



HAL
open science

Reality-based UTXO Ledger

Sebastian Müller, Andreas Penzkofer, Nikita Polyanskii, Jonas Theis, William Sanders, Hans Moog

► **To cite this version:**

Sebastian Müller, Andreas Penzkofer, Nikita Polyanskii, Jonas Theis, William Sanders, et al.. Reality-based UTXO Ledger. 2025. hal-04943527

HAL Id: hal-04943527

<https://hal.science/hal-04943527v1>

Preprint submitted on 12 Feb 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Reality-based UTXO Ledger

Sebastian Müller¹, Andreas Penzkofer², Nikita Polyanskii², Jonas Theis², William Sanders²,
and Hans Moog²

¹Aix Marseille Université, CNRS, Centrale Marseille, I2M - UMR 7373, 13453 Marseille, France,
sebastian.muller@univ-amu.fr

²IOTA Foundation, 10405 Berlin, Germany, research@iota.org

August 8, 2023

Abstract

The Unspent Transaction Output (UTXO) model is commonly used in the field of Distributed Ledger Technology (DLT) to transfer value between participants. One of its advantages is that it allows parallel processing of transactions, as independent transactions can be added in any order. This property of order invariance and parallelisability has potential benefits in terms of scalability. However, since the UTXO Ledger is an append-only data structure, this advantage is compromised through the presence of conflicting transactions. We propose an extended UTXO Ledger model that optimistically updates the ledger and keeps track of the dependencies of the possible conflicts. In the presence of a conflict resolution mechanism, we propose a method to reduce the extended ledger back to a consistent UTXO Ledger.

1 Introduction

The Unspent Transaction Output (UTXO) model is a design common to many cryptocurrencies, including Bitcoin [27] and many of its derivatives, Cardano [6], and IOTA [30]. In the UTXO model, transactions specify the outputs of previous transactions as inputs and create new outputs spending the inputs. Thus, a transaction consists of a list of inputs and a list of outputs. The outputs are associated to users' addresses by certain unlock conditions; in general, an account "possesses" a private key and addresses that allow to spend and receive UTXOs. Accounts then track their balance by maintaining a list of the received (unspent) outputs. This model differs from the account-based model used in most smart-contract-based cryptocurrencies, for example, Ethereum [5]. The latter represents assets as balances within accounts, and transactions describe how these balances change, see Figure 3. A comparison of blockchain data-models can be for example found in [11] and we refer to [4] for a detailed discussion on these two models.

A conceptual difference is that the account-based model updates user balances globally, while the UTXO model only records transaction receipts. This construction allows transactions to be

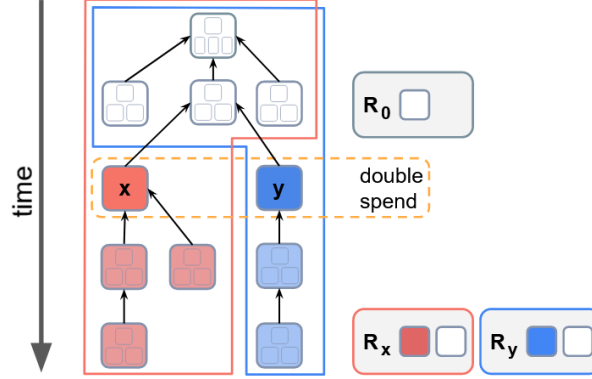


Figure 1: Reality-based Ledger: the double-spending transactions x and y create two realities, R_x and R_y . Each reality yields a valid ledger state.

processed in parallel, leading to performance benefits and possible scalability advantages. A notable difference is that account-based models need a total ordering to resolve conflicts, while in the UTXO model, a total ordering is not necessary. There is, however, a limiting effect on scalability in the presence of conflicting transactions. Here, conflicts are transactions that spent the same output. As the UTXO Ledger is an append-only structure, the conflicts must be sorted out before changes are made. This is currently done with the help of consensus protocols that are based on a (unique) leader or other mechanisms to create a total ordering of the transactions.

We propose a solution to this “bottleneck” problem that allows leaderless conflict resolution that does not require total ordering. To this end we define an augmented data structure, the *Reality-based Ledger*, which processes transactions optimistically and manages possible conflicts until they are resolved. As a consequence, we can update the ledger on the arrival of the transactions. This contrasts with a blockchain system where transactions can only be processed once they are included in a batch or block. In further work, [26], this feature is used to design a stream process-oriented DLT.

1.1 Results

A UTXO Ledger induces a partial order on its transactions and thus can be seen as a partially ordered set (poset). Posets are in a one-to-one correspondence¹ to directed acyclic graphs (DAGs). Typically, conflicts are excluded from an accepted state in such a ledger, however, this requires that participants reach a consensus on which transactions to add to the ledger. This selection is usually performed by choosing a “leader” among the participants, and only this leader can add transactions to the ledger. These transactions are added in batches that are called *blocks*. This leader is a “centralized” bottleneck that hinders scalability and annihilates most of the scalability advantages

¹The correspondence is in fact not one-to-one in the strict sense, but is in a weaker sense involving equivalence relations, see Section 3.

of the UTXO model. Some of these limitations remain even for leaderless consensus protocols that are based on total ordering, as the total ordering requires a “complete” or “non-sharded” view of the set of transactions.

We define the *Reality-based Ledger* as an augmented data structure of the conflict-free UTXO Ledger, which may contain conflicting transactions. This Reality-based Ledger can contain many different *realities*, where each of the realities corresponds to a conflict-free UTXO Ledger, see Figure 1 for an illustration. We show that this data structure does not depend on the order of the incoming transactions and forms therefore an eventual consistent distributed data structure. We propose an additional data structure, called the Branch DAG, to manage the dependencies of the different realities. We provide several algorithms for the efficient management and update of the above graph data structures. Notably, these algorithms update the data structures on the arrival of new data, thus minimizing the delay between the issuance of transactions and addition to the distributed ledger and enabling a higher degree of stream processing in the underlying DLT.

To obtain the most recent valid ledger state, e.g., to determine the current balance for an account owner, we provide algorithms that extract a reality, based on a weight function that is imposed on the branches. Finally, we prove that the augmented data structure can be pruned back into a conflict-free UTXO Ledger in the presence of a conflict resolution mechanism.

1.2 Related Work

The benefits and drawbacks of the UTXO model have been discussed extensively, and several extensions of the UTXO model have been proposed. We refer to [4] for an excellent overview. However, we want to note that most of the variants concern either the extension of the UTXO model itself, e.g., [6], are proposed to increase their applicability towards smart contracts, or are designed to combine the UTXO and the account-based model. In this paper, we address an extension of the UTXO model to track dependencies of the outputs to allow parallel processing of the transactions.

To allow such a parallel processing of transactions the first step is to move from the traditional blockchain structure of the ledger to a more general DAG-based structure. There are several approaches to improve performance by circumventing the linear chain structure. Most of them have in common that blocks can reference not only one previous block but also more than one, changing the underlying data structure from a chain to a directed acyclic graph (DAG). This natural idea of using DAGs has become quite popular in the last decade and led to higher throughput and in some cases to similar confirmation latency, e.g. [36, 28, 19, 20, 29, 33, 35, 2, 14, 23, 41, 8, 16, 18, 34] and the survey paper [39].

However, parallel writing is only a necessary requirement for efficient parallel processing of transactions but does not yet address the question of conflict resolution and eventual execution of the transactions. The various DAG protocols, indeed, differ significantly in how conflicts are resolved and transactions are executed. Let us consider these two aspects separately.

1.2.1 Conflict resolution

The various DAG protocols differ significantly in the form of how consensus is achieved. In particular, the utilisation of a DAG data structure does enable but does not require circumventing a total ordering of transactions. For example, in [29] nodes follow and attach to the heaviest DAG, while in most other proposed protocols, e.g. [35, 20, 14, 23, 41, 2, 17, 16] consensus is still achieved by constructing a total ordering over the set of transactions. A popular approach that uses the ordering of transactions to achieve consensus is via atomic broadcast protocols. Such protocols allow the network participants to reach a consensus on a (total) ordering of the received transactions, and this linearised output forms then the ledger, e.g., see [24, 13]. Improvements of these broadcast protocols are proposed, for example, in Hashgraph [3] and Aleph [14] and more recently in Narwhal [8] based on the encoding of the “communication history” in the form of a DAG. The protocols in question serve to alleviate the data dissemination bottleneck of the traditional Nakamoto consensus by decoupling the data dissemination process from the consensus determination procedure. Notable advancements have been made in the realm of consensus determination on top of DAG-based memory pools, as demonstrated in the works of DAG Rider [17] and Bullshark [16]. A more comprehensive and abstract examination of these protocols can be found in [31], which provides a description from a broader perspective. There are common points with the approach of IOTA 2.0 [26]. A DAG structure serves as a “testimony” of the communication among the nodes, and new blocks are used for (implicit) voting on previous blocks. This block structure “maps down” to a dependency structure of the contained transactions and UTXOs and the dependencies of the UTXO and the conflict resolution are covered by our model.

We also want to mention a prominent approach proposed by Prism [2]. This methodology outlines the explicit differentiation of the functions of blocks into three distinct categories: proposer blocks, transaction blocks, and voter blocks. The separation of transaction blocks enables participants to initiate transactions, eliminating the requirement for a memory pool. The three categories of blocks assemble into a structured Directed Acyclic Graph (DAG) that facilitates an efficient means of voting on “leader blocks,” resulting in consensus through total ordering. Again, the dependency structure and conflict resolution, in the setting of UTXO model, can be described with the model in our paper.

Total-ordering is, however, not necessary to achieve consensus. This approach is pursued by [26]. Similar to the DAG-based memory pools a causal dependency of the blocks is used to determine confirmation of the contained transactions. This approach, however, requires an active tracking of the dependencies and a certain “confirmation weight”. Our results on tracking of conflicts and their resolution is an important ingredient for the protocol proposed in [26], but the concepts are natural for an optimistic execution and *a posteriori* conflict resolution.

1.2.2 Execution and view change

Parallel booking and execution of transactions is essential for high throughput and scalability. There are three existing proposals for achieving this, each of which is contingent upon the method of consensus attainment. In the case of total-ordering, parallel execution of transactions following consensus

can be accomplished. For instance, the implementation of Prism [2] in [40] employed a scoreboard technique to parallelize the execution of ordered UTXO transactions. It was demonstrated in [38] that Prism can support smart contract platforms, with the execution of smart contracts rather than consensus serving as the bottleneck in their implementation.

Another possibility building on a total-ordering is the optimistic booking of the transactions and a *a posteriori* conflict resolution via total ordering. For instance, this is the direction of a current Leios proposal in Cardano [7] and in Sui [25].

Our contribution is not limited to merely parallel booking but encompasses the comprehensive mechanism for processing new transactions and instantaneously updating the ledger state as perceived by the majority of network participants. Our approach actively constructs a DAG, referred to as the Ledger DAG, which encodes the dependencies between transactions. This DAG is generated prior to consensus and facilitates the tracking of dependencies between pending or conflicting transactions. This natural idea has been explored in various academic papers. It does rely on the construction of a dependency graph that takes the form of a DAG and encodes the causal dependencies of the transactions, e.g. [10, 1], or on a pre-ordering, [15]. These works focus on the execution of smart contracts in an account-based model, i.e. with no local states, while our work covers the situations of local state transactions as in the UTXO-model. Finally, let us note that our approach does not rely on total-ordering; however, it still supports it, thus eliminating the dependence on the linear structure of total-ordering for view changes and transaction confirmation.

In conclusion, it is important to highlight that the preceding discussion does not aim to deliver a comprehensive depiction of the current state of all Directed Acyclic Graph (DAG)-based protocols and their multifaceted designs. The intention is not to exhaustively cover every aspect of these protocols, but rather to offer an overview of the aspects that relate closely to our work. We encourage the readers to delve into the referenced works for a more detailed understanding and to explore further literature for additional perspectives and developments that may not be included in this section.

1.2.3 Beyond DLTs

From a general point of view, our approach is natural and was already been successfully applied in various settings. It relies on constructing systems that allow working effectively and efficiently with inconsistent information and where “context switching” is a low-cost operation; e.g. [9]. We also want to note that this approach can be found in the field of belief revision.

Finally, we want to draw some connections with the field of replicated invariant data types. The UTXO model, if distributed on multiple nodes, falls into the class of a replicated data types as the nodes can make concurrently changes to the data. Its append-only design and the invariant constraints render the operation of adding a transaction not commutable. This means that the order of the incoming transaction matters for the construction of the ledger. However, if there is a deterministic rule on how to process conflicts or a “leader” that pre-filters the transactions, this data structure can be turned into a conflict-free replicated data type (CRDT), e.g., see [32]. The proposed Reality-based Ledger is already a CRDT without the need of a consensus mechanism since

all operations on the involved data types are commutative.

In the context of replicated data structure, some works increase the performances in “augmenting” the data structure and distinguishing between different notions of consistency, e.g., [22, 21]. From a conceptual idea, this resembles our approach. However, the concrete proposals are distinct due to different assumptions on the communication model and field of applications. To our knowledge, our method is the first that allows a leaderless “ex-post” conflict resolution on these types of models.

1.3 Structure of the Paper

The document is structured as follows. In Section 2 we provide an introduction to a standard conflict-free UTXO Ledger. In Section 3 we give an overview of some of the graph theoretical preliminaries used in this paper. In Section 4 we introduce the concept of a Reality-based Ledger and additional data structures that provide the necessary tools to manage this novel type of ledger. In Section 5 some of the core operations to maintain and access the ledger are presented.

We employ several graph structures to efficiently manage the Reality-based Ledger. Table 1 gives an overview of the utilised graphs.

2 Conflict-Free UTXO Ledger

In the standard UTXO model, transactions specify the outputs of previous transactions as inputs and create new outputs spending (or consuming) the inputs. Thus, a transaction consists of a list of inputs and a list of unique outputs. To every output, we associate a unique reference or output ID. Typically such an output ID is created with the involvement of a hash function.

Remark 2.1. *A collision-resistant hash function is used to map data of arbitrary size to a fixed-size binary sequence, i.e., $\text{hash} : \{0, 1\}^* \rightarrow \{0, 1\}^h$. Moreover, it is required that it is practicably impossible to find for a given sequence x another sequence x' such that $\text{hash}(x) = \text{hash}(x')$. Throughout the remainder of the paper, we assume that a particular hash function is fixed and used by all participants.*

For example, the output ID could be created through the concatenation of the index of the output within the transaction and the hash of the transaction’s content. Every output represents a

Graph	Vertices	Edges
UTXO DAG	in-, outputs, transaction IDs	spending relations
Ledger DAG	transactions	spending relations
Conflict DAG	conflicts	conflict dependencies
Conflict Graph	conflicts	conflict relations
Branch DAG	branches	branch dependencies

Table 1: Overview of the graphs used in this paper.

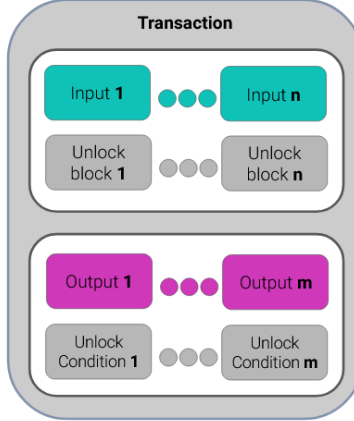


Figure 2: Simplified transaction layout. The fund owner signs the transaction in the unlock blocks. Inputs are consumed and new outputs are created. The new outputs can be spent once the unlock conditions are satisfied.

specific amount of the underlying cryptocurrency. The value of all inputs, i.e., spent outputs, must equal the value of all outputs of a transaction. Every output can be spent only once and, hence, value is conserved overall. With each output comes an unlock condition, which declares by whom and under which conditions it can be spent. With each input comes an unlock block containing a proof that the transaction issuer is allowed to spend the inputs and fulfills the unlock condition, e.g., a signature proving ownership of a given input’s address. We refer to Figure 2 for a general transaction layout. In Section 4.5 we propose a more general description of this model.

