint). An essential property of the Alias Output and Foundry Output is their continuity within the ledger's history. This is also referred to as the *chain constraint*. The property ensures that once a stateful output is created, it has a continuous existence across the ledger until explicitly deleted and removed from the ledger. For more on the motivation of the chain constraint in the IOTA EUTXO, see Section 1.4 IOTA EUTXO model.

For the sake of simplicity, let us assume that the stateful output in question is an Alias Output. Subsequently, the continuity can be mathematically defined as:

$$\begin{aligned} &\forall l_1, l_2 \in LedgerWithAlias, t \in TransactionWithAlias. \\ &\quad IsValidAliasTransacton(l_1, t) \\ &\quad \wedge l_2 = ApplyTransactionWithAlias(l_1, t) \\ &\quad \wedge a \in t.aliasInputs \\ &\qquad \rightarrow a \in l_2 \vee TransactionDeletesAlias(t, a) \end{aligned} \tag{18}$$

The formula asserts that for any two ledger states, l_1 and l_2, and a transaction t, if t is a valid Alias transaction in the context of l_1, and l_2 is the result of applying t to l_1, i.e., l_2 is the next state

of the ledger l_1, then for every Alias Output a consumed as an input in t, one of two conditions must hold in the subsequent ledger state l_2:

1. The Alias Output a must continue to exist in l_2 ledger.
2. The transaction t explicitly deletes the Alias Output a, indicating a deliberate end to its lifecycle within the ledger.

This formulation ensures that Alias state is either consistently carried forward during ledger state transitions, thus preserving its continuity, or it is explicitly removed.

While the statement (18) represents the essence of Alias continuity, it does not explicitly state that the Alias outputs form a single continuous range, or a *chain*, of ledger state transitions.

A less strict statement can be a more robust way to represent the same concept. Suppose we have a continuous, ordered history of a single ledger $LedgerWithAliasHistory \subseteq LedgerWithAlias$, where $l_i \in LedgerWithAliasHistory$ represents the ith ledger state. Then we can express the property in question as an invariant:

$$\begin{aligned} &\forall l_A, l_B \in LedgerWithAliasHistory, a \in AliasParts(l_A). \\ &\quad (A \leq B) \land (a \in AliasParts(l_A)) \\ &\qquad \rightarrow (l_A = l_B) \lor \exists l_{A+1}, \ldots, l_{B-1} \\ &\qquad\quad \land \forall i \in \{A+1, \ldots, B-1\}.a \in AliasParts(l_i) \end{aligned} \tag{19}$$

In this expression:

1. l_A and l_B are two specific states in the ledger history, with $A \leq B$ indicating that l_A precedes or is equal to l_B.
2. The Alias Output a is present in the ledger state l_A.
3. It is asserted that if l_A and l_B share the same Alias Output a, then either l_A and l_B are the same state, or there exists a sequence (or chain) of ledger states starting from the state immediately following l_A, denoted as $l_{A+1}, l_{A+2}, \ldots, l_B$, which leads up to l_{B-1}.
4. Each state in this sequence also contains the Alias Output a.

This representation ensures that once an Alias Output is added to the ledger, it is either present in the following state or is deleted.

Foundry Total Supply Consistency. An essential property specific to the Foundry Output in the IOTA EUTXO model is the consistency of the total supply of native tokens. This property ensures that the total number of each type of native token remains constant unless explicitly modified through transactions mint or burn tokens.

$$
\begin{aligned}
&\forall l_1, l_2 \in LedgerWithFoundry, t \in TransactionWithFoundry, \\
&\quad f_1 \in FoundryOutputs(l_1), f_2 \in FoundryOutputs(l_2). \\
&\quad IsValidFoundryTransaction(l_1, t) \\
&\quad \land l_2 = ApplyTransactionWithFoundry(l_1, t) \\
&\quad \land f_1.foundry_id = f_2.foundry_id \\
&\qquad \rightarrow f_2.supply = f_1.supply + t_{mint}(f_1) - t_{burn}(f_1)
\end{aligned} \tag{20}
$$

In this expression:

1. l_1 and l_2 represent two specific states in the ledger history, where l_2 is the state resulting from applying a transaction t to the state l_1.
2. f_1 and f_2 are specific Foundry Outputs within the states l_1 and l_2 respectively, both associated with the same foundry identified by $foundry_id$.
3. The transaction t is validated as a proper Foundry transaction in the context of the ledger state l_1 by $IsValidFoundryTransaction(l_1, t)$.
4. The terms $t_{mint}(f_1)$ and $t_{burn}(f_1)$ represent the amount of native tokens minted and burned, respectively, associated with the foundry f during the transaction t.

The equation states that for any transaction t, if a foundry output f_1 in ledger l_1 transitions to f_2 in ledger l_2 (identified by the same $foundry_id$), the new supply $f_2.supply$ in l_2 is the original supply $f_1.supply$ adjusted by the tokens minted (t_{mint}) and burned (t_{burn}) specific to that $foundry_id$ within the transaction t.

Note that, according to the property, any minting or burning of native tokens would require the update of the respective foundry. Thus, The foundry itself must participate in the transaction in order to transition its state and update its total supply property.

Alternatively, this property can also be expressed with the following invariant:

$$
\begin{aligned}
&\forall l \in LedgerWithFoundry, f \in FoundryOutputs(l). \\
&\quad \rightarrow f.supply = \sum_{o \in Outputs(l)} o.tokens[f.foundry_id]
\end{aligned} \tag{21}
$$

The equation states that, for each ledger state l containing Foundry Outputs f, the supply of native tokens $f.supply$ is equal to the sum of those specific tokens across all ledger outputs o.

3.3. IOTA EUTXO model in Isabelle/HOL

In this section we describe our overall approach to modeling the IOTA EUTXO model in Isabelle/HOL as well as our current progress formalizing the model.