In this model, an account that is controlled by an entity holding the corresponding private / public key pair, is a collection of UTXOs that can be unlocked through the key pair. This type of book keeping of balances and transactions differs fundamentally from the account-based model, such as it is used in Ethereum [5]. In the account-based model funds are represented as balances within accounts and transactions describe how these balances change, see Figure 3.

Let us define the UTXO Ledger model more formally. We follow the approach of [12]. Note that for the purpose of this work we consider a simplification of the UTXO models used in practice.

Definition 2.1 (Output and input). An *output* is a pair of a value $v \in \mathbb{R}^+$ and an unlock condition cond. We write $o = (v, \text{cond})$ to denote the output. An *input* i is a reference to an output. In such a case, we say the input consumes the output.

Definition 2.2 (Transaction). A transaction x is a collection of inputs $\text{in}(x)$, outputs $\text{out}(x)$, and an unlock data $\text{unlock}(x)$:

1. $\text{in}(x) = (i_1, \dots, i_n)$ is a list of inputs, i.e., references to outputs. We say that those outputs are *consumed* or *spent* by transaction x .
2. $\text{out}(x) = (o_1, \dots, o_m)$ is a list of new outputs produced by transaction x .

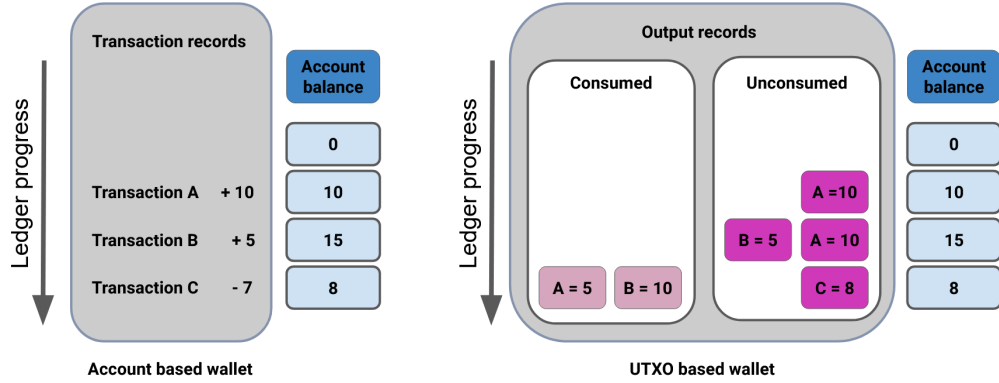


Figure 3: Comparison between account- and UTXO-based model.

3. $\text{unlock}(x)$ is a data which unlocks the inputs. This is usually done by cryptographic proof of authorization that ensures that the issuer of the transaction satisfies the condition cond of the consumed outputs.

Definition 2.3 (Ledger). The *ledger* is a set of transactions and denoted as \mathcal{L} .

Definition 2.4 (Ledger state). The *ledger state*, written as $\text{state}(\mathcal{L})$, is the set of all outputs that are not consumed by a transaction in the ledger \mathcal{L} . In other words, the ledger state is the set of outputs, for which no input exists that references them.

The ledger progresses through the addition of new transactions. Furthermore, it is an append-only structure, i.e., transactions can only be added and not removed from the ledger.

In a distributed system, the append-only nature of this data structure makes it necessary that the operators of the distributed ledger have consensus on which transactions should be added to the ledger. In a blockchain setting, such as Bitcoin, this can be achieved by the selection of a leader who typically extends the longest chain. The longest chain then determines which transactions are included in the ledger and in which order.

To provide consistency there can be specific *ledger constraints* which ought to be fulfilled before a certain transaction x can be added to the ledger \mathcal{L} . The ledger constraints are generally enforced by transaction validation rules applied prior to addition of the transaction to the ledger. We define $\text{constr}(\mathcal{L}, x) = 1$ if all imposed constraints are satisfied by transaction x and $\text{constr}(\mathcal{L}, x) = 0$ otherwise. The constraints of a transaction being added to the ledger adhere typically to the following assumption:

Assumption 2.1 (Ledger constraints). A transaction x is added to the ledger \mathcal{L} if it follows the following rules:

1. the transaction x is syntactically correct;
2. the sum of values of $\text{in}(x)$ equals the sum of values of $\text{out}(x)$;

3. the unlocking data $\text{unlock}(x)$ is valid;
4. $\text{in}(x)$ are references to existing unspent (not yet spent) outputs in the ledger state $\text{state}(\mathcal{L})$;

Definition 2.5 (Consistent Ledger). We say that the ledger \mathcal{L} is *consistent* if and only if $\text{constr}(\mathcal{L}, x) = 1$ for all $x \in \mathcal{L}$.

The UTXO Ledger starts at the so-called *genesis* ρ , i.e., the transaction that is the ultimate predecessor of any transaction of the UTXO Ledger. The genesis transaction only contains outputs and no inputs. Each new transaction is an atomic update of the ledger and the ledger state. A transaction x is added to a consistent ledger \mathcal{L} if $\mathcal{L} \cup \{x\}$ is a consistent ledger or, equivalently, if $\text{constr}(\mathcal{L}, x) = 1$. The above constraints or consistency rules imply that each output can be consumed by at most one transaction and thus a consistent ledger can not contain a so-called *double spend*.

An important property of the UTXO Ledger is that the validity of the state update (adding a new transaction) can be determined by only using the context of the transaction itself, i.e., inputs, outputs, and unlock conditions. This allows a certain degree of parallelism and turns the UTXO Ledger into a partially ordered data structure.

Remark 2.2. We can talk about a ledger invariant $\mathbf{Invar}(\mathcal{L})$ which is preserved by each addition of a transaction to the ledger. In other words, $\mathbf{Invar}(\mathcal{L}) = \mathbf{Invar}(\mathcal{L}')$ for any two consistent ledgers \mathcal{L} and \mathcal{L}' . The prime example is that the sum of the values of all unspent outputs in the ledger state remains constant.

3 Graph Theoretical Preliminaries

In this section, we summarize basic graph theoretical notations and results that are used in the remaining part of the paper.

The set of integers between 1 and m is denoted by $[m]$. A *graph* G is a pair (V, E) , where V denotes the set of vertices and E denotes the set of edges. A graph is called *directed* if every edge has its direction, e.g., for an edge (u, v) , the direction goes from u to v .

Definition 3.1 (DAG). A *directed acyclic graph (DAG)* is a directed graph with no directed cycles, i.e., by following the directions of edges, we never form a closed loop.

A vertex v in a graph $G = (V, E)$ is called *adjacent* to a vertex u if $(u, v) \in E$. An edge $e \in E$ is said to be adjacent to a vertex $v \in V$ if e contains v . The *out-degree* and *in-degree* of a vertex v in a directed graph $G = (V, E)$ is the number of adjacent edges of the form (v, u) and, respectively, (u, v) . A vertex in a graph is called *isolated* if there is no edge adjacent to it.

Definition 3.2 (Neighbours in a graph). Let $G = (V, E)$ be a graph. For a vertex $v \in V$, define the *set of neighbours* (or G -neighbours), written as $N_V(v)$ ², to be the vertices adjacent to v .

²In the remainder of the paper, we will often identify the graph with its vertex set, since for a given set of vertices V , we will have only one DAG $D = (V, E)$. Thereby, the set of neighbours $N_V(v)$ and other concepts that use V as a subscript will be clear from the context.

Definition 3.3 (Parents, children and leaves in a DAG). Let $D = (V, E)$ be a DAG. For a vertex $v \in V$, define the set of *parents*, written as $\text{par}_V(v)$, to be the set of vertices $u \in V$ such that $(v, u) \in E$. Similarly, we define the set of *children*, written as $\text{child}_V(v)$, to be the set of vertices $u \in V$ such that $(u, v) \in E$. A vertex $v \in V$ with in-degree zero is called a *leaf*.

Definition 3.4 (Partial order induced by a DAG). Let $D = (V, E)$ be a DAG. We write $u \leq_V v$ for some $u, v \in V$ if and only if there exists a directed path from u to v , i.e., there are some vertices $w_0 = u, w_1, \dots, w_{s-1}, w_s = v$ such that $(w_{i-1}, w_i) \in E$ for all $i \in [s]$. Furthermore, we write $u <_V v$ if $u \leq_V v$ and $u \neq v$.

Note there could be different DAGs producing the same partial order. The DAG with the fewest number of edges that gives the partial order \leq_V is usually called the *transitive reduction* of D or the *Hasse diagram* of \leq_V . In the following definition, we give a more general definition of the minimal subDAG of $D = (V, E)$ induced by a set of vertices $S \subseteq V$ which coincides with the transitive reduction of D when $S = V$.

Definition 3.5 (Minimal subDAG induced by a set of vertices). Let $D = (V, E)$ be a DAG. For a subset of vertices $S \subseteq V$, we define the *minimal subDAG* of D induced by S to be the DAG $D' = (V', E')$ whose vertex set is $V' = S$ and there is an edge $(v, u) \in E'$ if and only if $u, v \in S$, $v <_V u$ and there is no $w \in S \setminus \{u, v\}$ such that $v <_V w <_V u$.

Definition 3.6 (Maximal and minimal elements). Let $D = (V, E)$ be a DAG and let \leq_V be the partial order induced by D . For a subset of vertices $S \subseteq V$, an element $u \in S$ is called *D-maximal* (*D-minimal*) in S if there is no $v \in S \setminus \{u\}$ such that $u \leq_V v$ ($v \leq_V u$). Define $\text{max}_V(S)$ and $\text{min}_V(S)$ to be the set of *D-maximal* and, respectively, *D-minimal* elements in S .

Remark 3.1. *The maximal (minimal) elements of a DAG D are also called the geneses (tips) of D . Usually, we consider DAGs with only one genesis, whereas the number of tips can be large.*

Definition 3.7 (Future and past cones). Let $D = (V, E)$ be a DAG. For $x \in V$, define the *past cone* of x in D , written as $\text{cone}_V^{(p)}(x)$ to be the set of all vertices $y \in V$ such that $x \leq_V y$. Similarly, define the *future cone* of x in D , written as $\text{cone}_V^{(f)}(x)$ to be the set of all vertices $y \in V$ such that $y \leq_V x$.

Definition 3.8 (Future-closed and past-closed sets). Let $D = (V, E)$ be a DAG. A subset $S \subset V$ is called *D-past-closed* if and only if for every $u \in S$, the past cone $\text{cone}_V^{(p)}(u)$ is contained in S . Similarly, a subset $S \subset V$ is called *D-future-closed* if and only if for every $u \in S$, the future cone $\text{cone}_V^{(f)}(u)$ is contained in S .

We conclude with a definition of maximal independent sets for general graphs.

Definition 3.9 (Maximal Independent Set). Let $G = (V, E)$ be a finite graph. A subset $S \subset V$ is an *independent set* if and only if for every two vertices $u, v \in S$ there is no edge connecting the two, i.e., $(u, v) \notin E$. An independent set S is called a *maximal independent set* if and only if there is no other independent set S' such that $S \subsetneq S'$.

4 Reality-based Ledger

In Section 2 we described the model of a conflict-free UTXO Ledger that is suitable for an environment where transactions are pre-filtered by a consensus mechanism. Since that ledger was conflict-free a valid ledger state could be readily extracted, see Definition 2.4.

To alleviate the restriction of requiring a conflict-free data structure, we propose an augmented version of the standard conflict-free UTXO Ledger model that allows more than one output spend. The ledger \mathcal{L} continues to be defined by Definition 2.3, however, the total set of transactions must not be conflict-free. We will derive a concept, called a *reality*, which allows to reduce \mathcal{L} to a subset of transactions that yield a valid (*Reality-based*) Ledger state, see also Definition 2.4. We refer to Figure 4 for an overview on the dependencies of the principal definition that are required to describe the reality-based ledger and to Figure 5 for an overview of the used notations.

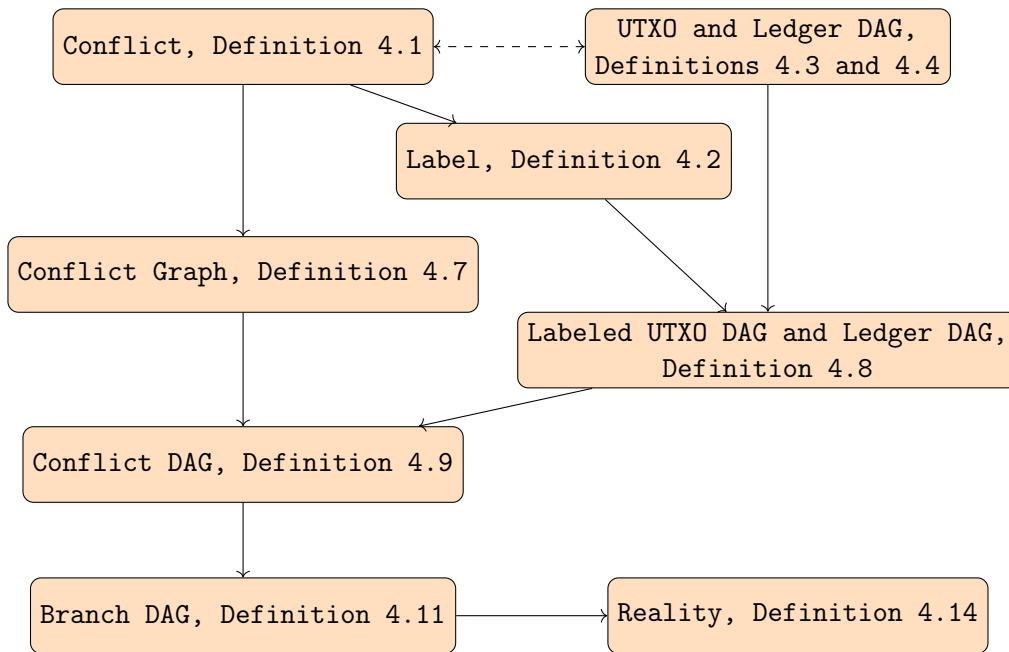


Figure 4: Dependencies of the main definitions around the reality-based ledger.

The purpose of the augmentation is to identify and track possible conflicting transactions. We add this information to each transaction and output. For a given transaction we may set an additional flag or *label* label.

Let us highlight that the transaction layout does not change and label is only assigned a value if needed. We can therefore think of an output as a triplet of a value v , an unlock condition cond , and a label label , i.e., $o = (v, \text{cond}, \text{label})$. Let us now define what we mean by conflicts.

Definition 4.1 (Conflicts). A transaction $x \in \mathcal{L}$ is called a *conflict* if and only if there exists a

transaction $y \in \mathcal{L} \setminus \{x\}$ such that x and y contain at least one same input. The set of all conflicts is denoted by \mathcal{C} and dubbed the *conflict set* of the ledger \mathcal{L} .

We can now define how we set the label label:

1. if a transaction x is not a conflict, the label is not set;
2. otherwise the label is set to a generic unique reference to the transaction, e.g., the transaction ID.

More formally we define the label as follows.

Definition 4.2 (Label). We define \perp to be the label of the genesis ρ . Let \mathcal{Y} be a label space such that $\perp \in \mathcal{Y}$, and $\text{label} : \mathcal{L} \rightarrow \mathcal{Y}$ be a function with the following properties:

1. if $x \in \mathcal{L} \setminus \{\mathcal{C}\}$, then $\text{label}(x) = \perp$;
2. the restriction of the function label on the set $\mathcal{C} \cup \{\rho\}$ is injective, i.e., the image $\text{label}(\mathcal{C} \cup \{\rho\}) := \{\text{label}(x) : x \in \mathcal{C} \cup \{\rho\}\}$ has size $|\mathcal{C}| + 1$.

Remark 4.1. *One natural choice to set a unique (with high probability) label function is to utilize a hash function $\text{hash}(\cdot) : \{0, 1\}^* \rightarrow \{0, 1\}^h$ (cf. Remark 2.1) for large enough h . Then the label set \mathcal{Y} is $\{0, 1\}^h \cup \perp$.*

Remark 4.2. *As two conflicting transactions x and y may be perceived at different times, the detection of a conflict can only be achieved after having received both transactions. If transaction x is perceived first, we do not see it yet as a conflict and do not set any flag. Only, when transaction y arrives, we can identify x as a conflict and set both flags for x and y .*

In the presence of this label, we can remove the forth constraint from the ledger, see Assumption 2.1. However, to avoid the data structure becoming “meaningless” we still need a notion of “consistency” as we will see in Assumption 4.1, and a mechanism to track the dependencies of the conflicts.

The next part of this section defines several data structures that can be derived from the UTXO inter-dependencies. These structures are used to track conflicting transactions without the need for consensus. More precisely, in Section 4.1 we will explain how the UTXO transactions and their in- and outputs result in a DAG structure. In Section 4.2 we present how we can use the UTXO data structure to manage the conflicting transactions efficiently.

In Section 4.3 the information contained in the UTXO DAG is split into the Conflict Graph, which keeps track of the conflicting transactions, and the Conflict DAG, which describes the inherited dependencies of conflicts. Finally, in Section 4.4 branches are introduced, which form a possible non-conflicting state of the ledger. Combining non-conflicting branches can create maximally independent sets of conflicts called realities. Each reality can be associated to a consistent ledger, see Theorem 4.1.

Set symbols

\mathcal{C} set of conflicts
 \mathcal{L} ledger or set of transactions

DAG-related notation

$D=(V, E)$ directed acyclic graph (DAG) with vertex set V and edge set E
 $D_{\mathcal{L}}$ Ledger DAG
 $D_{\mathcal{C}}$ Conflict DAG
 $\text{child}_V(x)$ set of children of vertex x in DAG $D=(V, E)$
 $\text{par}_V(x)$ set of parents of vertex x in DAG $D=(V, E)$
 $\text{cone}_V^{(f)}(x)$ future cone of vertex x in DAG $D=(V, E)$
 $\text{cone}_V^{(p)}(x)$ past cone of vertex x in DAG $D=(V, E)$
 $\text{label}^{(p)}(x)$ set of labels in past cone of x in $D_{\mathcal{L}}$

Order and relationship definitions

\leq_V partial order on set V (usually induced by a given DAG $D=(V, E)$)
 $N_V(x)$ set of neighbours of a vertex x in graph $G=(V, E)$
 $\max_V(S)$ set of maximal elements in set S (maximal according to DAG $D=(V, E)$)
 $\min_V(S)$ set of minimal elements in set S (minimal according to DAG $D=(V, E)$)

Figure 5: Overview of the main notations.

4.1 UTXO and Ledger DAGs

We introduced the concept of UTXO and defined UTXO based transactions as an operation spending inputs and creating outputs in Section 2. The inputs and outputs in one transaction are “atomic” in the sense that either all inputs are consumed and all outputs are created or the transaction is not added to the ledger at all. The atomic nature of a transaction is represented by a unique transaction ID. In our graphical representation, these dependencies are expressed using an additional vertex identified with the corresponding transaction ID, see Figure 6.

The collection of all transactions since genesis, i.e., the ledger \mathcal{L} , provides the content for a DAG, which we call the UTXO DAG.

Definition 4.3 (UTXO DAG). The vertex set of the UTXO DAG consists of all in- and outputs and all transaction IDs. The interrelations between these form the set of directed edges. More specifically, directed edges exist from inputs to outputs, from the transaction ID to its inputs, and from the outputs to the ID of the transaction creating these outputs. We allow here to appear inputs several times as vertices, turning the vertex set formally into a multi-set. This allows to track

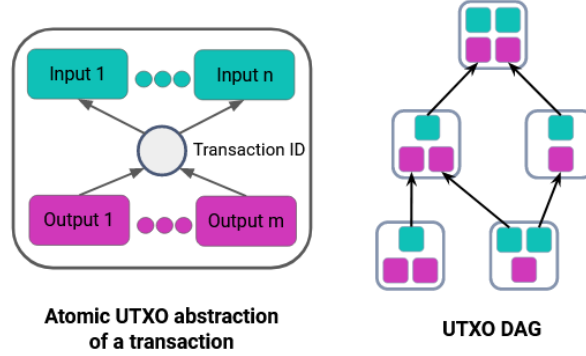


Figure 6: Atomic UTXO abstraction of a transaction and an example of a UTXO DAG.

possible double spends.

Example 4.1. For a simplified illustration of an example for such a DAG see Figure 6. The depicted UTXO DAG contains five transactions.

Definition 4.4 (Ledger DAG). We define the Ledger DAG $D_{\mathcal{L}}$ to be a DAG whose vertex set is the ledger \mathcal{L} . There is a directed edge (x, y) in the edge set of $D_{\mathcal{L}}$ if and only if an input of x references an output of y .

Remark 4.3. *The results derived in this paper are in respect to the Ledger DAG. However, due to the atomic nature of transactions the results also apply for the UTXO DAG.*

Equipped with the notion from Section 3, we write $\leq_{\mathcal{L}}$ to denote the partial order on the ledger \mathcal{L} induced by $D_{\mathcal{L}}$. In other words, $y \leq_{\mathcal{L}} x$ if transaction y spends (indirectly) from x . Note that the genesis ρ is the only $D_{\mathcal{L}}$ -maximal element in \mathcal{L} . Further, we write $\text{cone}_{\mathcal{L}}^{(p)}(x)$ to denote the Ledger past cone and $\text{cone}_{\mathcal{L}}^{(f)}(x)$ to denote the Ledger future cone.

As a typical rule in a DLT with a UTXO model, an output can only be spent once. Thus, if there are multiple transactions that attempt to spend the same output, it is the role of the consensus mechanism to select at most one transaction that is allowed to consume the output. Once some consensus mechanism decided on which conflicts to keep and which to reject, we can reduce or prune the augmented ledger as described in Section 5.3.

4.2 Conflict Graph

In Definition 4.1 we introduced the notion of conflicts. Due to the causal dependency of ordered transactions, transactions can be conflicting even if they do not consume the same output.

Definition 4.5 (Conflicting transactions). Two distinct transactions $x, y \in \mathcal{L}$ are *directly conflicting* if they have at least one input in common. Two distinct transactions $x_1, y_1 \in \mathcal{L}$ are said to be

indirectly conflicting if there exist distinct $x_2, y_2 \in \mathcal{L}$ with either $x_1 <_{\mathcal{L}} x_2$ and $y_1 \leq_{\mathcal{L}} y_2$ or $x_1 \leq_{\mathcal{L}} x_2$ and $y_1 <_{\mathcal{L}} y_2$ such that x_2 and y_2 are directly conflicting. Two transactions are said to be *conflicting* if they are directly or indirectly conflicting.

Remark 4.4. Note that due to Definition 4.1 some (or possibly the majority of) conflicting transactions are not necessarily conflicts. On the other hand, if two transactions are directly conflicting, then they are conflicts.

Definition 4.6 (Conflict-free set and conflicting sets). A subset of transactions $S \subseteq \mathcal{L}$ is called *conflict-free* if it does not contain any two conflicting transactions. We also say that $S_1 \subseteq \mathcal{L}$ is *conflict-free with respect to* $S_2 \subseteq \mathcal{L}$ if there is no $c_1 \in S_1$ and $c_2 \in S_2$ such that c_1 and c_2 are conflicting. Alternatively, S_1 is *conflicting* with S_2 if S_1 is not conflict-free with respect to S_2 .

By Remark 4.4, conflicting transactions are not necessarily conflicts. However, the $D_{\mathcal{L}}$ -maximal transactions that are conflicting with a given transaction have to be conflicts as described below.

Proposition 4.1. For a transaction $x \in \mathcal{L}$, define $C = C(x)$ to be the set of transactions that are conflicting with x . Then C is $D_{\mathcal{L}}$ -future-closed and it holds that $\max_{\mathcal{L}}(C) \subseteq C$.

Proof. By Definition 4.5, if x is conflicting with a transaction $y \in \mathcal{L}$, then x is conflicting with z , where z is any transaction $z \in \mathcal{L}$ such that $z \leq_{\mathcal{L}} y$. Thus, the set C has to be $D_{\mathcal{L}}$ -future-closed. Let $u \in \max_{\mathcal{L}}(C)$. By definition of conflicting transactions, there exists some x' and u' such that $x \leq_{\mathcal{L}} x'$, $u \leq_{\mathcal{L}} u'$ and x' and u' are directly conflicting. Then by Definitions 4.1 and 4.5, $u' \in C$. Note that u' and x are conflicting and, thus, $u' \in C$. Since $u \leq_{\mathcal{L}} u'$ and $u' \in \max_{\mathcal{L}}(C)$, we conclude that $u = u' \in C$. \square

The set of relations between conflicts can be described with the notion of a Conflict Graph.

Definition 4.7 (Conflict Graph). The *Conflict Graph* $G_{\mathcal{C}}$ has vertex set \mathcal{C} . Two vertices in $G_{\mathcal{C}}$ are connected by an undirected edge if and only if the corresponding two conflicts are conflicting.

Example 4.2. We refer the reader to Figure 7 for an illustration of conflicts and the Conflict Graph. On the left part of the figure, we depict a UTXO DAG, where by coloring the box of a transaction, we indicate whether the transaction is a conflict. For instance, it can be seen that the orange transaction is directly conflicting with the red transaction, whereas the blue transaction is indirectly conflicting with the purple transaction. The relations between conflicts are demonstrated with the help of the corresponding Conflict Graph which is depicted on the right part of the figure.

In contrast to the standard UTXO model, where no conflicts are allowed, they can be present in our generalization. We, however, require any two conflicting transactions to be not comparable by the partial order $\leq_{\mathcal{L}}$. The constraints in Assumption 2.1 about the addition of a transaction are relaxed:

Assumption 4.1 (Reality-based Ledger constraints). A transaction x is added to the ledger \mathcal{L} if it follows the following rules:

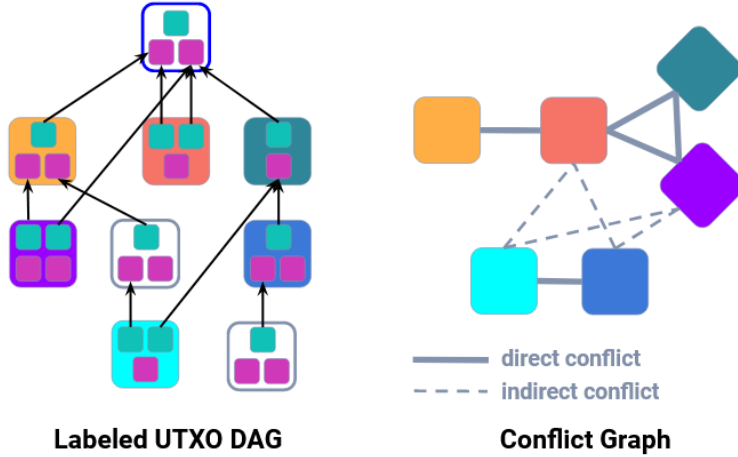


Figure 7: The UTXO DAG representation and the Conflict Graph

1. the transaction x is syntactically correct;
2. the sum of values of the inputs equals the sum of values of the outputs;
3. the unlocking data $\text{unlock}(x)$ is valid;
4. $\text{in}(x)$ are references to outputs which are not already consumed in $\text{cone}_{\mathcal{L}}^{(p)}(x) \setminus \{x\}$

Remark 4.5 (Conflict-free past cone). *A consequence of the 4th point in Assumption 4.1 is that all past cones are conflict-free. In other words, we have that for every transaction $x \in \mathcal{L}$, $\text{cone}_{\mathcal{L}}^{(p)}(x)$ does not contain any pair of two conflicting transactions.*

4.3 Labeled UTXO and Ledger DAGs and Conflict DAG

The existence of conflicts in the Ledger DAG plays a crucial role, as they eventually need to be resolved. In the following, we extract the necessary information for conflict resolution from the Ledger DAG. To this end, we labeled the transactions, see Definition 4.2. We add this labeling to our UTXO and Ledger DAGs to keep track of the conflicts and their dependencies.

Definition 4.8 (Labeled UTXO and Ledger DAGs). *The labeled UTXO and Ledger DAGs are the UTXO and Ledger DAGs with the additional labels as described in Definition 4.2.*

Example 4.3. For an illustration of a labeled Ledger DAG, we refer the reader to Figures 7 and 8. Note that the different colors correspond to the different labels in these figures.

Now we define the restriction of the labeled Ledger DAG to the conflict set and the genesis using Definition 3.5.

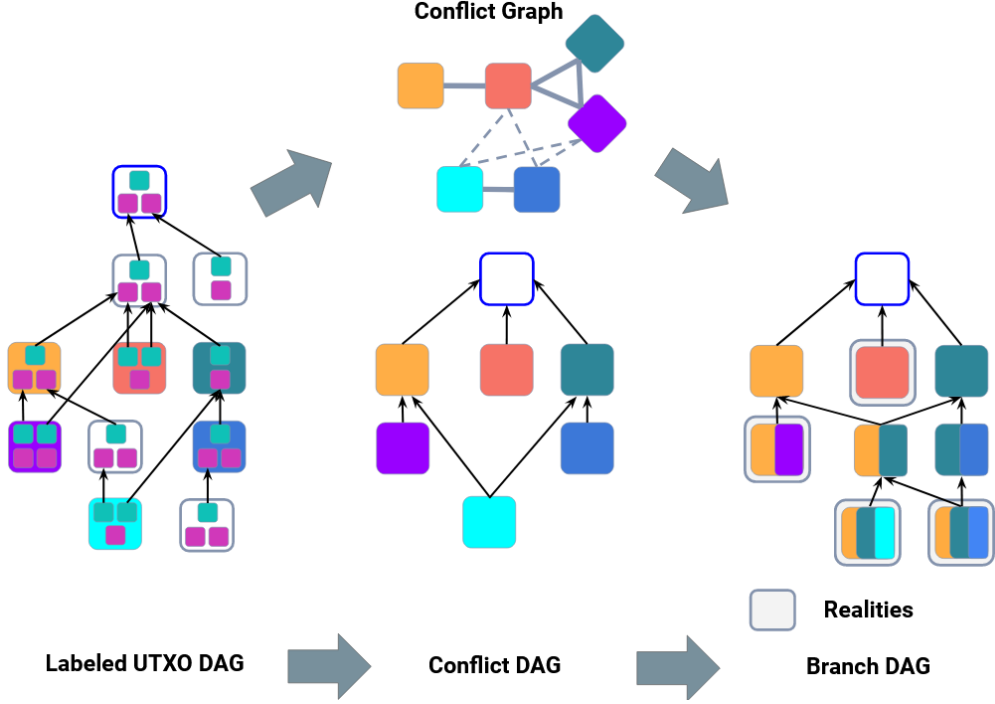


Figure 8: Derivation of the various graphs from the labeled UTXO DAG and the resulting realities (leaves of the Branch DAG). The colors represent the different labels of the conflicts.

Definition 4.9 (Conflict DAG). The *Conflict DAG*, written as $D_{\mathcal{C}}$, is defined as the minimal subDAG of $D_{\mathcal{C}}$ induced by the set of vertices $\mathcal{C} \cup \{\rho\}$.