3.3.1. Modeling UTXO extensions in Isabelle/HOL in a modular way

At a high-level view, the overall approach is to loosely, informally, tie the IOTA EUTXO model as described in RFC 38 to a concrete specification of the IOTA EUTXO model – an Isabelle/HOL locale called $IotaUtxoLedger$. It would then be shown that the $IotaUtxoLedger$ locale satisfies the properties of the $BasicUtxoLeger$ – a locale modeling the base UTXO model.

The modular approach allows flexibility in the specification, enabling future extensions or modifications to be added with less effort. This is important for keeping the model relevant and practical as distributed ledger and formal method technologies progress and evolve.

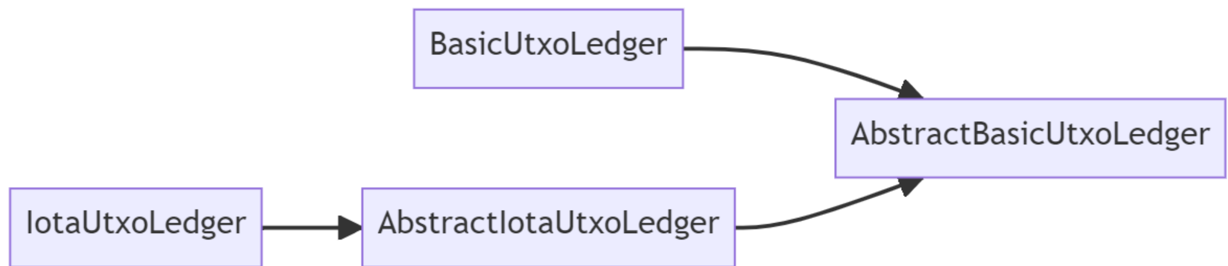

Figure 9. A high-level view of the locales used in the specification. The nodes represent the locales and the edges show that the source locale implements the destination locale.

This high-level structuring also allows a clear separation of concerns, allowing different aspects of the EUTXO model to be modeled independently. Moreover, $AbstractIotaUtxoLedger$ can itself be composed of a "basic" UTXO ledger component (such as $AbstractBasicUtxoLedger$), an Alias ledger component, a Foundry ledger component, and so on.

3.3.2. Abstract Alias UTXO

Alias Output Set. We begin by defining the *alias_output_set* locale, which represents a collection of Alias outputs.

```
1 locale alias_output_set =
2   fixes alias_outs :: 'ao set
3     and aid :: 'ao ⇒ 'id
4   assumes aliases_unique: fset_unique alias_outs aid
```

In this locale, *alias_outs* is the set of Alias outputs, and *aid* is a function that retrieves the unique identifier of each Alias output. The key assumption here is *aliases_unique*, which ensures that each Alias output in the set has a distinct identifier.

Alias Transaction. Next, we define the *alias_transaction* locale, encapsulating the behavior and constraints of transactions involving Alias outputs.

```
1 locale alias_transaction =
2   fixes inp_alias_outs :: 'ao set
3     and out_alias_outs :: 'ao set
4     and aid :: 'ao ⇒ 'id
5   assumes inp_valid: alias_output_set inp_alias_outs aid
6     and out_valid: alias_output_set out_alias_outs aid
```

Here, *in_alias_outs* and *out_alias_outs* represent the sets of Alias outputs used as inputs and outputs in a transaction, respectively. The assumptions *inp_valid* and *out_valid* ensure that these sets adhere to the Alias output set constraints that we defined above, such as *aliases_unique*.

Alias Ledger. The *alias_ledger* locale represents the state of the ledger containing Alias outputs.

```
1 locale alias_ledger =
2   fixes ledger :: 'ao set
3     and aid :: 'ao ⇒ 'id
4   assumes unique_ids: fset_unique ledger aid
```

The *ledger* is a set of Alias outputs, and *aid* is a function that retrieves the unique identifier of each Alias output. The *unique_ids* assumption ensures that each Alias output in the ledger has a distinct identifier.

Alias Transaction in Ledger. The *alias_transaction_in_ledger* locale combines the *alias_transaction* and *alias_ledger* locales to model transactions that operate on the Alias ledger.

```
1  locale alias_transaction_in_ledger =
2    alias_transaction +
3    alias_ledger +
4    assumes inp_in_ledger: inp_alias_outs ⊆ ledger
5      and out_ids_not_in_ledger:
6        fset_intersect out_alias_outs ledger aid ⊆ fset_intersect
      ↪ inp_alias_outs out_alias_outs aid
7  begin
8
9  definition apply_transaction where
10   apply_transaction = (ledger - inp_alias_outs) ∪ out_alias_outs
11
12 lemma apply_transaction_preserves_validity:
13   shows alias_ledger (apply_transaction) aid
```

The *inp_in_ledger* assumption ensures that all input Alias outputs used in the transaction are present in the ledger. The *out_ids_not_in_ledger* assumption states that the Alias identifiers in the transaction outputs that are already in the ledger must be a subset of the identifiers in both the transaction inputs and outputs. This constraint prevents the creation of new Alias outputs with identifiers that already exist in the ledger.

The *apply_transaction* definition describes the state transition of the ledger when a transaction is applied. It removes the input Alias outputs from the ledger and adds the output Alias outputs.

Finally, the *apply_transaction_preserves_validity* lemma shows that applying a valid Alias transaction to a valid Alias ledger results in a new valid Alias ledger state.

3.3.3. Foundry UTXO

Foundry Output Set. We begin the formal definition with the *foundry_output_set* locale, representing a collection of Foundry outputs. Each output in this set manages a unique set of native tokens, identified by a unique identifier.

```
1 locale foundry_output_set =
2   fixes foundry_outs :: 'fo set
3     and fid :: 'fo ⇒ 'fid
4     and total_supply :: 'fo ⇒ nat
5   assumes foundries_unique: fset_unique foundry_outs fid
6     and fin: finite foundry_outs
```

In this locale, *foundry_outs* is the set of Foundry outputs, *fid* retrieves the unique identifier for each Foundry output, and *total_supply* – the total number of native tokens that the Foundry claims to be in circulation in the ledger. The assumption *foundries_unique* ensures that each output in the set has a distinct identifier. The *fin* assumption ensures that we are working with a finite set of outputs.