Example 4.4. For a more visual explanation of the above concept, we depict Figure 8. Specifically, we demonstrate a UTXO DAG on the left part of the figure and the corresponding Conflict DAG in the middle. We note that the purple transaction in the Conflict DAG is not connected by an edge with the white one since there is a path connecting them which goes through the yellow transaction.

Remark 4.6. We observe that the Conflict DAG and the Conflict Graph represent only some partial information about the labeled UTXO DAG. Specifically, they are used to demonstrate different relations between the conflicts. Note that in general, it is not possible to construct the Conflict DAG using only the Conflict Graph and vice versa.

Using the notion from Section 3, we denote the partial order on the set \mathcal{C} induced by $D_{\mathcal{C}}$ by $\leq_{\mathcal{C}}$. The past and future cones of a conflict $x \in \mathcal{C}$ in the Conflict DAG are written as $\text{cone}_{\mathcal{C}}^{(p)}(x)$ and $\text{cone}_{\mathcal{C}}^{(f)}(x)$, respectively. Finally, we give an observation saying that if a transaction is conflicting with some subset of conflicts S (see Definition 4.6), it is possible to find a $D_{\mathcal{C}}$ -minimal conflict in S which is conflicting with that transaction.

Proposition 4.2 (Transaction conflicting with a set). *Let a subset of conflicts $S \subseteq \mathcal{C}$ be conflicting with a transaction $x \in \mathcal{L}$. Then there exists $c \in \min_{\mathcal{C}}(S)$ such that x and c are conflicting.*

Proof. By Definition 4.6, there exists some $y \in S$ such that y and x are directly or indirectly conflicting. From Definition 3.6, it follows that there exists some $c \in \min_{\mathcal{C}}(S)$ such that $c \leq_{\mathcal{C}} y$. Then by Definition 4.5, we conclude that c is conflicting with x . \square

4.4 Branches and Branch DAG

This section introduces the concepts of branches and Branch DAG, which help to handle the conflicting transactions.

Definition 4.10 (Branch and set of branches). A set of conflicts $B \subseteq \mathcal{C}$ is called a *branch* if and only if the two properties hold:

1. B is conflict-free (cf. Definition 4.6);
2. B is $D_{\mathcal{C}}$ -past-closed (cf. Definition 3.8).

Define \mathcal{B} to be the set of all branches, including the so-called *main branch* which represents the empty set.

Note that by Assumption 4.1(4), for any conflict $x \in \mathcal{C}$, the past cone $\text{cone}_{\mathcal{C}}^{(p)}(x)$ is a branch. In the following statement we discuss a sufficient condition for the union of branches to be a branch.

Lemma 4.1. *Let $B_1, \dots, B_n \in \mathcal{B}$ be branches such that there exists a branch $A \in \mathcal{B}$ with $B_1, \dots, B_n \subseteq A$. Then, the union*

$$B_1 \cup \dots \cup B_n$$

is also a branch, called the aggregate branch of B_1, \dots, B_n .

Proof. Since, every B_i is $D_{\mathcal{C}}$ -past-closed, the union $B_1 \cup \dots \cup B_n$ is also $D_{\mathcal{C}}$ -past-closed. Since A contains no conflicting pairs, the union of must not either. \square

We proceed with a crucial observation saying that each branch can be represented as the aggregated branch of certain past cones.

Lemma 4.2 (Aggregated branch). *Every branch B can be uniquely written as the aggregated branch*

$$B = \text{cone}_{\mathcal{C}}^{(p)}(c_1) \cup \dots \cup \text{cone}_{\mathcal{C}}^{(p)}(c_n),$$

where conflicts c_1, \dots, c_n are the $D_{\mathcal{C}}$ -minimal elements in B .

Proof. The branch B is finite. Hence, there exist unique $D_{\mathcal{C}}$ -minimal conflicts c_1, \dots, c_n in B , i.e., for any c_i and any other conflict $d \in B$, either it holds that $c_i \leq_{\mathcal{C}} d$ or d and c_i are not comparable by the partial order. Since B contains the past cone of all its conflicts, it follows that B is the aggregated branch of the branches $\text{cone}_{\mathcal{C}}^{(p)}(c_i)$. \square

Remark 4.7. Lemma 4.2 has some fundamental consequences for how we can implement the branches into the protocol. For instance, if a branch B has a unique decomposition into

$$B = \text{cone}_{\mathcal{C}}^{(p)}(c_1) \cup \dots \cup \text{cone}_{\mathcal{C}}^{(p)}(c_n),$$

then we can set the branch ID of B to be the hash of the concatenation of the transaction IDs of the c_i 's (ordered in a canonical way). Then, the branch ID of a branch of the form $\text{cone}_{\mathcal{C}}^{(p)}(c)$ with $c \in \mathcal{C}$ is the same as the hash of the conflict c .

A branch B_1 is called a *subbranch* of branch B_2 if $B_1 \subseteq B_2$. Lemma 4.2 already shows the recursive structure of the branches. This recursive structure can be encoded in the Branch DAG.

Definition 4.11 (Branch DAG). Ordered by inclusion, \mathcal{B} is a partially ordered set and defines a DAG. Specifically, we put a directed edge from a branch A to a branch B if $A = B \cup c$ for some conflict $c \in \mathcal{C} \setminus B$. The corresponding DAG is called the *Branch DAG* and denoted by $D_{\mathcal{B}}$.

Remark 4.8. We show an example of a Branch DAG in Figure 8. In this example, the number of vertices in the Branch DAG is nine which is larger than six, the number of conflicts. We note that in general, the number of vertices in a Branch DAG can be exponentially large in the number of conflicts. For instance, if there exist t pairs of directly conflicting transactions such that any two transactions from different pairs are not conflicting, then the number of vertices in the Branch DAG is lower bounded by 2^t . We refer the reader to Section 6, where we explain how some functionalities based on the natural concept of a Branch DAG can be implemented in a more efficient way.

Applying the notion from Section 3, we denote the partial order on the set \mathcal{B} induced by $D_{\mathcal{B}}$ by $\leq_{\mathcal{B}}$; the set of parents and children of branch $B \in \mathcal{B}$ is written as $\text{par}_{\mathcal{B}}(B)$ and $\text{child}_{\mathcal{B}}(B)$.

Conflicting transactions owe the existence of their conflict state to the presence of conflicts in their Ledger past cone. Since conflicts are labelled transactions, we can define a function that extracts all labels in this past cone.

Definition 4.12 (Maximal contained label set). Let \mathcal{Y} be the label space, and $\text{label}^{(p)} : \mathcal{L} \rightarrow 2^{\mathcal{Y}}$ be a function that for a given transaction $x \in \mathcal{L}$ returns all labels of the transactions in its Ledger past cone i.e., $\text{label}^{(p)}(x) = \{\text{label}(y) : y \in \text{cone}_{\mathcal{L}}^{(p)}(x)\}$.

Remark 4.9. Practically this operation can be performed, e.g., through a graph search algorithm applied to the Ledger DAG (more computational intensive), or through a transaction-by-transaction record of conflict dependencies (more memory expensive). On one hand, for a given transaction x we can identify all transactions with labels in the past cone $\text{cone}_{\mathcal{L}}^{(p)}(x)$ by traversing the graph by means of depth-first search. We can discontinue to search deeper than certain elements by cross-checking with the conflict set. On the other hand, we can inherit the maximal contained label set for a new arriving transaction from its parents and if a new conflict with x is created, we traverse the future cone $\text{cone}_{\mathcal{L}}^{(f)}(x)$ and update the maximal contained label set for all transactions there.

We can also define an equivalent function to obtain branch dependencies.

Definition 4.13 (Maximal contained branch). Let \mathcal{B} be the set of all branches, and $\text{branch}_{\mathcal{L}}^{(p)} : \mathcal{L} \rightarrow \mathcal{B}$ be a function that for a given transaction $x \in \mathcal{L}$ returns the maximal branch contained in $\text{cone}_{\mathcal{L}}^{(p)}(x)$.

We note that there could not be two maximal branches in the Ledger past cone of a transaction (which is conflict-free) since, otherwise, we could consider their aggregate branch. The above two definitions have the following correlation.

Lemma 4.3. *The maximal contained label set of a transaction x translates to the maximal branch that is contained in the past cone $\text{cone}_{\mathcal{L}}^{(p)}(x)$. More precisely, we have that*

$$\text{label}^{(p)}(x) \setminus \{\perp\} = \bigcup_{c \in \text{branch}_{\mathcal{L}}^{(p)}(x)} \text{label}(c).$$

Proof. By definition, $\text{label}^{(p)}(x) = \{\text{label}(y) : y \in \text{cone}_{\mathcal{L}}^{(p)}(x)\}$. The branch $\text{branch}_{\mathcal{L}}^{(p)}(x)$ is included to $\text{cone}_{\mathcal{L}}^{(p)}(x)$ and \perp is not a label for any conflict by Definition 4.2. This implies that

$$\bigcup_{c \in \text{branch}_{\mathcal{L}}^{(p)}(x)} \text{label}(c) \subseteq \text{label}^{(p)}(x) \setminus \{\perp\}.$$

Toward a contradiction, assume that the equality in the above formula does not hold, i.e., there exists some label $\ell \in \mathcal{Y}$ which is not present in the left-hand side. Consider the unique conflict $y \in \text{cone}_{\mathcal{L}}^{(p)}(x)$ such that $\text{label}(y) = \ell$. We shall prove that this conflict should be included to the maximal contained branch. Indeed, the union $\text{cone}_{\mathcal{L}}^{(p)}(y) \cup \text{branch}_{\mathcal{L}}^{(p)}(x)$ is $D_{\mathcal{C}}$ -past-closed and is conflict-free as included to $\text{cone}_{\mathcal{L}}^{(p)}(x)$. It follows that $y \in \text{branch}_{\mathcal{L}}^{(p)}(x)$ and ℓ is present in the left-hand side of the displayed equation, which contradicts the assumption. \square

4.5 Realities in the Branch DAG

In this section, we discuss maximal aggregated branches. They are branches that present maximal acceptable valid versions of the ledger.

Definition 4.14 (Maximal branch and reality). A branch $B \in \mathcal{B}$ is *maximal* if there exists no other branch $A \in \mathcal{B}$ such that $B \subset A$. A maximal branch is called a *reality*.

Note that a reality always contains the main branch by definition since the empty set is included to all branches. An immediate consequence of the above definition is the following lemma.

Lemma 4.4. *The set of realities equals the set of leaves in the Branch DAG.*

Example 4.5. Following the example depicted in Figure 8, we observe that there are exactly four realities or leaves in the Branch DAG.

The following statement shows a link between realities and maximal independent sets in the Conflict Graph.

Proposition 4.3. *There is a one-to-one correspondence between maximal independent sets of the Conflict Graph and realities.*

Proof. Let $I = \{c_1, \dots, c_n\} \subseteq \mathcal{C}$ be a maximal independent set in the Conflict Graph. We define the set B as follows

$$B := B(c_1, \dots, c_n) := \text{cone}_{\mathcal{C}}^{(p)}(c_1) \cup \dots \cup \text{cone}_{\mathcal{C}}^{(p)}(c_n).$$

Since each past cone is $D_{\mathcal{C}}$ -past-closed, the union is either. Assume that B contains a conflicting pair of transactions, say d and e . There must exist conflicts c_i and c_j such that $d \in \text{cone}_{\mathcal{C}}^{(p)}(c_i)$ and $e \in \text{cone}_{\mathcal{C}}^{(p)}(c_j)$. From Definition 4.5 it follows that c_i and c_j are conflicting which implies a contradiction to the fact that I is an independent set in the Conflict Graph.

The branch B is also maximal. To see this assume the existence of a larger branch A containing B , i.e., $B \subset A$ and let $d \in A \setminus B$ be a conflict from A not included to B . Then d and c_1, \dots, c_n are pairwise indirectly non-conflicting which contradicts the fact that the independent set I is maximal.

Conversely, let $B = \{c_1, \dots, c_n\}$ be a reality. Hence, c_1, \dots, c_n are not conflicting and $I := B$ is an independent set of the Conflict Graph. Toward a contradiction assume that I is not maximal, i.e., there exists $d \in \mathcal{C} \setminus B$ such that d and c_1, \dots, c_n are pairwise non-conflicting. Define A to be $\text{cone}_{\mathcal{C}}^{(p)}(d) \cup \text{cone}_{\mathcal{C}}^{(p)}(c_1) \cup \dots \cup \text{cone}_{\mathcal{C}}^{(p)}(c_n)$. Clearly, A is a branch containing B as a subbranch. We arrive to a contradiction with Definition 4.14. \square

Definition 4.15 (Ledger of a reality). Let R be a reality. We define the R -ledger as

$$\mathcal{L}_r(R) := \{x \in \mathcal{L} : \text{label}^{(p)}(x) \subseteq \{\perp\} \cup \text{label}(R)\},$$

where $\text{label}(R) := \bigcup_{c \in R} \text{label}(c)$.

We can now give one of our main results as a direct consequence of the construction the Reality-based Ledger.

Theorem 4.1. *Let R be a reality. Then, $\mathcal{L}_r(R)$ is a consistent ledger (cf. Definition 2.5).*

Proof. We have to prove that Assumption 4.1 implies Assumption 2.1 for every R -ledger. Constraints 1) – 3) are trivially satisfied. Constraint 4) in Assumption 2.1 follows from the fact that a reality is a branch (and, hence, conflict-free) together with Constraint 4) in Assumption 4.1. \square

Similar to a conflict-free ledger and Remark 2.2, we provide invariance properties of the Reality-based Ledger.

Remark 4.10 (Invariance properties of Reality-based Ledger). *The Reality-based Ledger upholds certain invariance properties and can thus be seen as an invariant data structure. More specifically, we notice that*

1. the Reality-based Ledger might depend on the time parameter t , i.e. $\mathcal{L} = \mathcal{L}(t)$;
2. for any given reality $R \in \mathcal{B}$ at any given time t , it holds that the sum of output values of the state of the R -ledger remains constant

$$\sum_{\substack{o=(v,\text{cond}) \\ o \in \text{state}(\mathcal{L}_r(R))}} v = \text{const.}$$

5 Operations on the Reality-based Ledger

In this section, we define several operations that we can perform on the Reality-based Ledger. In Section 5.1, we describe what happens when new conflicts are added to the ledger. In Section 5.2 we provide an algorithm for the selection of a reality, i.e., a valid conflict-free ledger state, in the presence of a weight function imposed on transactions and branches. Finally, in Section 5.3 we describe how and when conflicts can be pruned to maintain a reasonable data consumption for the operation of the Ledger. We refer to Figure 9 for an overview of the different operations and their dependencies.

5.1 Adding Conflicts

In this section, we explain how new conflicting transactions result in updating the Conflict DAG and the Conflict Graph. Throughout this section we assume that there is only one new transaction x which is a leaf in the Ledger DAG. In this case, the Ledger DAG is updated with only directed edges of the form (x, y) for some $y \in \mathcal{L}$.

First, we introduce a concept of closest conflicts in the past and future cones of a transaction which will be used for updating the Conflict DAG.

Definition 5.1 (Closest conflicts in past and future cones). For a transaction $z \in \mathcal{L}$, we define $\text{conf}_{\mathcal{L}}^{(p)}(z)$ to be $\min_{\mathcal{C}}(S^{(p)})$ for $S^{(p)} := \{y \in \mathcal{C} \cup \{\rho\} : y <_{\mathcal{L}} z\}$, i.e., $S^{(p)}$ is the set of all conflicts in $\text{cone}_{\mathcal{L}}^{(p)}(z) / \{z\}$. Similarly, we define $\text{conf}_{\mathcal{L}}^{(f)}(z)$ to be $\max_{\mathcal{C}}(S^{(f)})$ for $S^{(f)} := \{y \in \mathcal{C} : y <_{\mathcal{L}} z\}$, i.e., $S^{(f)}$ is the set of all conflicts in $\text{cone}_{\mathcal{L}}^{(f)}(z) / \{z\}$.

This notion resembles Definition 4.12 and now we give a remark similar to Remark 4.9 on how to find these sets.

Remark 5.1. Observe that $\text{conf}_{\mathcal{L}}^{(p)}(z)$ and $\text{conf}_{\mathcal{L}}^{(f)}(z)$ can be obtained by first finding closest conflicts using breadth-first search (BFS) and reverse breadth-first search (RBFS). That means that in both cases we traverse $D_{\mathcal{L}}$ starting at z and stop traversing through transactions that are conflicts. Eventually, we identify the minimal/maximal elements in the obtained sets.

We provide a possible scheme to update the Conflict DAG in Algorithm 1. Let x be a new transaction and $Y \subseteq \mathcal{L}$ be the set of all transaction conflicting with x . Then, in Algorithm 1, we

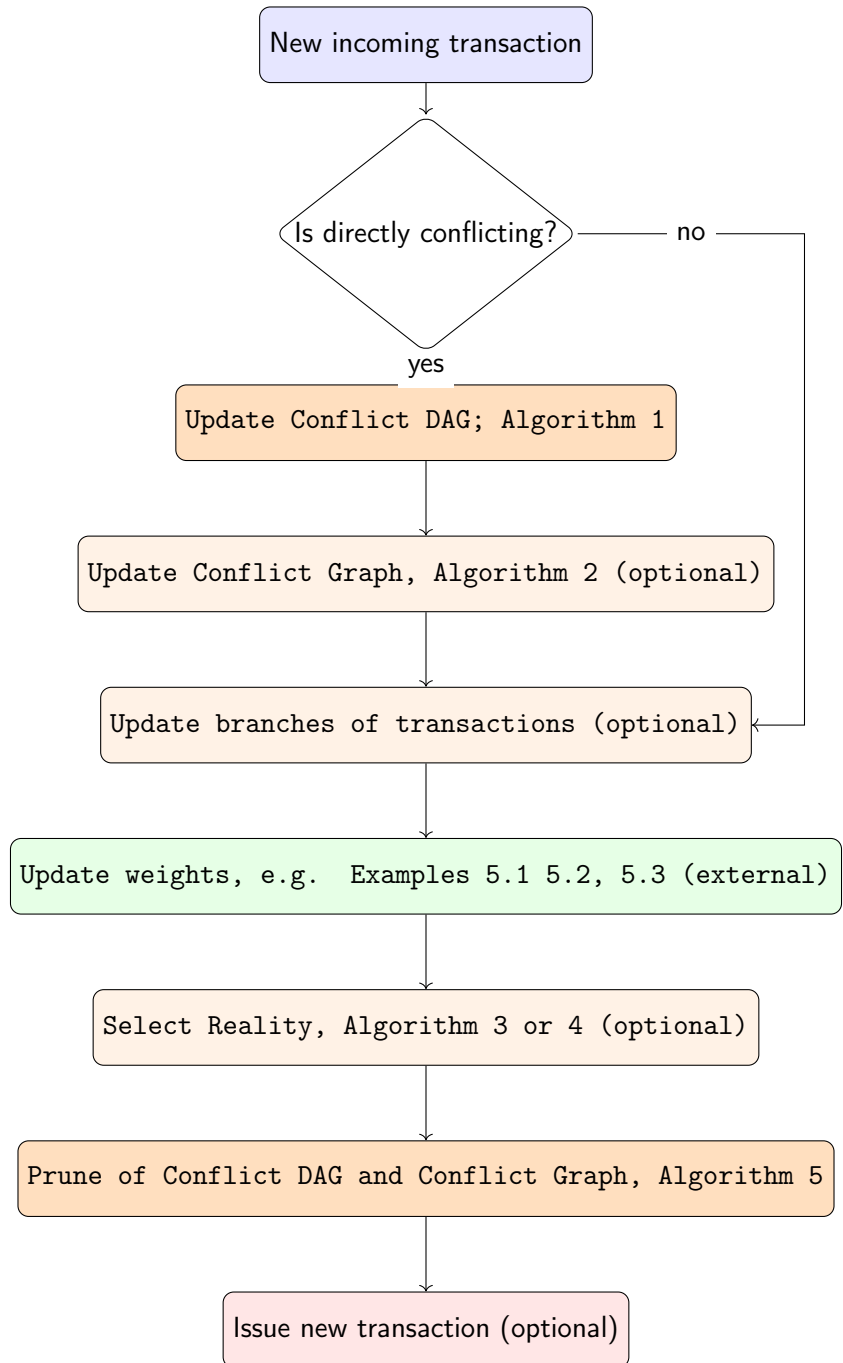


Figure 9: Flow of the different operations on the reality-based ledger.

Algorithm 1: Algorithm to update Conflict DAG

Data: Conflict DAG $D_C = (\mathcal{C} \cup \{\rho\}, E)$; new transaction $x \in \mathcal{L}$ that is directly conflicting with transactions $Y \subseteq \mathcal{L}$

Result: updated Conflict DAG $D_C = (\mathcal{C} \cup \{\rho\}, E)$

```
1  $\mathcal{C} \leftarrow \mathcal{C} \cup \{x\} \cup Y$ 
2 for  $\forall y \in Y \cup \{x\}$  do
3   for  $\forall v \in \text{conf}_{\mathcal{L}}^{(p)}(y)$  do
4      $E \leftarrow E \cup \{(y, v)\}$ 
5   end
6   for  $\forall v \in \text{conf}_{\mathcal{L}}^{(f)}(y)$  do
7      $E \leftarrow E \cup \{(v, y)\}$ 
8   end
9 end
10 for  $\forall y \in Y$  do
11   for  $\forall p \in \text{par}_{\mathcal{C}}(y)$  do
12     for  $\forall c \in \text{child}_{\mathcal{C}}(p)$  do
13       if  $c \in \text{cone}_{\mathcal{C}}^{(f)}(y)$  then
14          $E \leftarrow E \setminus \{(c, p)\}$ 
15       end
16     end
17   end
18 end
```

1. update the set of vertices by adding the conflicts $x \cup Y$;
2. add edges to the Conflict DAG using the notion of closest conflicts in the past and future cones of transaction from $x \cup Y$;
3. remove unnecessary edges in the Conflict DAG to keep it in the minimal form.

Lemma 5.1. *The resulting graph $D_C = (\mathcal{C} \cup \rho, E)$ in Algorithm 1 is the Conflict DAG as defined in Definition 4.9.*

Proof. Recall that the Conflict DAG is the minimal subDAG of the Ledger DAG induced by \mathcal{C} . It is sufficient to check that we remove all unnecessary edges after adding correct edges using the notions $\text{conf}_{\mathcal{L}}^{(p)}(y)$ and $\text{conf}_{\mathcal{L}}^{(f)}(y)$.

Assume that some edge (c, p) , which was an edge in the original Conflict DAG, has to be removed to keep the Conflict DAG in the minimal form. This means that both c and p were already conflicts such that $c <_{\mathcal{C}} p$ with no other conflict between them and now some conflict $y \in Y$ satisfies

Algorithm 2: Algorithm to update Conflict Graph

Data: Conflict Graph $G_C = (\mathcal{C}_{old}, E)$; new transaction $x \in \mathcal{L}$ that is directly conflicting with transactions $Y \subseteq \mathcal{L}$; updated Conflict DAG $D_C = (\mathcal{C} \cup \rho, E)$;

Result: updated Conflict Graph $G_C = (\mathcal{C}, E)$

```
1 for  $\forall y \in Y$  do
2   | for  $\forall z \in \text{cone}_C^{(f)}(y)$  do
3   |   |  $E \leftarrow E \cup \{(x, z)\}$ 
4   |   end
5   end
6 for  $\forall y \in Y \cup \{x\}$  do
7   | ; /* assume  $D_C$ -descending order */
8   | for  $\forall p \in \text{par}_C(y)$  do
9   |   | for  $\forall z \in N_C(p)$  do
10  |   |   | ; /*  $N_C(p)$  denotes the set of neighbours of  $p$  in  $G_C$  */
11  |   |   |  $E \leftarrow E \cup \{(y, z)\}$ 
12  |   |   end
13  |   end
14  end
```

$c <_C y <_C p$. Let y^* be a D_C -minimal conflict among all y satisfying the latter inequality, i.e., $c \in \text{child}_C(y^*)$. One can check that all such edges (c, p) are removed from the Conflict DAG in line 14 of Algorithm 1. \square

Assume that Algorithm 1 is already completed. In Algorithm 2 we describe a possible procedure to update the Conflict Graph. In this algorithm, we only add new edges to the Conflict Graph G_C . Specifically, all conflicts in the future cones of conflicts in Y become adjacent with x in G_C . In addition, all conflicts in $x \cup Y$ inherit G_C -neighbours from their parents in D_C .

Lemma 5.2. *The resulting graph $G_C = (\mathcal{C}, E)$ in Algorithm 2 is the Conflict Graph as defined in Definition 4.7.*

Proof. By Definition 4.7, if two conflicts $c_1, c_2 \in \mathcal{C}$ are connected by an edge in the Conflict Graph, then there exist $e_1, e_2 \in \mathcal{C}$ with $c_1 \leq_C e_1$ and $c_2 \leq_C e_2$ such that e_1 and e_2 are directly conflicting. Let E' denote the set of edges to be included to the Conflict Graph after transaction x arrives. We claim that any edge from E' should contain at least one conflict from the set $x \cup Y$. This is true since x is a leaf in both the Ledger DAG and the Conflict DAG.

Let us start with considering edges from E' of type (x, z) for some $z \in \mathcal{C}$ such that there exists $y \in \mathcal{C}$ with $z \leq_C y$, and such that x and y are directly conflicting. Clearly, y has to be from Y since x is directly conflicting with transactions from Y only. In line 3 of Algorithm 2, by traversing over all transactions z in the Conflict DAG that are contained in the future cones of conflicts from Y , we add all such edges (x, z) to E .

Observe that there is no edges in E' of type (y, z) for $y \in Y$ and $z \in \mathcal{C} \setminus \{x\}$ such that there exists $w \in \mathcal{C}$ with $z \leq_{\mathcal{C}} w$ and such that y and w are directly conflicting. That is true since we do not include any edge in the Ledger DAG that contains $y \in Y$ when transaction x arrives.

Thus, it remains to add edges of type (y, z) with $y \in Y \cup \{x\}$ such that there exist $e_1, e_2 \in \mathcal{C}$ with $y <_{\mathcal{C}} e_1$ and $z <_{\mathcal{C}} e_2$ such that e_1 and e_2 are directly conflicting. Note that the above inequalities in the partial order relations are strict and we can utilize the notion of parents in $D_{\mathcal{C}}$ to add edges recursively. Thereby, by assuming some $D_{\mathcal{C}}$ -descending order over the set $Y \cup \{x\}$, we iteratively perform the following step in line 9 of Algorithm 2. For every conflict $y \in Y \cup \{x\}$, the set of $G_{\mathcal{C}}$ -neighbors of y is updated by looking at $G_{\mathcal{C}}$ -neighbours of parents of y in $D_{\mathcal{C}}$ as follows

$$N_{\mathcal{C}}(y) \leftarrow N_{\mathcal{C}}(y) \cup \left\{ \bigcup_{p \in \text{par}_{\mathcal{C}}(y)} N_{\mathcal{C}}(p) \right\}.$$

□

5.2 Reality Selection Algorithms

In a system with a conflict-free UTXO Ledger, the account owners can learn about their current balance by inspecting the state of an instance of a ledger, see Section 2. In the case of the Reality-based Ledger this is more complicated, since only a single reality represents a valid conflict-free ledger, and thus can yield a valid ledger state. It is, therefore, up to the operator of the ledger instance, to choose which ledger state to evaluate to inform the account owners about their balance. Alternatively, and in a more trustless fashion, the account owner and the operator of the ledger instance constitute the same entity.

In this section, we propose reality selection algorithm that construct a preferred reality by utilizing a weight function for transactions. We impose several natural constraints on the weight function.

Assumption 5.1. We assume that there exists a weight function on the set of transactions $\mathbf{w} : \mathcal{L} \rightarrow [0, 1]$ which satisfies the following properties

1. unitarity: $\mathbf{w}(\rho) = 1$;
2. monotonicity: for any two transactions $x, y \in \mathcal{L}$ such that $x \leq_{\mathcal{L}} y$, it holds that

$$\mathbf{w}(x) \leq \mathbf{w}(y);$$

3. consistency: let x_1, \dots, x_s be pairwise conflicting transactions.³ Then it holds that

$$\sum_{i=1}^s \mathbf{w}(x_i) \leq 1.$$

³We say that transactions $S \subseteq \mathcal{L}$ are pairwise conflicting if any pair of transactions $x, y \in S$ are conflicting.

We naturally extend the domain of the function \mathbf{w} to the set of all branches \mathcal{B} as follows. For a branch $B \in \mathcal{B}$, we define $\mathbf{w}(B)$ to be

$$\mathbf{w}(B) := \min_{x \in B} \mathbf{w}(x).$$

Remark 5.2. A weight function \mathbf{w} induces a weight function on the set of branches \mathcal{B} with the following monotonicity property: let $B_1, B_2 \in \mathcal{B}$ such that $B_1 \subseteq B_2$, it holds that

$$\mathbf{w}(B_1) \geq \mathbf{w}(B_2).$$

If the weight function satisfies Assumption 5.1, we also have that $\mathbf{w}(B_1) + \mathbf{w}(B_2) \leq 1$ for any two conflicting branches $B_1, B_2 \in \mathcal{B}$. We also observe that for a conflict $c \in \mathcal{C}$, $\mathbf{w}(c) = \mathbf{w}(\text{cone}_{\mathcal{C}}^{(p)}(c))$.

Let us give some examples of weight functions satisfying Assumption 5.1.

Example 5.1 (Minimal hash). Transactions can be ordered, relatively to each other, through several means. One such way is to hash the content of the transaction, see Remark 2.1. Here we assume the existence of such a function $\text{hash} : \mathcal{L} \rightarrow \{0, 1\}^h$. We define the weight function using the following steps:

1. $\mathbf{w}(\rho) = 1$;
2. for every $x \in \mathcal{C}$ we set $\mathbf{w}(x) = 1$ if

$$\text{hash}(x) = \min_{y \in N_{\mathcal{C}}(x)} \text{hash}(y),$$

and $\mathbf{w}(x) = 0$ otherwise;

3. inductively starting from the genesis for every $x \in \mathcal{L} \setminus \mathcal{C}$ we set

$$\mathbf{w}(x) = \min_{p \in \text{par}_{\mathcal{L}}(x)} \mathbf{w}(p)$$

and update the weight of $x \in \mathcal{C}$ to

$$\mathbf{w}(x) = \min_{p \in \text{par}_{\mathcal{L}}(x)} \mathbf{w}(p)$$

if $\mathbf{w}(x)$ was set to 1 in the second step.

Example 5.2 (Minimal timestamp). Transactions can carry additional information, for instance, timestamps. These can be used to decide between two conflicting transactions. Replacing the hashes by timestamps in Example 5.1 yields a weight function based on the timestamps of the transactions.

The two examples above make the most sense in a distributed system under the assumptions of eventual consistency. For applications in the DLT space, more efficient and robust weights are appropriate.