Native Output Set. To represent the native tokens being carried by other outputs, we define a Native output type:

```
1 locale native_utxo_set =
2   fixes native_utxos :: 'nuo set
3     and id :: 'nuo ⇒ 'nid
4     and tokens :: 'nuo ⇒ ('fid ⇀ nat)
5   assumes utxos_unique: fset_unique native_utxos id
6     and fin: finite native_utxos
```

In the locale *native_utxo_set*, *native_utxos* is a set of Native outputs, *id* it the unique identifier for each Native output, and *tokens* is a map from token IDs to a natural number representing their amount. The assumption *utxos_unique* ensures that each output in the set has a unique identity. The *fin* assumption ensures that we are working with a finite set of Native outputs.

Foundry Transaction. We then define the *foundry_transaction* locale, encapsulating the behavior and constraints of transactions that involve Foundry outputs.

```
1 locale foundry_transaction_definitions =
2   fixes inp_foundry_outs :: 'fo set
```

```
3      and out_foundry_outs :: 'fo set
4      and inp_native_outs :: 'nuo set
5      and out_native_outs :: 'nuo set
6      and fid :: 'fo ⇒ 'fid
7      and id :: 'nuo ⇒ 'nid
8      and tokens :: 'nuo ⇒ ('fid ⇀ nat)
9      and total_supply :: 'fo ⇒ nat
10   assumes inp_valid: foundry_output_set inp_foundry_outs fid
11     and native_utxo_set inp_native_outs id
12     and out_valid: foundry_output_set out_foundry_outs fid
13     and native_utxo_set out_native_outs id
14     and foundry_not_present_input_output_tokens_equal
15     and foundry_ids_persisted
16     and foundry_present_total_supply_updated
```

Within this locale:

- *inp_foundry_outs* and *out_foundry_outs* represent the sets of Foundry outputs as inputs and as the outputs of the transaction, respectively.
- *inp_native_outs* and *out_native_outs* are the sets of Native outputs in the transaction.
- *fid*, *id*, *tokens*, and *total_supply* serve the same purposes as in the previous locales.

The assumptions made within this locale ensure that the transaction is correct:

- *inp_valid* and *out_valid* confirm the validity of Foundry output sets, using the definitions we defined earlier. Analogous assumptions are defined for Native outputs.
- *foundry_not_present_input_output_tokens_equal* ensures that the tokens are consistent in the transaction when if the respective Foundry is not a part of the transaction.
- *foundry_ids_persisted* ensures that the Foundry identifiers are maintained across the transaction.
- *foundry_present_total_supply_updated* verifies that the total supply of tokens is correctly adjusted for foundries in the transaction, accounting for cases such as minting or burning of native tokens.

Based on the definitions in the *foundry_transaction* locale, we define the properties and operations, such as token minting, transferring, and burning within transactions. These rules are expressed in the definitions for *foundry_not_present_input_output_tokens_equal* and *foundry_present_total_supply_updated* that we have used in the assumptions for the *foundry_transaction* locale.

```
1 definition foundry_not_present_input_output_tokens_equal where
2   foundry_not_present_input_output_tokens_equal ≡ ∀token_id. token_id ∉
    ↪ fset_map inp_foundry_outs fid ⟶
```

```
3     input_tokens token_id = output_tokens token_id
4
5 definition foundry_present_total_supply_updated where
6   foundry_present_total_supply_updated ≡ ∀token_id. ∀fi ∈ inp_foundry_outs.
    ↪ fid fi = token_id ⟶
7     (
8       (∃fo ∈ out_foundry_outs. fid fo = token_id ∧ total_supply fo =
    ↪ total_supply fi + output_tokens token_id - input_tokens token_id)
9       ∨ ¬(∃fo ∈ out_foundry_outs. fid fo = token_id)
10    )
```

The property *foundry_not_present_input_output_tokens_equal* ensures that if a token ID is not present among the Foundry outputs in the input set, then the number of native tokens for that ID remains unchanged in the transaction output. Otherwise, the *foundry_present_total_supply_updated* property ensures that for every token ID associated with Foundry outputs involved in the transaction, the total supply of tokens is accurately updated – the Foundry output with the same token ID in the transaction output set has the total supply updated to reflect the amounts of respective native tokens minted and burned during the transaction.

These definition ensure that token quantities are transformed correctly: either they remain constant or are updated only if their respective Foundry output is part of the transaction inputs.

Foundry Ledger. The *foundry_ledger* locale represents the state of the ledger containing both Foundry and Native outputs.

```
1 locale foundry_ledger =
2   fixes fo_ledger :: 'fo set
3     and nuo_ledger :: 'nuo set
4     and fid :: 'fo ⇒ 'fid
5     and id :: 'nuo ⇒ 'nid
6     and total_supply :: 'fo ⇒ nat
7     and tokens :: 'nuo ⇒ ('fid ⇀ nat)
8   assumes foundry_output_set_valid: foundry_output_set fo_ledger fid
    ↪ total_supply
9     and native_utxo_set_valid: native_utxo_set nuo_ledger id tokens
10    and total_supply_consistent: ∀f ∈ fo_ledger. total_supply f =
    ↪ sum_nuo_tokens nuo_ledger tokens (fid f)
```

The *fo_ledger* and *nuo_ledger* represent the sets of Foundry and Native outputs in the ledger, respectively. The functions *fid*, *id*, *total_supply*, and *tokens* serve the same purposes as described in the previous locales.

The assumptions ensure that the ledger state is valid:

- *foundry_output_set_valid* and *native_utxo_set_valid* ensure that the Foundry and Native output sets in the ledger are valid, as per the definitions in their respective locales.

- *total_supply_consistent* ensures that for each Foundry output in the ledger, its total supply matches the sum of the corresponding native tokens held in the Native outputs.