Algorithm 3: Reality selection in Branch DAG

Data: Branch DAG $D_{\mathcal{B}} = (\mathcal{B}, E)$
Result: reality $R \in \mathcal{B}$

```
1  $R \leftarrow \emptyset$  ; /* main branch in  $D_{\mathcal{B}}$  */
2 while  $R$  is not a leaf in  $D_{\mathcal{B}}$  do
3   |  $B^* \leftarrow \arg \max\{\mathbf{w}(B) : B \in \text{child}_{\mathcal{B}}(R)\}$  ; /* use min hash( $B \setminus R$ ) for breaking ties */
4   |  $R \leftarrow B^*$ 
5 end
```

Example 5.3 (External consensus). An external consensus protocol can determine the weights. For example, nodes could agree on the weights via additional direct communication and employ Byzantine-fault-tolerance mechanisms. Weights can also be inherited by the data structure that carries the ledger; for instance, $\mathbf{w}(x) = 1$ if x is contained in the longest chain, [27], or in the heaviest subtree, [37]; and $\mathbf{w}(x) = 0$ otherwise. Finer weights can be obtained using the distance between the longest and second-longest chain.

Example 5.4 (Approval weight). The monotonicity property, see Assumption 5.1, suggests that we can define the weights recursively using the underlying DAG structure of the DLT. This definition enables an internal consensus mechanism on the weights and, therefore, on the preferred reality. These ideas are expanded in detail in [26].

To determine which reality an operator of the ledger should prefer, we propose to perform the recursive exploration algorithm described in Algorithm 3. In this algorithm, we start at the main branch of the Branch DAG and walk on this graph until we reach a leaf. The algorithm prefers to go to the child with the highest value of the weight function. We observe that the resulting R is a maximal branch or a reality by construction. One can readily see that the provided algorithm has reasonable complexity despite the fact that the Branch DAG can be exponentially large in the number of conflicts (cf. Remark 4.8). Indeed, the number of iterations in the while-loop is bounded by the depth of the Branch DAG which is at most $|\mathcal{C}|$. The number of elements in $\text{child}_{\mathcal{B}}(R)$ is also bounded by $|\mathcal{C}|$. Thereby, the complexity of Algorithm 3 can be estimated as $O(|\mathcal{C}|^2)$. These observations are summarized below.

Proposition 5.1. *The resulting set R in Algorithm 3 is a reality. The complexity of this algorithm is $O(|\mathcal{C}|^2)$.*

As there is a one-to-one correspondence between realities and maximal independent sets of the Conflict Graph by Proposition 4.3, we propose an alternative reality selection procedure based on the Conflict Graph in Algorithm 4. In this algorithm, we start with the empty set and iteratively construct a subset R of conflicts. Specifically, we add a conflict to this set if this conflict is not conflicting with R and attains the highest value of the weight function. By construction, Algorithm 4 leads to a maximal independent set in the Conflict Graph or a reality in the Branch DAG. The

Algorithm 4: Reality selection in Conflict Graph

Data: Conflict Graph $G_{\mathcal{C}} = (\mathcal{C}, E)$
Result: reality $R \in \mathcal{B}$

- 1 $R \leftarrow \emptyset$
- 2 $U \leftarrow \mathcal{C}$
- 3 **while** $|U| \neq 0$ **do**
- 4 $c^* \leftarrow \arg \max\{\mathbf{w}(c) : c \in \max_{\mathcal{C}}(U)\}$; /* use minhash(c) for breaking ties */
- 5 $R \leftarrow R \cup \{c^*\}$
- 6 $U \leftarrow U \setminus \{N_{\mathcal{C}}(c^*) \cup \{c^*\}\}$
- 7 **end**

number of iterations in the while-loop is bounded by $|\mathcal{C}|$ and the number of $G_{\mathcal{C}}$ -neighbours is also bounded by $|\mathcal{C}|$. Thus, it is possible to implement this algorithm with complexity $O(|\mathcal{C}|^2)$. In the following statement we verify that the outcomes of the two algorithms coincide.

Theorem 5.1. *Algorithms 3 and 4 provide the same reality as output.*

Proof. The proof is done by induction on the number of iterations in the while-loops. Both algorithms start with the empty set and add at the first step the same conflict c^* , namely the one achieving the highest value of the weight function $\mathbf{w}(\cdot)$. Note that c^* is $D_{\mathcal{C}}$ -maximal in the set of conflicts \mathcal{C} and represents a child of the main branch in $D_{\mathcal{B}}$.

Let us assume that both algorithms constructed the same branch R after some number of steps. Then on one hand, Algorithm 4 will pick the conflict c^* with the highest value of $\mathbf{w}(\cdot)$ among all $D_{\mathcal{C}}$ -maximal elements in the set U , where U is the set of conflicts in $\mathcal{C} \setminus R$ that are not conflicting with R . On the other hand, Algorithm 3 will pick a branch B^* with the highest value of $\mathbf{w}(\cdot)$ over all children of R in $D_{\mathcal{B}}$. It remains to show that B^* obtained in Algorithm 3 coincides with $R \cup \{c^*\}$ obtained in Algorithm 4.

First, we prove that $R \cup \{c^*\}$ is a branch (cf. Definition 4.10). The set $R \cup \{c^*\}$ does not contain conflicting transactions since all transactions conflicting with R were removed from U at the previous steps and $c^* \in U$. Seeking a contradiction assume that $R \cup \{c^*\}$ is not $D_{\mathcal{C}}$ -past-closed. Since R is a branch by the inductive hypothesis, it may happen only when there exists some $b \in \text{cone}_{\mathcal{C}}^{(p)}(c^*)$ such that $b \notin R$. From $c^* \in U$ it follows that $b \in U$. Since $c^* \in \max_{\mathcal{C}}(U)$ and $c^* \leq_{\mathcal{C}} b$, we conclude that $c^* = b$. Thus, $R \cup \{c^*\}$ is indeed a branch which belongs to the set of children $\text{child}_{\mathcal{B}}(R)$ by Definition 4.11.

Since $B^* \in \text{child}_{\mathcal{B}}(R)$, the branch B^* can be represented as $d \cup R$ for some $d \in \mathcal{C}$. The set B^* is a branch and does not contain conflicting transactions and, thus, we have that $d \in U$. Moreover, $d \in \max_{\mathcal{C}}(U)$ since B^* is $D_{\mathcal{C}}$ -past-closed.

Now we will show that

$$\mathbf{w}(d) = \mathbf{w}(B^*). \tag{1}$$

By definition of the weight function $\mathbf{w}(B^*) = \min\{\mathbf{w}(c) : c \in d \cup R\}$. Seeking a contradiction,

assume that the minimum is attained at some $e \neq d$, i.e., $\mathbf{w}(e) < \mathbf{w}(d)$. Let U' and R' be the sets U and R right before the element e was included to R . Then one can find $d' \in \text{cone}_C^{(p)}(d)$ such that $d' \in \max_C(U')$. By the monotonicity of the weight function, we get that $\mathbf{w}(d') \geq \mathbf{w}(d) > \mathbf{w}(e)$ which contradicts the fact that e was included to U' and, thus, achieves the maximum of the weight function over all conflicts in $\max_C(U')$.

Using similar ideas as above one can prove that

$$\mathbf{w}(c^*) \leq \mathbf{w}(R). \quad (2)$$

Finally, it remains to show that $\mathbf{w}(d) = \mathbf{w}(c^*)$. By definition of c^* in Algorithm 4, $\mathbf{w}(c^*) \geq \mathbf{w}(d)$ as both c^* and d belong to $\max_C(U)$. On the other hand, by definition of B^* in Algorithm 3, $\mathbf{w}(B^*) \geq \mathbf{w}(c^* \cup R)$. Combining the latter inequality with (1)-(2) implies

$$\mathbf{w}(d) = \mathbf{w}(B^*) \geq \mathbf{w}(c^* \cup R) = \mathbf{w}(c^*),$$

which leads to $\mathbf{w}(d) = \mathbf{w}(c^*)$. To ensure that $d = c^*$ we observe that both algorithms use $\text{min hash}(\cdot)$ for breaking ties. \square

Example 5.5. In Figure 10, we depict an illustrative example that demonstrates how both reality selection algorithms work. We make use the same labeled UTXO DAG, the Conflict DAG and the Conflict Graph as in Figure 8. First, we note that there are three iterations in the while-loops of the both algorithms. Conflicts in Figure 10 are represented by colorful boxes. The value of the function $\mathbf{w}(\cdot)$ is depicted inside the boxes. The selected set of conflicts R at every step is highlighted by green borders. At the first step both algorithms include to R , which was initialized as the empty set, the yellow conflict which has the highest weight 0.7. Since the red conflict is conflicting with the yellow conflict, it should be removed from the set U . At the second step, Algorithm 3 takes the child of R in D_B that has the highest weight. This child is a branch that consists of two conflicts, the yellow one and the aquamarine one. At the same moment Algorithm 4 finds the conflict with the highest weight in the set of D_C -maximal elements of U . This set consists of the purple conflict and the aquamarine conflict and the latter has the highest weight. Finally, at last step, both algorithms update set R by adding the blue conflict.

5.3 Pruning Conflicts

In this section, we explain how we prune conflicting transactions and update the Conflict DAG, the Conflict Graph and the Ledger DAG when the weight function \mathbf{w} of a certain branch exceeds a given threshold.

Definition 5.2 (Confirmed branch). Let $\theta \in (0.5, 1]$ be a fixed threshold. A branch $B \in \mathcal{B}$ is called *\mathbf{w} -confirmed* if $\mathbf{w}(B) \geq \theta$.

Lemma 5.3. *Let $\mathcal{B}' \subseteq \mathcal{B}$ be the set of \mathbf{w} -confirmed branches. Then there exists a unique D_B -minimal branch in the set \mathcal{B}' . In other words, the subDAG of D_B induced by \mathcal{B}' has a unique leaf.*

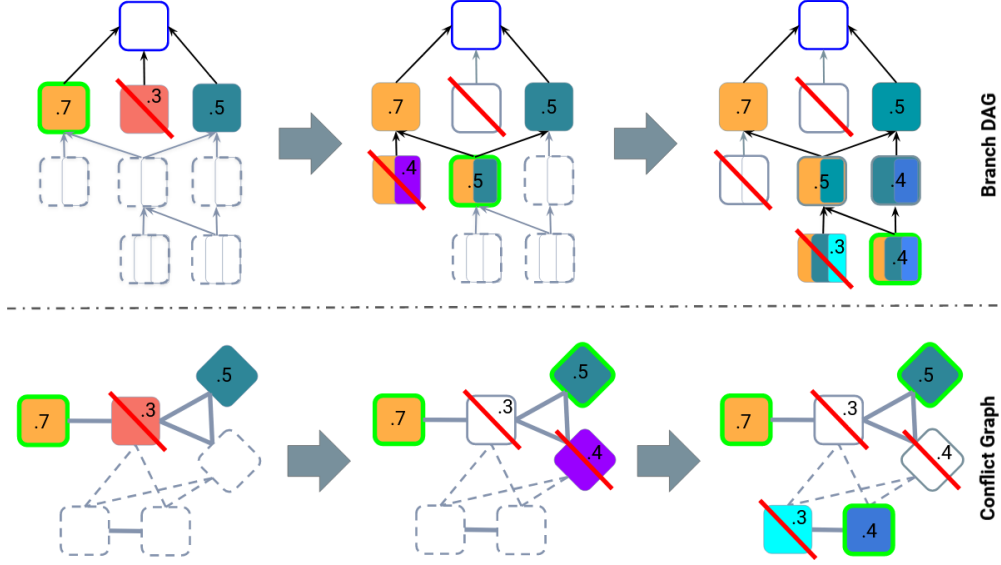


Figure 10: Demonstrative example of the reality selection algorithms based on the Branch DAG and the Conflict Graph.

Proof. Seeking a contradiction, assume the existence of two $D_{\mathcal{B}}$ -minimal confirmed branches B_1 and B_2 . If these branches are conflicting, then $\mathbf{w}(B_1) + \mathbf{w}(B_2) \leq 1$ by Remark 5.2 and we come to a contradiction as $\mathbf{w}(B_1) + \mathbf{w}(B_2) \geq 2\theta > 1$. Thus, the union of these branches does not contain a pair of conflicting transactions and $B := B_1 \cup B_2$ is also a branch. Moreover, B is \mathbf{w} -confirmed as

$$\mathbf{w}(B) = \min_{x \in B_1 \cup B_2} \mathbf{w}(x) \geq \min(\mathbf{w}(B_1), \mathbf{w}(B_2)) \geq \theta.$$

Clearly, $B <_{\mathcal{B}} B_1$ and $B <_{\mathcal{B}} B_2$ and we arrive to a contradiction with the fact that B_1 and B_2 are $D_{\mathcal{B}}$ -minimal in the set of confirmed branches \mathcal{B}' . \square

Let B be the unique $D_{\mathcal{B}}$ -minimal \mathbf{w} -confirmed branch as in Lemma 5.3. We propose to update all graph structures such that the conflicts in B are no longer conflicts, i.e., all transactions conflicting with B should be removed from the ledger \mathcal{L} . A possible way to achieve that is described in Algorithm 5.

In this algorithm, we first identify the set of conflicts \mathcal{C}' conflicting with B and the set of conflicts \mathcal{C}'' that should be removed from \mathcal{C} . Note that $B \cup \mathcal{C}' \subseteq \mathcal{C}''$. Then we

1. remove all edges adjacent to \mathcal{C}' from the Conflict Graph $G_{\mathcal{C}}$. We note that \mathcal{C}'' is the set of isolated vertices in $G_{\mathcal{C}}$. We remove \mathcal{C}'' from $G_{\mathcal{C}}$.
2. prune the Ledger DAG by removing all transactions that are directly or indirectly conflicting with B and all edges adjacent to them by traversing future cones of \mathcal{C}' ;

3. remove all vertices corresponding to \mathcal{C}'' and all edges adjacent to \mathcal{C}'' from the Conflict DAG. If necessary, we add some edges to the Conflict DAG to make it connected.

Recall that the Conflict DAG and the Conflict Graph are concepts which are derived from the Ledger DAG, i.e., $D_{\mathcal{C}} = D_{\mathcal{C}}(D_{\mathcal{L}})$ and $G_{\mathcal{C}} = G_{\mathcal{C}}(D_{\mathcal{L}})$. Thereby, one needs to show consistency between the resulting three graphs.

Lemma 5.4. *The resulting Conflict DAG and the resulting Conflict Graph are consistent with the resulting Ledger DAG in Algorithm 5.*

Proof. By Proposition 4.2, to identify the set of all conflicts that are conflicting with B , it suffices to consider the $D_{\mathcal{C}}$ -minimal elements in B . In Algorithm 5, we set M to be the set of these minimal conflicts. Then we construct the set $\mathcal{C}' \subseteq \mathcal{C}$ of conflicts that are conflicting with branch B by looking at the neighbours of M in the Conflict Graph. Similar to Proposition 4.1, one can check that the set \mathcal{C}' is $D_{\mathcal{C}}$ -future-closed. Let $\mathcal{L}' \subseteq \mathcal{L}$ denote the set of transactions that are conflicting with B . By Proposition 4.1, \mathcal{L}' is $D_{\mathcal{L}}$ -future-closed and $\max_{\mathcal{L}}(\mathcal{L}') = \max_{\mathcal{C}}(\mathcal{C}')$. In Algorithm 5, we

1. remove all edges adjacent to \mathcal{C}' from the Conflict Graph and construct the set \mathcal{C}'' of isolated vertices in the updated Conflict Graph. Clearly, all conflicts in \mathcal{C}'' are no longer conflicts;
2. update the Ledger DAG by recursive traversing the Ledger future cone of $\max_{\mathcal{C}}(\mathcal{C}')$ and removing transactions \mathcal{L}' and all edges adjacent to them;
3. update the Conflict DAG by removing all conflicts \mathcal{C}'' from \mathcal{C} and all edges adjacent to them. It is possible that after performing this step some vertices in $\mathcal{C} \setminus \mathcal{C}''$ have out-degree zero. To make the Conflict DAG connected again, we add edges from all such conflicts to the genesis ρ .