Foundry Transaction in Ledger. The *foundry_transaction_in_ledger* locale combines the *foundry_transaction* and *foundry_ledger* locales to model transactions that operate on the Foundry ledger.

```
1  locale foundry_transaction_in_ledger =
2    foundry_transaction inp_foundry_outs out_foundry_outs inp_native_outs
     ↪ out_native_outs fid id tokens total_supply +
3    foundry_ledger fo_ledger nuo_ledger fid id total_supply tokens
4    for fid :: 'fo ⇒ 'fid
5      and id :: 'nuo ⇒ 'nid
6      and total_supply :: 'fo ⇒ nat
7      and tokens :: 'nuo ⇒ ('fid ⇀ nat)
8      and inp_foundry_outs out_foundry_outs :: 'fo set
9      and inp_native_outs out_native_outs :: 'nuo set
10     and fo_ledger :: 'fo set
11     and nuo_ledger :: 'nuo set
12   +
13   assumes inp_in_ledger: inp_foundry_outs ⊆ fo_ledger ∧ inp_native_outs ⊆
     ↪ nuo_ledger
14     and foundry_ids_persisted: fset_intersect out_foundry_outs fo_ledger fid
     ↪  ⊆ fset_intersect inp_foundry_outs out_foundry_outs fid
15     and out_not_in_ledger: fset_intersect out_native_outs nuo_ledger id = {}
```

The additional assumptions ensure that the transaction is valid within the context of the ledger:

- *inp_in_ledger* ensures that all input Foundry and Native outputs used in the transaction are present in the ledger.
- *foundry_ids_persisted* ensures that the Foundry identifiers in the transaction outputs that are already in the ledger must be a subset of the identifiers in both the transaction inputs and outputs. This constraint prevents the creation of new Foundry outputs with identifiers that already exist in the ledger.
- *out_not_in_ledger* ensures that the Native outputs in the transaction outputs are not already present in the ledger.

The locale also defines the *apply_transaction_foundry* and *apply_transaction_native_tokens* functions, which describe the state transitions of the Foundry and Native output sets in the ledger when a transaction is applied.

```
1  definition apply_transaction_foundry :: 'fo set where
2    apply_transaction_foundry = (fo_ledger - inp_foundry_outs) ∪
     ↪ out_foundry_outs
3
```

```
4 definition apply_transaction_native_tokens :: 'nuo set where
5   apply_transaction_native_tokens = (nuo_ledger - inp_native_outs) ∪
    ↪ out_native_outs
```

The locale also includes several lemmas that prove the consistency of the total supply after a transaction is applied:

- *total_supply_consistent_if_foundry_not_present* ensures that if a Foundry output was not part of the transaction inputs, its total supply remains consistent with the sum of the corresponding native tokens in the updated Native output set.
- *total_supply_constant_if_foundry_in_transaction* ensures that if a Foundry output is present in the transaction outputs, its total supply is updated to reflect the minted and burned native tokens from Native outputs.

Finally, the *apply_transaction_preserves_validity* lemma proves that applying a valid Foundry transaction to a valid Foundry ledger results in a new valid Foundry ledger state

```
1 lemma apply_transaction_preserves_validity:
2   shows foundry_ledger apply_transaction_foundry
    ↪ apply_transaction_native_tokens fid id total_supply tokens
```

3.3.4. Implementation model

While we chose to make the abstract models modular, we believe that the implementation model should instead try to be closer to the model of an implementation that would be used in a real-life scenario. Thus, we define a single ledger representing the IOTA EUTXO model's ledger.

UTXO Data Types. We begin by defining the data types that represent different kinds of UTXOs in our model, including the Alias UTXOs.

```
1 datatype BasicT = Basic (basic_id : hash) (basic_amount : nat) (native_tokens
    ↪ : (hash ⇀ nat))
2 datatype AliasT = Alias (alias_id : hash)
3 datatype FoundryT = Foundry (foundry_id : hash)  (total_supply : nat)
4
5 datatype UTXO = BasicU BasicT
6              | AliasU BasicT AliasT
7              | FoundryU BasicT FoundryT
```

In this definition, $BasicT$ represents the basic structure of a UTXO, including its identifier (a hash, as described in Section 2.2 Hashing) and amount. The *native_tokens* contains the map native token IDs to their respective amounts. $AliasT$ is a specific type for Alias UTXOs, encapsulating attributes unique to Alias outputs, such as *alias_id*. Similarly, the $FoundryT$ type encapsulates the *foundry_id* and *total_supply* properties relevant to Foundry UTXOs. The $UTXO$ datatype then combines these to form a unified representation of UTXOs in the ledger,

allowing for $BasicT$, $AliasT$, and $FoundryT$ types. More formally, $UTXO$ is a sum type with three constructors – $BasicU$, $AliasU$, and $FoundryU$.

Using the data types defined above, we provide definitions for the implementation of a transaction and ledger:

```
1 datatype TransactionT = TransactionT (tx_inp: UTXO set) (tx_out: UTXO set)
2 type_synonym Ledger = UTXO set
```

Note that the defined types are still quite abstract, because we have not specified any constraints on them that would be applied in a real-life scenario. To give a specific example – while the $Ledger$ definition might look correct at first glance, the definition above does not ensure that the $Ledger$ is finite, which, in real-life, it would definitely be. Because data types in Isabelle/HOL do not provide a way to specify constraints such as preconditions or postconditions, we can use a locale to constrain our types:

```
1 locale iota_utxo_ledger_implementation = hashes +
2   fixes DB :: Ledger
3     and AliasDB :: AliasT set
4     and FoundryDB :: FoundryT set
5     and ValidTransaction :: TransactionT
6   assumes db_valid: DB_valid DB
7     and AliasDB = take_alias DB
8     and FoundryDB = take_foundry DB
9     and tx_valid: transaction_valid DB ValidTransaction
```

The *iota_utxo_ledger_implementation* locale serves to represent the environment constraints that are required for an accurate definition of a UTXO model implementation. Note that, while the locale feature is used here, we use it to represent a different concept than when we used a locale to define a contract of an abstraction (for example, the locale *alias_output_set*) – in this case, we use it as an environment with implicit assumptions instead of a building block with properties that can be used outside of it.

UTXO Operations and Ledger State. We define functions that interact with the data types described above. *apply_tx* represents the application of a transaction to the ledger, resulting in a new ledger state:

```
1 definition apply_tx :: Ledger ⇒ TransactionT ⇒ Ledger where
2   apply_tx DB tx = (DB - tx_inp tx) ∪ tx_out tx
```

Here, *apply_tx* takes the current ledger state DB and a transaction tx, and computes the new ledger state by removing the inputs and adding the outputs of the transaction. This mimics the implementation of this functionality in a real-world system.

Tying Implementation to Abstract. The final step in our implementation model is to tie these concrete representations back to our abstract model. This means demonstrating that our implementation adheres to the properties and constraints defined in the abstract locales.

For instance, we need to show that transactions involving Alias UTXOs in our implementation respect the rules set out in the *alias_transaction* locale. This requires verifying that Alias UTXOs in the transaction inputs and outputs are unique and follow the state transition rules:

```
1 definition take_alias :: UTXO set ⇒ AliasT set where
2   take_alias utxos = {a. ∃b. AliasU b a ∈ utxos}
3
4 definition AliasDB :: AliasT set where
5     AliasDB = take_alias DB
6
7 interpretation concrete_iota_alias_ledger: alias_ledger AliasDB alias_id
8   by ...
```

We define *take_alias* to represent the Alias UTXOs in our ledger and then prove that these Alias UTXOs together with the *alias_id* property "getter" function of *AliasT* form a valid *alias_ledger*.

Once the interpretation is proven, we can then use the properties of the *alias_ledger*. In the following snippet, we use the *unique_ids* property of *alias_ledger*, referenced through the name of the fact *concrete_iota_alias_ledger* that we proved above, to show that the identifiers in *AliasDB* are unique:

```
1 theorem alias_uniq:
2   ⋀a1 a2. a1 ∈ AliasDB ∧ a2 ∈ AliasDB ∧ alias_id a1 = alias_id a2 ⟹ a1 =
    ↪ a2
3   by (meson concrete_iota_alias_ledger.unique_ids fset_unique_def)
```

Besides Alias, an analogous interpretation is made for the *foundry_ledger*. We demonstrate that the foundry outputs and basic outputs of the ledger form a valid *foundry_ledger* locale:

```
1 interpretation concrete_iota_foundry_ledger: foundry_ledger FoundryDB (
    ↪ take_basics DB) foundry_id basic_id total_supply native_tokens
```

To summarize, by establishing these connections from the abstract model to the implementation model, we ensure that our implementation is not only theoretically sound but also can be practically implemented.

3.3.5. Verification of IOTA EUTXO model's properties

In our formalization of the IOTA EUTXO model, we have formalized and successfully verified all three main properties of the base UTXO model as well as three additional properties specific to the IOTA EUTXO model – Alias chain constraint, Foundry chain constraint, and Foundry total supply consistency.

Base UTXO properties. The three main properties of the base UTXO model were proven to hold for the IOTA EUTXO model: constant supply, unspent output consumption,

and no double spending. These properties are proven in a way analogous to the base UTXO implementation, so we do not go into further detail and instead refer the readers to the respective base UTXO model's property verification section.

```
theorem constant_supply:
  shows sum_amount DB = sum_amount (apply_valid_transaction)

theorem unspent_outputs_consumption:
  shows ∀u ∈ tx_inp ValidTransaction. u ∈ DB ∧ u ∉ apply_valid_transaction

theorem no_double_spending:
  assumes transaction_valid DB tx1
      and nextDB = (apply_tx DB tx1 tx_inp tx_out)
      and transaction_valid nextDB tx2
    shows tx_inp tx1 ∩ tx_inp tx2 = {}
```

Alias Chain Constraint. Alias continuity is ensured through a chain constraint mechanism, which guarantees that an Alias output, once created, maintains a continuous existence across the ledger states until it is explicitly removed.

```
theorem alias_continuity:
  assumes alias ∈ AliasDB
      and inputs = (tx_inp ValidTransaction)
      and outputs = (tx_out ValidTransaction)
      and nextDB = apply_valid_transaction
    shows alias ∈ take_alias (nextDB) ∨ transaction_removes_alias alias
   ↪ inputs outputs
```

This theorem states that if an alias is part of the current ledger, after a valid transaction is applied, the alias either remains in the database or is explicitly removed, adhering to the specified chain constraints as formalized in Equation 18.

Foundry Chain Constraint. Similarly, the Foundry chain constraint ensures the continuity of Foundry outputs.

```
theorem foundry_continuity:
  assumes foundry ∈ FoundryDB
      and inputs = (tx_inp ValidTransaction)
      and outputs = (tx_out ValidTransaction)
      and nextDB = apply_valid_transaction
    shows foundry ∈ take_foundry (nextDB) ∨ transaction_removes_foundry
   ↪ foundry inputs outputs
```

In an analogous way to the Alias continuity, this theorem states that if a foundry is a part of the current ledger, then applying valid transaction will result in a ledger that either has the foundry state maintained, or the transaction had explicitly removed the foundry. This property is similar to the Alias chain constraint and aligns with the requirements as expressed in the more generalized chain constraint model (see Equation 18).

Foundry Total Supply Consistency. The Foundry total supply consistency property ensures that the total supply of native tokens defined in their respective Foundry outputs remains accurate across all transactions.

```
1 theorem foundry_native_token_amount_constant:
2   assumes nextDB = apply_valid_transaction
3       and f ∈ (take_foundry nextDB)
4   shows total_supply f = ledger_total_token_sum nextDB (foundry_id f)
```

This theorem asserts that for any Foundry output present in the ledger after a transaction, the total supply of its native tokens aligns with the sum of the respective native token amounts across the whole ledger. This theorem aligns with the model we have defined in Equation 21.