□

We conclude this section with a sufficient condition for the pruned data structure to be again conflict-free.

Theorem 5.2. *Let \mathcal{L} be a ledger and B be a \mathbf{w} -confirmed reality. Then, the set of transactions in the pruned Ledger DAG resulting from Algorithm 5 is conflict-free and coincides with the B -ledger $\mathcal{L}_r(B)$ as in Definition 4.15.*

Proof. Suppose B is a reality. Then, we note that the set of conflicts to be removed from \mathcal{C} is $\mathcal{C}'' = \mathcal{C}$. Indeed, since B is a maximal independent set in the Conflict Graph (cf. Proposition 4.3), all conflicts in $\mathcal{C} \setminus B$ are conflicting with B and has to be removed from the Ledger DAG. Thus, the conflicts in B are no longer conflicts and become ordinary transactions. Thus, the resulting set of transactions does not contain a pair of directly conflicting transactions. In addition, all transactions x such that $\text{label}^{(p)}(x) \subseteq \text{label}^{(p)}(B)$ are kept in the set of transactions. □

6 Numerical Experiments

We have implemented a standalone program⁴ in Go 1.20 that provides several functionalities described in the paper. We have conducted all benchmarks on Windows 11 with a single CPU of Intel Core i7-11370H with 3.30GHz, and 8GB of memory. The simulation results presented in this section provide a lower bound on the number of transactions that can be processed by a node with hardware of this kind, provided that one CPU is dedicated to managing a reality-based ledger and the reality-based ledger is completely stored in RAM. While it is natural to parallelize transaction processing for a reality-based ledger, the implementation aspect of this question is beyond the scope of the paper. We would like to note that our benchmarks only address updating all basic data structures and do not include signature checks or transaction executions. In our numerical experiment, we have generated a stream of pseudo-random UTXO transactions that meet specific criteria. Specifically, we have used the following guidelines:

- The number of inputs for a new transaction is randomly sampled from a uniform distribution on the set $\{1, 2\}$.
- For non-conflicting transactions, inputs are selected randomly and uniformly from the set of all unspent outputs.
- A new transaction becomes conflicting (or directly conflicting with an existing transaction) with a given probability of $p_{\text{conflict}} \in \{0.01, 0.05, 0.1, 0.5\}$. To accomplish this, we randomly select one input label from the set of already consumed outputs and the remaining input (if there is one) from the unspent outputs.
- The number of outputs for a new transaction is randomly sampled from a uniform distribution on the set $\{1, 2, 3\}$, with each new output created as a hash digest (using SHA256) of a random value.
- The genesis has 16 outputs and 0 inputs.

It is expected that the number of conflicts using this model will fall between Np_{conflict} and $2Np_{\text{conflict}}$ with high probability, where N is the total number of transactions. This is because a new transaction can cause a previously non-conflicting transaction to directly conflict with the new one.

Remark 6.1. *Let us comment on the choice of the above values. We investigated all transactions, around 300k, of the Shimmer main net from 2022/09/27 to 2023/01/02. In Figure 11 we present the two-dimensional empirical distribution of in and out-degrees of the UTXO transactions. We can make three main observations. First, around 95% of the transactions use not more than 2 inputs and not more than 3 outputs. Second, we can observe a “horizontal line” with two outputs and inputs varying from 1 to 128. This effect can be explained by a wallet functionality that tries to keep the number of unspent outputs as small as possible. Third, there is a “vertical” line using 3 inputs, and*

⁴available at <https://github.com/nikitapolyskii/reality-ledger>

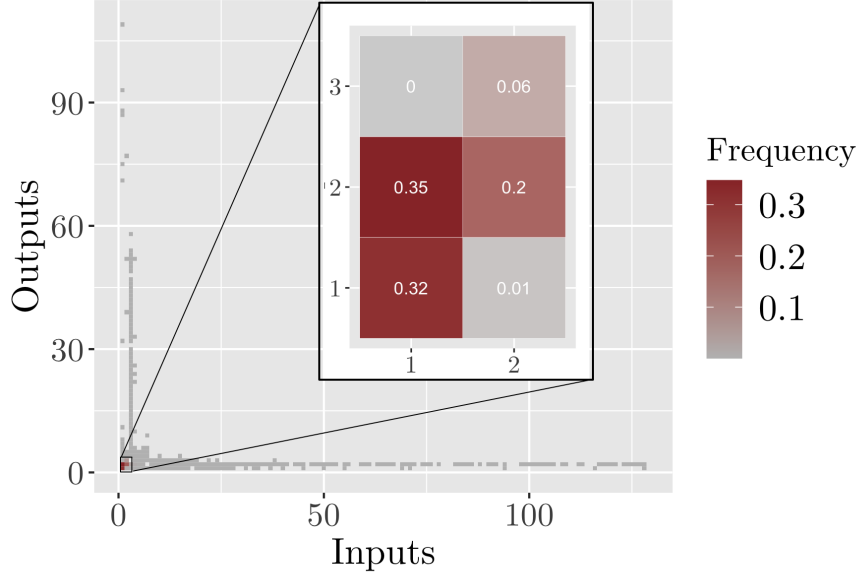


Figure 11: Empirical distribution of the number of inputs and outputs in the Shimmer network (from 2022/09/27 to 2023/01/02).

this effect can be explained by using a special output type called “alias-output” for the minting of NFTs. These observations show that the actual distribution of inputs and outputs heavily relies on different uses-cases and functionalities. For this reason, we choose to model the distribution of the number of inputs and outputs as a uniform distribution, the one with the highest entropy, on the typical “input-output” relation. Moreover, the observed conflict rate is $p_{\text{conflict}} \approx 0.03$, and we use the two values 0.01 and 0.05 to get bounds on the “likely” behaviour. The choices of $p_{\text{conflict}} \in \{0.1, 0.5\}$ are motivated to cover scenarios where an attacker spam conflicts.

To store and update the main data structures, we utilize map containers. For each transaction x , we store (and update when necessary) the following fields in a reality-based ledger:

- x .InputLabels, a list of inputs $\text{in}(x)$ from the set of labels $\{0, 1\}^{256}$;
- x .OutputLabels, a list of outputs $\text{out}(x)$ from the set of labels $\{0, 1\}^{256}$;
- x .Parents, a list of parents $\text{par}_{\mathcal{L}}(x)$, transactions whose outputs are spent by x ;
- x .Children, a list of children $\text{child}_{\mathcal{L}}(x)$, transactions that consumes the output(s) of x ;
- x .ParentsConflicts, a list of the closest conflicts $\text{conf}_{\mathcal{L}}^{(p)}(x)$ in the past cone of x ;
- x .ChildrenConflicts, a list of the closest conflicts $\text{conf}_{\mathcal{L}}^{(f)}(z)$ in the future cone of x ;

- x .DirectConflicts, a list of transactions that are directly conflicting with x
- x .InputConflictLabels, a list of inputs that are consumed with some other transactions (directly conflicting with x)

Storing branches associated with each transaction and conflict might increase the storage overhead by a factor of the number of conflicts. Thereby, for a transaction $x \in \mathcal{L}$, we have implemented a function `GETBRANCH(\cdot)` that returns $\text{label}^{(p)}(x)$, all conflicts in the past cone of x . Note that the fields `.ParentsConflicts` and `.ChildrenConflicts` allow traversing a DAG on the set of conflicts. Denote this DAG (the DAG on the set of conflicts whose edges are defined by `.ParentsConflicts` and `.ChildrenConflicts`) as $\hat{D}_{\mathcal{C}}$. We note that $\hat{D}_{\mathcal{C}}$ contains all the edges of the Conflict DAG $D_{\mathcal{C}}$ and some other extra edges, but the reachability properties of $D_{\mathcal{C}}$ are preserved in $\hat{D}_{\mathcal{C}}$, i.e., $D_{\mathcal{C}}$ is a transitive reduction of $\hat{D}_{\mathcal{C}}$.

We have implemented a function `GETREALITY(\cdot)` that returns the same reality as Algorithms 3-4 do for the case when the weights of all transactions are zeros. Storing and updating the Branch DAG and the Conflict Graph is impractical for a large number of conflicts since their sizes can be exponential and quadratic in the number of conflicts (see Remark 4.8). Instead, the structure of the DAG $\hat{D}_{\mathcal{C}}$ is utilized in our implementation. Specifically, we iteratively construct the reality (similar to Algorithm 4): at every step, we choose c^* that is $D_{\mathcal{C}}$ -maximal and attains the minimal hash (see line 3). We have also implemented a function `PRUNEREJECTEDTRANSACTIONS(\cdot)` that takes as input a reality and prunes all transactions conflicting with at least one conflict from the preferred reality. This function works similarly to Algorithm 5 when the latter takes as an input the reality obtained from Algorithms 3-4.

In Figure 12, for different probabilities $p_{\text{conflict}} \in \{0.01, 0.05, 0.1, 0.5\}$, we depict the growth of a reality-based ledger over time with and without computing `GETBRANCH(\cdot)` for each new transaction. We employ the model of a randomized stream of transactions described above. The number of transactions is limited by 8GB of RAM allocated to the reality-based ledger. The worst performance corresponds to the case when conflicts are located sparsely in the ledger ($p_{\text{conflict}} = 0.01$). The latter can be explained by the fact that for $p_{\text{conflict}} = 0.01$, one needs to traverse more vertices in the Ledger DAG on average to update the fields `.ChildrenConflicts` and `.ParentsConflicts` for transactions in the future and past cones of a new conflicting transaction. Specifically, for each non-conflicting transaction, there could be some paths between the closest conflicts that contain the transaction; one needs to traverse through that transaction in the Ledger DAG as many times as one of such paths is updated. The computational complexity of `GETBRANCH(\cdot)` heavily depends on the height of the closest conflicts of a given transaction in $D_{\mathcal{C}}$. For instance, in the considered model, the expected height of a random conflict in $D_{\mathcal{C}}$ is asymptotically logarithmic with the total number of conflicts. This is seen in Figure 12 as there is no significant performance degradation when one additionally computes the branch for each new transaction by calling `GETBRANCH(\cdot)`. In all cases, the rate of transactions per second is 50,000 – 150,000 tx/sec.

In Figure 13, we show the statistics of the time that it takes to compute `GETREALITY(\cdot)` when the number of conflicts $|\mathcal{C}| \in \{10000, 20000, 40000\}$. To generate conflicts, we utilize the same model of randomized transactions with $p_{\text{conflict}} = 0.05$. Recall that, by Proposition 5.1, the complexity of

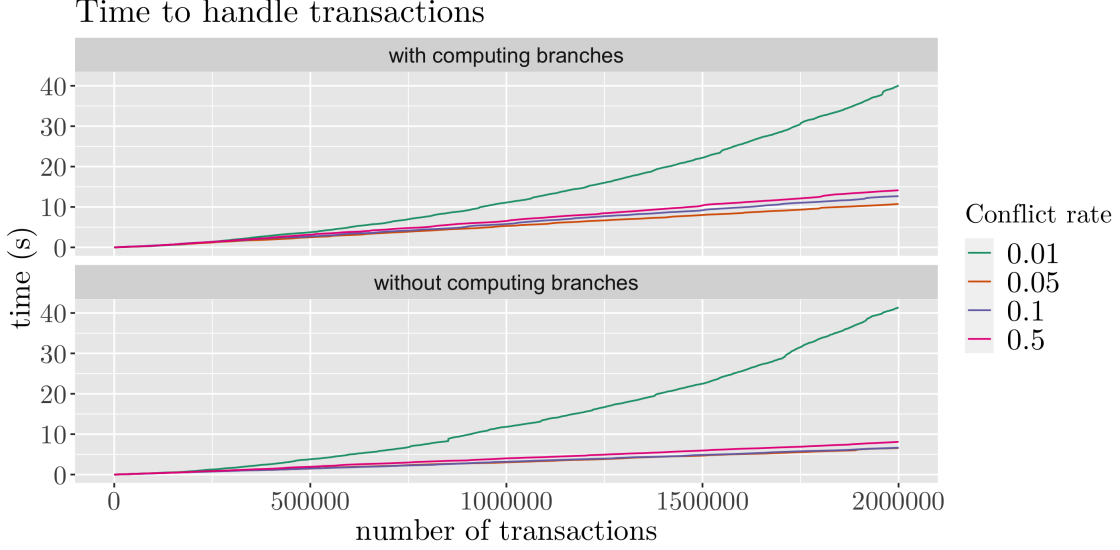


Figure 12: Time to handle the incoming flux of transactions, with and without computing `GETBRANCH(·)` for every transaction. The benchmark is taken for different probabilities $p_{\text{conflict}} \in \{0.01, 0.05, 0.1, 0.5\}$.

the reality selection algorithm in the worst case can be bounded as $O(|\mathcal{C}|^2)$ if one follows Algorithm 3-4. In a more practical implementation that we employ in our simulation, the worst-case complexity is even worse as finding $D_{\mathcal{C}}$ -maximal elements and the element with the largest hash among the maximal elements in a list are not necessarily $O(1)$ -operations. However, in Figure 13, the amortized complexity of our benchmarks seems linear with the number of conflicts $|\mathcal{C}|$. We can observe a concentrated distribution for 10,000 conflicts and that the distribution flattens as the number of conflicts increases. Interestingly, the eventual size of the preferred reality is more robust in the increase of conflicts; see Figure 14.

As the time to calculate the preferred reality increases with the number of conflicts, we periodically remove confirmed transactions from our data structures. In Figure 15, we depict the growth of a reality-based ledger, the number of confirmed transactions and conflicts over time. We generate a stream of transactions with parameter $p_{\text{conflict}} = 0.01$ (the case providing the worst performance in Figure 12). Whenever the number of conflicts exceeds the set upper limit of 5,000, we apply the functions `GETREALITY(·)` and `PRUNEREJECTEDTRANSACTIONS(·)`. In addition, we treat all remaining transactions as confirmed and update the data structure of the reality-based ledger by taking the confirmed ledger state as a new genesis. We highlight that the number of pruned transactions is much larger than that of pruned conflicts since a transaction that indirectly spends from an output of a pruned conflict must also be pruned. In summary, by periodically pruning rejected transactions and snapshotting confirmed transactions, the protocol allows handling an ongoing stream of transactions without performance loss. The rate of confirmed transactions per second in this simulation

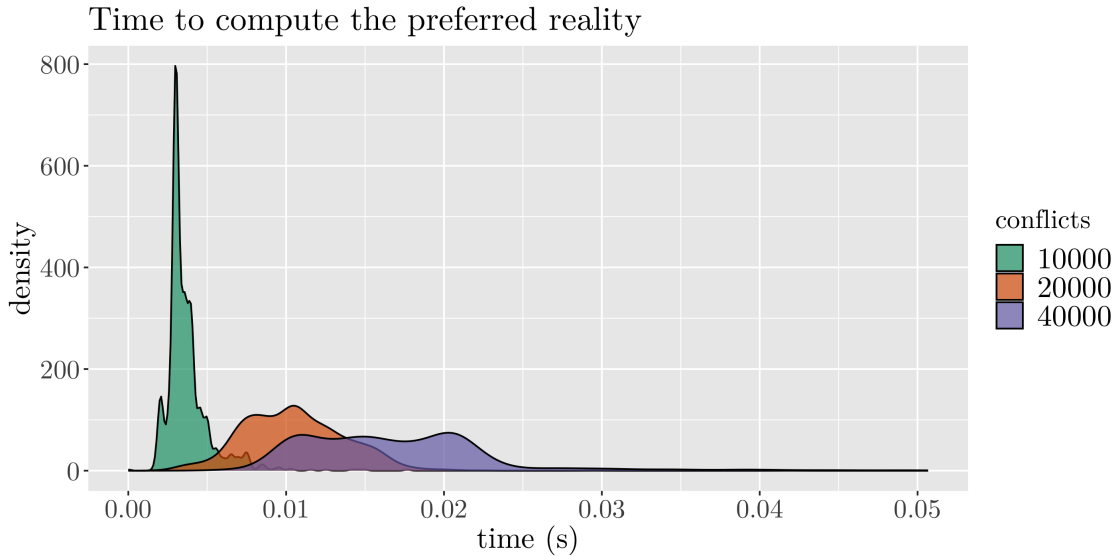


Figure 13: Time to compute the preferred reality for different numbers of conflicts.

is over 70,000 txs/sec. This experiment indicates the robustness and scalability of our proposed reality-based UTXO ledger for real-world use cases.

7 Future Work

The Reality-based Ledger provides a framework for parallel transaction processing capability. Typically in blockchains, transactions are processed in blocks or batches, creating a total order. This linearisation creates an artificial bottleneck in the propose and vote paradigm of DLTs. However, particularly in a UTXO setting, this is not necessary, and blockchain systems can be designed using the presented framework. We follow this approach in [26], which builds on the foundations laid out in this paper.

We provided a first quantitative analysis of our proposed algorithms. Future work in this area should focus on benchmarking the performance of different DAG-based DLTs against each other. This could involve comparing throughput by measuring different kinds of transactions, e.g. simple value transfers or transactions of smart contracts, as well as assessing the potential for quick view changes. Additionally, it is essential to consider the manipulation potential in the various systems, including the possibility of annulling or frontrunning transactions. This future work must also address the challenge of constructing appropriate and meaningful performance measures and test scenarios for the different DLT solutions, as their distinct natures and other underlying use cases will impact the results. These benchmarks will ensure a fair and more comprehensive evaluation of the strengths and limitations of various DAG-based DLT solutions.

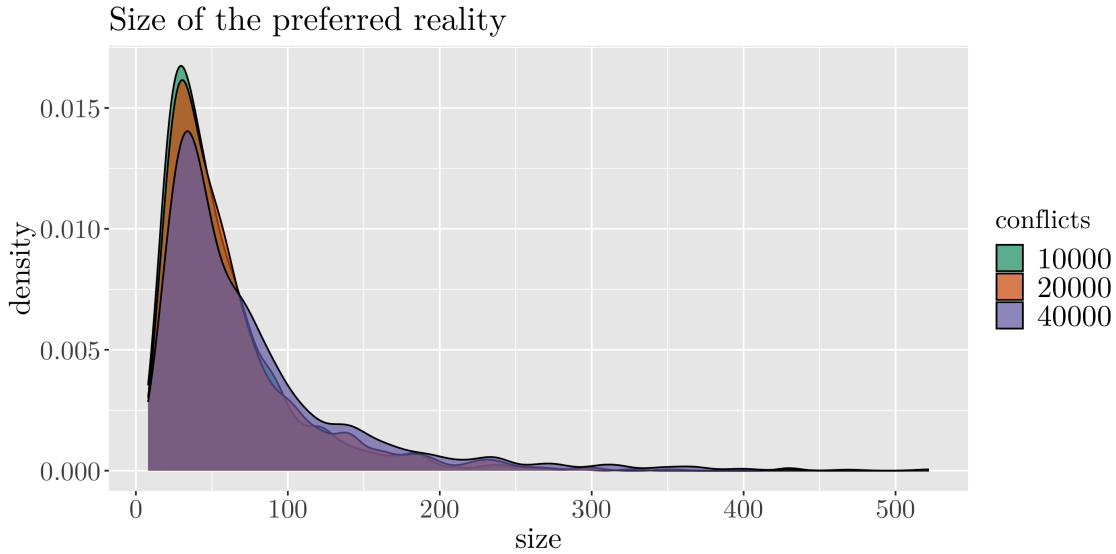


Figure 14: Size of the preferred reality for different numbers of conflicts.

Acknowledgments

The authors would like to thank precious staff members of the IOTA Foundation and members of the IOTA community for their feedback and criticism.

References

- [1] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. Optsmart: a space efficient optimistic concurrent execution of smart contracts. *Distributed and Parallel Databases*, 2022.
- [2] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, page 585–602, New York, NY, USA, 2019. Association for Computing Machinery.
- [3] L. Baird and A. Luykx. The Hashgraph Protocol: Efficient Asynchronous BFT for High-Throughput Distributed Ledgers. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–7, 2020.
- [4] M. Belotti, N. Božić, G. Pujolle, and S. Secci. A Vademecum on Blockchain Technologies: When, Which, and How. *IEEE Communications Surveys Tutorials*, 21(4):3796–3838, 2019.

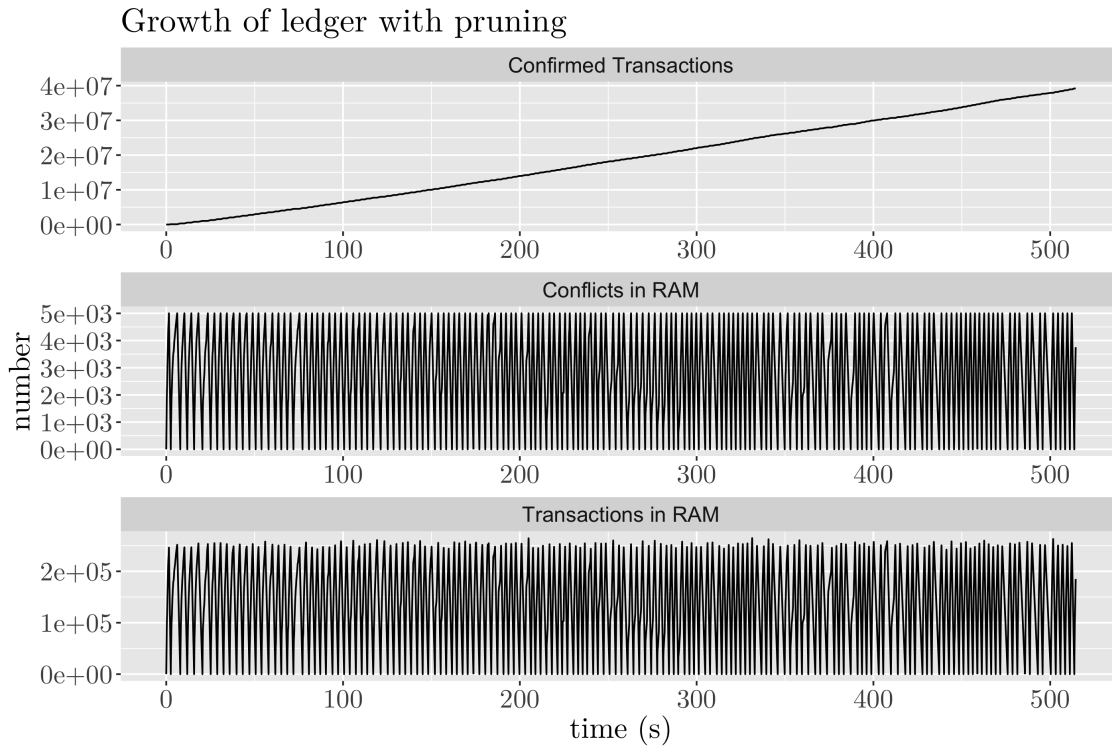


Figure 15: Growth of the ledger and number of transactions and conflicts kept in RAM.

- [5] Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform, 2013.
- [6] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The Extended UTXO Model. In Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 525–539, Cham, 2020. Springer International Publishing.
- [7] Duncan Coutts. Implement Ouroboros Leios to increase Cardano throughput, 2022.
- [8] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 34–50, New York, NY, USA, 2022. Association for Computing Machinery.
- [9] Johan de Kleer. An assumption-based tms. *Artificial Intelligence*, 28(2):127–162, 1986.

- [10] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 303–312, New York, NY, USA, 2017. Association for Computing Machinery.
- [11] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. Untangling Blockchain: A Data Processing View of Blockchain Systems. *IEEE Transactions on Knowledge and Data Engineering*, 30(7):1366–1385, 2018.
- [12] E. Dr̄sutis. IOTA Smart Contracts, accessed January 2022.
- [13] Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: Asynchronous BFT Made Practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2028–2041, New York, NY, USA, 2018. Association for Computing Machinery.
- [14] Adam Gagol, Damian Leśniak, Damian Straszak, and Michał Świątek. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 214–228, 2019.
- [15] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing, 2022.
- [16] Neil Giridharan, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Bullshark: DAG BFT protocols made practical. *CoRR*, abs/2201.05677, 2022.
- [17] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All You Need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 165–175, New York, NY, USA, 2021. Association for Computing Machinery.
- [18] Idit Keidar, Oded Naor, and Ehud Shapiro. Cordial miners: Fast and efficient consensus for every eventuality, 2022.
- [19] Sergio Demian Lerner. DagCoin: a cryptocurrency without blocks., 2015.
- [20] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.
- [21] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014. USENIX Association.
- [22] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *10th*

- {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12), pages 265–278, 2012.
- [23] Chenxing Li, Fan Long, and Guang Yang. Ghost: Breaking confirmation delay barrier in nakamoto consensus via adaptive weighted blocks. *ArXiv*, abs/2006.01072, 2020.
- [24] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 31–42, New York, NY, USA, 2016. Association for Computing Machinery.
- [25] Mysten Lab. The Sui Smart Contracts Platform, 2022.
- [26] Sebastian Müller, Andreas Penzkofer, Nikita Polyanskii, Jonas Theis, William Sanders, and Hans Moog. Tangle 2.0 leaderless nakamoto consensus on the heaviest dag. *IEEE Access*, 10:105807–105842, 2022.
- [27] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
- [28] People on nxtforum.org. DAG, a generalized blockchain, 2014.
- [29] Serguei Popov. The Tangle, 2015.
- [30] Serguei Popov, Hans Moog, Darcy Camargo, Angelo Caposelle, Vassil Dimitrov, Alon Gal, Andrew Greve, Bartosz Kusmierz, Sebastian Mueller, Andreas Penzkofer, Olivia Saa, William Sanders, Luigi Vigneri, Wolfgang Welz, and Vidal Attias. The Coordicide. 2019.
- [31] Maria A. Schett and George Danezis. Embedding a Deterministic BFT Protocol in a Block DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing, PODC'21*, page 177–186, New York, NY, USA, 2021. Association for Computing Machinery.
- [32] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [33] Yonatan Sompolsky, Yoad Lewenberg, and Aviv Zohar. Spectre: A fast and scalable cryptocurrency protocol. Cryptology ePrint Archive, Report 2016/1159, 2016.
- [34] Yonatan Sompolsky and Michael Sutton. The dag knight protocol: A parameterless generalization of nakamoto consensus. Cryptology ePrint Archive, Paper 2022/1494, 2022. <https://eprint.iacr.org/2022/1494>.
- [35] Yonatan Sompolsky, Shai Wyborski, and Aviv Zohar. *PHANTOM GHOSTDAG: A Scalable Generalization of Nakamoto Consensus: September 2, 2021*, page 57–70. Association for Computing Machinery, New York, NY, USA, 2021.

- [36] Yonatan Sompolinsky and Aviv Zohar. Accelerating bitcoin’s transaction processing. fast money grows on trees, not chains. *IACR Cryptol. ePrint Arch.*, 2013:881, 2013.
- [37] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In Rainer Böhme and Tatsuaki Okamoto, editors, *Financial Cryptography and Data Security*, pages 507–527, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
- [38] Gerui Wang, Shuo Wang, Vivek Bagaria, David Tse, and Pramod Viswanath. Prism removes consensus bottleneck for smart contracts. In *2020 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 68–77, 2020.
- [39] Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. Sok: Diving into dag-based blockchain systems. *ArXiv*, abs/2012.06128, 2020.
- [40] Lei Yang, Vivek Bagaria, Gerui Wang, Mohammad Alizadeh, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Scaling bitcoin by 10,000x. *arXiv preprint arXiv:1909.11261*, 2019.
- [41] Haifeng Yu, Ivica Nikolic, Ruomu Hou, and P. Saxena. Ohie: Blockchain scaling made simple. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 90–105, 2018.

Algorithm 5: Algorithm to update Ledger DAG, Conflict DAG and Conflict Graph after branch confirmation

Data: Ledger DAG $D_{\mathcal{L}} = (\mathcal{L}, E_{D_{\mathcal{L}}})$, Conflict DAG $D_{\mathcal{C}} = (\mathcal{C} \cup \{\rho\}, E_{D_{\mathcal{C}}})$, Conflict Graph $G_{\mathcal{C}} = (\mathcal{C}, E_{G_{\mathcal{C}}})$, confirmed branch B

Result: updated Ledger DAG $D_{\mathcal{L}} = (\mathcal{L}, E_{D_{\mathcal{L}}})$, Conflict DAG $D_{\mathcal{C}} = (\mathcal{C} \cup \{\rho\}, E_{D_{\mathcal{C}}})$, Conflict Graph $G_{\mathcal{C}} = (\mathcal{C}, E_{G_{\mathcal{C}}})$

```

1  $M \leftarrow \min_{\mathcal{C}}(B)$ 
2  $\mathcal{C}' \leftarrow \emptyset$ ;
3  $\mathcal{C}'' \leftarrow \emptyset$ ;
4 for  $\forall x \in M$  do
5   for  $\forall y \in N_{\mathcal{C}}(x)$  do
6      $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{y\}$ 
7     for  $\forall z \in N_{\mathcal{C}}(y)$  do
8        $E_{G_{\mathcal{C}}} \leftarrow E_{G_{\mathcal{C}}} \setminus \{(y, z)\}$ 
9     end
10  end
11 end
12 for  $\forall x \in \mathcal{C}$  do
13   if  $x$  is isolated in  $G_{\mathcal{C}}$  then
14      $\mathcal{C}'' \leftarrow \mathcal{C}'' \cup \{x\}$ 
15   end
16 end
17 for  $\forall y \in \max_{\mathcal{C}}(\mathcal{C}')$  do
18   for  $\forall z \in \text{cone}_{\mathcal{L}}^{(f)}(y)$  do
19      $\mathcal{L} \leftarrow \mathcal{L} \setminus \{z\}$ 
20     for  $\forall p \in \text{par}_{\mathcal{L}}(z)$  do
21        $E_{D_{\mathcal{L}}} \leftarrow E_{D_{\mathcal{L}}} \setminus \{(z, p)\}$ 
22     end
23   end
24 end
25 for  $\forall y \in \mathcal{C}''$  do
26   for  $\forall p \in \text{par}_{\mathcal{C}}(y)$  do
27      $E_{D_{\mathcal{C}}} \leftarrow E_{D_{\mathcal{C}}} \setminus \{(y, p)\}$ 
28   end
29   if  $y \in \max_{\mathcal{C}}(\mathcal{C}'')$  then
30     for  $\forall c \in \text{child}_{\mathcal{C}}(y)$  do
31        $E_{D_{\mathcal{C}}} \leftarrow E_{D_{\mathcal{C}}} \setminus \{(c, y)\}$ 
32       if  $\text{par}_{\mathcal{C}}(c) \subseteq \mathcal{C}''$  then
33          $E_{D_{\mathcal{C}}} \leftarrow E_{D_{\mathcal{C}}} \cup \{(c, \rho)\}$ 
34       end
35     end
36   end
37    $\mathcal{C} \leftarrow \mathcal{C} \setminus \{y\}$ 
38 end

```