To summarize, we have successfully verified the basic properties of the base UTXO model as well as three properties of the IOTA EUTXO model. Below, a table is provided to clarify which properties have been formalized and verified.

No.	Property	Verification Result
	Base UTXO Properties	
1	Constant Supply	Successfully Verified
2	Unspent Output Consumption	Successfully Verified
3	No Double Spending	Successfully Verified
4	Signature Verification	Out of Scope
5	Conservation of Value	N/A
6	Progress	Not Verified
	IOTA EUTXO Properties	
7	Chain Constraint Alias	Successfully Verified
8	Chain Constraint Foundry	Successfully Verified
9	Foundry Total Supply Consistency	Successfully Verified

Table 3. Summary of verification results for IOTA EUTXO model properties

3.3.6. Comparison with Cardano UTXO model

The Cardano UTXO model is an Isabelle/HOL theory file which formalizes the base UTXO model. The formalization defines the key components, transition rules, and properties of the UTXO model. For more information about the Cardano UTXO model, see Section 1.5.3.

One of the main differences between our and Cardano's models is the modeling of sequences of transactions. In our work, the history of the ledger, or, in other words, the sequence of transactions that has been applied to the ledger, is not modeled. This decision was made because the modeling of the sequence of transactions was not required for proving the properties that are verified in this work. The Cardano model, however, defines the effect of a transaction sequence on the system state. Furthermore, the Cardano model defines UTXO transaction fees – amounts of tokens that are consumed when a transaction is applied. In our work, we have not modeled the transaction fees explicitly, as the fee collection is equivalent to having another output with a small amount of tokens in the transaction.

The formalization defines and proves three key properties of the UTXO system:

1. **No Double Spending:** In a valid transaction sequence, the inputs of all transactions are disjoint, ensuring that no UTXO is spent more than once. This property is identical to the no double spending property verified in our UTXO and IOTA EUTXO models.

2. **Ledger is Outputs minus Inputs:** The UTXO set at any point is equivalent to the difference between the outputs and inputs of all transactions applied so far. As our models do not track the history of transactions applied to the ledger, we have not verified this property. Instead, we used the assumption that the input and output token sums in a transaction are equal to verify other properties.

3. **Constant Money Supply:** The total money supply (sum of all UTXO values and fees) remains constant as transactions are applied, demonstrating the conservation of value in the system. This property, with the exception of the fee tracking, is equivalent to the constant supply property we have verified in our UTXO and IOTA EUTXO models.

In summary, the Cardano UTXO model is not as comprehensive as the UTXO and IOTA EUTXO models presented in this work. Nevertheless, it offers insights in managing ledger history as transaction sequences and calculating transaction fees.

Results

This work has verified the functional correctness of the changes proposed in RFC 38 by formalizing the essential parts of the IOTA EUTXO model and verifying it using the Isabelle/HOL proof assistant. In this work, we:

1. Provided mathematical formalizations for the UTXO model, the IOTA EUTXO model, and their respective properties.

2. Produced Isabelle/HOL specifications matching the mathematical formalizations of the models and their properties.

3. Successfully verified the correctness of the base UTXO model's properties: constant supply, output consumption, and no double spending.

4. Successfully verified the IOTA EUTXO model's properties: the chain constraint for Alias outputs, chain constraint for Foundry outputs, total supply consistency for Foundry outputs.

5. We presented a framework for formalizing distributed ledger technologies in a modular way using the Isabelle/HOL locale feature. Moreover, we have demonstrated a convenient approach to modeling unique identifier uniqueness in Isabelle/HOL using locales.

The results of our research were published at the "Lithuanian MSc Research in Informatics and ICT" conference [DP24].

Conclusions

In this work, we have demonstrated that:

1. The IOTA EUTXO Alias and Foundry output type extensions are sound. This gives confidence in the correctness of the UTXO extensions proposed by IOTA.

2. There is a streamlined way to model the IOTA EUTXO output types by analyzing only subsets of the model. We used this approach to cope the complexity of the Isabelle proofs, which comes from the increasingly large amount of cases that need to be considered when introducing new features.

3. While our work focuses specifically on the IOTA EUTXO model, our modular formalization approach could be used to help verify the correctness of other distributed ledger technologies that are based on the UTXO model.

References

[ABB+18] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 66–77, 2018.

[ABL+18] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of bitcoin transactions. In *International Conference on Financial Cryptography and Data Security*, pp. 541–560. Springer, 2018.

[AFP+11] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simao Melo De Sousa. *Rigorous software development: an introduction to program verification*, vol. 1. Springer, 2011.

[AMN+21] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. Extracting smart contracts tested and verified in coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 105–121, 2021.

[Avi00] Jeremy Avigad. Classical and constructive logic. *Descargado de http://www. andrew. cmu. edu/user/avigad/Teaching/classical. pdf (Class Notes.)*, 2000.

[Bal03] Clemens Ballarin. Locales and locale expressions in isabelle/isar. In *Types for Proofs and Programs*, 2003.

[BBC+97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, et al. The coq proof assistant : reference manual, version 6.1. In 1997.

[BDF+16a] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, et al. Formal verification of smart contracts: short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pp. 91–96, 2016.

[BDF+16b] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, et al. Short paper: formal verification of smart contracts. In *Proceedings of the 11th ACM Workshop on Programming Languages and Analysis for Security (PLAS), in conjunction with ACM CCS*, pp. 91–96, 2016.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda – a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, 2009.

[BG20] Lars Brünjes and Murdoch J Gabbay. Utxo-vs account-based smart contract blockchain programming paradigms. In *International Symposium on Leveraging Applications of Formal Methods*, pp. 73–88. Springer, 2020.

[BKC+17] Woubshet Behutiye, Pertti Karhapää, Dolors Costal, Markku Oivo, and Xavier Franch. Non-functional requirements documentation in agile software development: challenges and solution proposal. In *International conference on product-focused software process improvement*, pp. 515–522. Springer, 2017.

[Bla16] Jasmin Christian Blanchette. My life with an automatic theorem prover. In *Vampire Workshop*, 2016.

[BTG83] Robert Balzer, Cheatham TE Jr, and Cordell Green. Software technology in the 1990's: using a new paradigm. *Computer*, 16(11):39–45, 1983.

[But+14] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[CCM+20a] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. Native custom tokens in the extended UTXO model. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, Lecture Notes in Computer Science, pp. 89–111, Cham. Springer International Publishing, 2020. ISBN: 978-3-030-61467-6. DOI: 10.1007/978-3-030-61467-6_7.

[CCM+20b] Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended utxo model. In *International Conference on Financial Cryptography and Data Security*, pp. 525–539. Springer, 2020.

[CCM+20c] Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, and Joachim Zahnentferner. Utxo: utxo with multi-asset support. In *International Symposium on Leveraging Applications of Formal Methods*, pp. 112–130. Springer, 2020.

[CD16] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2016.2566339. Conference Name: IEEE Access.

[Chl13] Adam Chlipala. Certified programming with dependent types - a pragmatic introduction to the coq proof assistant. In 2013.

[Con+19] John P Conley et al. *Encryption, Hashing, PPK, and Blockchain: A Simple Introduction*. Vanderbilt University, Department of Economics, 2019.

[Cou+07] National Research Council et al. *Software for dependable systems: Sufficient evidence?* National Academies Press, 2007.

[CZ17] Lianhua Chi and Xingquan Zhu. Hashing techniques: a survey and taxonomy. *ACM Computing Surveys (CSUR)*, 50(1):1–36, 2017.

[DMH17] Vikram Dhillon, David Metcalf, and Max Hooper. The dao hacked. In *Blockchain Enabled Applications*, pp. 67–78. Springer, 2017.

[dMKA+15] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover. In 2015.

[dMU21] Leonardo Mendonça de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *CADE*, 2021.

[DP24] Edvardas Dlugauskas and Karolis Petrauskas. Formalizing IOTA Extended UTXO in Isabelle. *Vilnius University Open Series*:26–35, 2024-05. DOI: 10.15388/LMITT.2024.3. URL: https://www.journals.vu.lt/open-series/article/view/35357.

[DPN+18] Sergi Delgado-Segura, Cristina Pérez-Sola, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartí. Analysis of the bitcoin utxo set. In *International Conference on Financial Cryptography and Data Security*, pp. 78–91. Springer, 2018.

[Flo93] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pp. 65–81. Springer, 1993.

[FLP85] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[Gab22] Murdoch J Gabbay. Algebras of utxo blockchains. *Mathematical Structures in Computer Science*:1–56, 2022.

[HBM20] Joseph Herkert, Jason Borenstein, and Keith Miller. The boeing 737 max: lessons for engineering ethics. *Science and engineering ethics*, 26(6):2957–2974, 2020.

[HM05] Tony Hoare and Jay Misra. Verified software: theories, tools, experiments vision of a grand challenge project. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pp. 1–18. Springer, 2005.

[HR17] Garrick Hileman and Michel Rauchs. 2017 global cryptocurrency benchmarking study. *SSRN Electronic Journal*, 2017. ISSN: 1556-5068. DOI: 10.2139/ssrn.2965436. URL: http://www.ssrn.com/abstract=2965436.

[IO05] MG Ilieva and Olga Ormandjieva. Automatic transition of natural language software requirements specification into formal presentation. In *International Conference on Application of Natural Language to Information Systems*, pp. 392–397. Springer, 2005.

[KKU13] Muhammad Naeem Ahmed Khan, Muhammad Khalid, and Sami UlHaq. Review of requirements management issues in software development. *International Journal of Modern Education & Computer Science*, 5(1):21–27, 2013.

[Kon] Igor Konnov. Specification of tendermint in tla. GitHub. URL: https://github.com/informalsystems/verification.

[KR18] Johannes Krupp and Christian Rossow. Teether: gnawing at ethereum to automatically exploit smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 1317–1333, 2018.

[KT18] Max J. Krause and Thabet Tolaymat. Quantification of energy and carbon costs for mining cryptocurrencies. *Nature Sustainability*, 1(11):711–718, 2018-11. ISSN: 2398-9629. DOI: 10.1038/s41893-018-0152-7. URL: http://www.nature.com/articles/s41893-018-0152-7.

[LFL19] Yi-Chen Liu, Jingchao Fang, and Jia-Wei Liang. Account-wise ledger: a new design of decentralized system, 2019. URL: https://github.com/ECS-251-W2020/final-project-triple-l-group/blob/master/Thesis/Account-Wise%20Ledger.pdf.

[MA19] Yvonne Murray and David A Anisi. Survey of formal verification methods for smart contracts on blockchain. In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–6. IEEE, 2019.

[MDH+20] Shunli Ma, Yi Deng, Debiao He, Jiang Zhang, and Xiang Xie. An efficient nizk scheme for privacy-preserving transactions over account-model blockchain. *IEEE Transactions on Dependable and Secure Computing*, 18(2):641–651, 2020.

[Mel19] Orestis Melkonian. *Formalizing Extended UTxO and BitML Calculus in Agda*. MA thesis, 2019. URL: https://omelkonian.github.io/data/msc-thesis.pdf.

[MPP+22] Sebastian Müller, Andreas Penzkofer, Nikita Polyanskii, Jonas Theis, William Sanders, and Hans Moog. Reality-based utxo ledger. *arXiv preprint arXiv:2205.01345*, 2022.

[MRV20] Leonid Al'bertovich Merkin-Janson, Ruslan Maratovich Rezin, and Nikolay Konstantinovich Vasilyev. Architecture of the formally-verified distributed ledger system innochain. *Modeling and Analysis of Information Systems*, 27(4):472–487, 2020.

[Nor09] Ulf Norell. Dependently typed programming in agda. In *ACM SIGPLAN International Workshop on Types In Languages Design And Implementation*, 2009.

[NPD18] Zeinab Nehai, Pierre-Yves Piriou, and Frederic 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)*, pp. 980–987. IEEE, 2018.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, vol. 2283. Springer Science & Business Media, 2002.

[Pap21] Levente Pap. IOTA protocol RFCs, 2021-11-23. URL: https://github.com/lzpap/protocol-rfcs/blob/fab001c2834752f86bdbed15d6af607244a23f33/text/0038-output-types-for-tokenization-and-sc/0038-output-types-for-tokenization-and-sc.md. original-date: 2021-02-17T09:16:09Z.

[Pel19] Doron A Peled. Formal methods. In *Handbook of Software Engineering*, pp. 193–222. Springer, 2019.

[PL19] Serguei Popov and Q Lu. Iota: feeless and free. *IEEE Blockchain Technical Briefs*, 2019.

[PMM+18] Deepak Puthal, Nisha Malik, Saraju P Mohanty, Elias Kougianos, and Gautam Das. Everything you wanted to know about the blockchain: its promise, components, processes, and problems. *IEEE Consumer Electronics Magazine*, 7(4):6–14, 2018.

[Pop] Serguei Popov. The tangle.

[Ren19] Ling Ren. Analysis of nakamoto consensus. *Cryptology ePrint Archive*, 2019.

[Rup10] Nayan B Ruparelia. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3):8–13, 2010.

[Rus07] John M Rushby. Automated formal methods enter the mainstream. *J. Univers. Comput. Sci.*, 13(5):650–660, 2007.

[Sas86] James T Sasaki. Extracting efficient code from constructive proofs. Tech. rep., Cornell University, 1986.

[Sch02] Rolf Schwitter. English as a formal specification language. In *Proceedings. 13th International Workshop on Database and Expert Systems Applications*, pp. 228–232. IEEE, 2002.

[Tav21] Hanneli C. A. Tavante. A data-centered user study for proof assistant tools. In *Annual Workshop of the Psychology of Programming Interest Group*, 2021.

[Vil21] Jørgen Villadsen. Teaching automated reasoning and formally verified functional programming in agda and isabelle / hol. In 2021.

[Win90] Jeannette M Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–22, 1990.

[Woo+14] Gavin Wood et al. Ethereum: a secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[WOY+19] Shuai Wang, Liwei Ouyang, Yong Yuan, Xiaochun Ni, Xuan Han, and Fei-Yue Wang. Blockchain-enabled smart contracts: architecture, applications, and future trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(11):2266–2277, 2019.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced research working conference on correct hardware design and verification methods*, pp. 54–66. Springer, 1999.

[YMR+19] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. Blockchain technology overview. *arXiv preprint arXiv:1906.11078*, 2019.

[Zah18a] Joachim Zahnentferner. An abstract model of utxo-based cryptocurrencies with scripts. *Cryptology ePrint Archive*, 2018.

[Zah18b] Joachim Zahnentferner. Chimeric ledgers: translating and unifying utxo-based and account-based cryptocurrencies. *IACR Cryptol. ePrint Arch.*, 2018:262, 2018.

[ZTC+21] Jinnan Zhang, Rui Tian, Yanghua Cao, Xueguang Yuan, Zefeng Yu, Xin Yan, and Xia Zhang. A hybrid model for central bank digital currency based on blockchain. *IEEE Access*, 9:53589–53601, 2021.

[ZXD+17] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: architecture, consensus, and future trends. In *2017 IEEE international congress on big data (BigData congress)*, pp. 557–564. Ieee, 2017.

Appendices

A. Isabelle/HOL Codebase Walkthrough

The Isabelle/HOL codebase accompanying this thesis is available in GitHub[3]. This appendix provides an overview of the repository's structure and the purpose of each file.

The codebase is organized into three main directories: *abstract*, *implementation*, and *shared*.

The *abstract* directory contains the abstract definitions of the UTXO ledgers:

- *AbstractBasicUtxoLedger.thy*: Contains the abstract definition of a basic UTXO ledger.
- *AbstractIotaAliasUtxoLedger.thy*: Contains the abstract definition of an IOTA Alias UTXO ledger.
- *AbstractIotaFoundryUtxoLedger.thy*: Contains the abstract definition of an IOTA Foundry UTXO ledger.

The *implementation* directory contains the concrete implementations of the UTXO ledgers and their properties:

- *BasicUtxoLedger.thy*: Contains the concrete implementation of a basic UTXO ledger.
- *BasicUtxoLedgerProperties.thy*: Contains the essential properties of the basic UTXO ledger.
- *IotaUtxoLedger.thy*: Contains the concrete implementation of an IOTA UTXO ledger.
- *IotaUtxoLedgerAlias.thy*: Extends *IotaUtxoLedger.thy* and contains the concrete implementation of an IOTA Alias UTXO ledger.
- *IotaUtxoLedgerFoundry.thy*: Extends *IotaUtxoLedger.thy* and contains the concrete implementation of an IOTA Foundry UTXO ledger.
- *IotaUtxoLedgerProperties.thy*: Contains the essential properties of the IOTA UTXO ledgers.

The *shared* directory contains shared definitions and utilities used across the project:

- *FiniteNatSet.thy*: Contains helper lemmas used throughout the other files.
- *Hash.thy*: Contains the definition of a unique hash identifier.

To explore the codebase, we recommend to start by examining the base UTXO model specification in *BasicUtxoLedger.thy* and its properties in *BasicUtxoLedgerProperties.thy*. For the IOTA EUTXO model specification, refer to *IotaUtxoLedger.thy* and *IotaUtxoLedgerProperties.thy*.

[3]https://github.com/EdvardasDlugauskas/iota-eutxo-isabelle