Universal Token Forest

Mojtaba Eshghie, Gustav Andersson Kasche

October 2022

Abstract

This work describes a non-monolithic blockchain-based data structure called Universal Token Forest (UTF). UTF can act as the blockchain-based infrastructure for circular supply chains. This data structure will enable the tokenization of products (physical/virtual) in a hierarchical structure. It makes it possible to create a token on the blockchain for the product that not only keeps the product passport, but also corresponds to the building blocks of the product itself using a tree-based data structure. The tokens in UTF reflect the product accurately to the extent that in the same way a product can be changed, disassembled, and refurbished they also do. This hierarchy is reflected in the on-chain data structure itself with the help of the smart contract-based design. Furthermore, the uniqueness of UTF is manifested in its compact yet non-monolithic architecture, mutability, and its two-layer action authorization scheme.

1 Introduction

Collecting, refurbishing, and reusing products or product parts has environmental and socio-economic benefits. This closed loop of reuse and recycling that aims to minimize resource consumption is called a circular economy [1]. Figure 1 shows an example of a circular economy where part of a product can also be part of another life cycle when the original product is disposed of.
To the best of our knowledge, there is no practical design and implementation of a blockchain-based infrastructure that fulfills the real-world requirements (requirements that section 1.1) of the circular economy. UTF is an answer to the problems in the hierarchical tokenization of products.

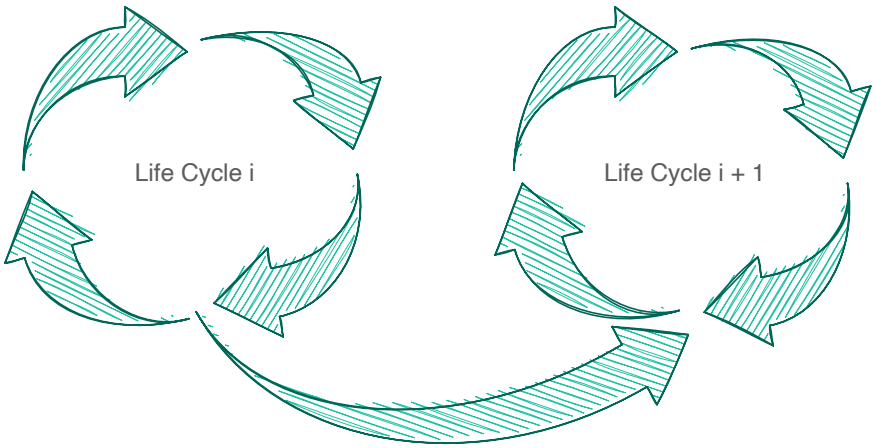

Figure 1: Use of a product part in multiple life-cycles

1.1 Requirements

For a blockchain-based solution to be useful for keeping the product passport as well as manage it through multiple product lifecycles, we identified a few requirements:

1. **Mutable structure**: Since products or product parts might live through multiple life cycles, every single bond in the system should be breakable.
2. **Non-monolithic**: We define a monolithic approach as using one single smart contract to manage a token tree or the whole forest (look at section 3). Such structure does not fit real-world usages because of the following reasons:
    - Each smart contract has limitations regarding the number of Algorand Standard Assets (ASAs) it can list in the global state variables.
    - Because the Algorand ASA primitive does not directly support a tree-based structure, it should be implemented using the logic inside a contract. A slight flaw in such logic would put the whole ecosystem at risk. It is therefore desirable to isolate token trees or token nodes from each other or logic operations from each other to contain any possible flaw or vulnerability.
3. **Managing authority**: The authority changing the token tree structure should be mutable. By default authorization of any action on sub-products

goes through the root of the token tree. However, it is desirable also to be able to modify products through roots further down in the tree structure. manager addresses in a tree should be the same so that a product can only be broken down by the actual manager, not by arbitrary people managing the sub-product.

4. **Delegates**: The structure needs to support delegated organizations. Delegates do not own the token. However, they can be authorized to perform certain actions on behalf of the product/token owner.

5. **Product passport**: All the product-related information is held within the smart contract owning the ASA. Furthermore, desirable physical sub-product that are the building blocks of the physical product may also include such information in their respective smart contract.

2 System Architecture

This section will motivate and describe the architecture of the UTF system. It will also briefly describe how the system could be monetized.

2.1 API

The system architecture is defined to facilitate universal and simple use by organizations outside of the blockchain ecosystem without any engagement with blockchain-specific technology. Therefore the system core is an API where all the functions needed to build, mutate and manage the data structure of UTF on-chain are exposed. How the user interacts with our API is completely up to them as seen in 2. This gives the user the freedom to govern their use of UTF as their industry sector demands. Various organizations may have further requirements that they can define on or off-chain. Our architecture gives the organization complete freedom over this. When the organization invokes a function that requires on-chain authentication, it will use a wallet manager. This greatly simplifies the UX of the system since no knowledge of Algorand or any blockchain is necessary to use UTF. All the user needs to do is request the API for the UTF actions they wish to perform, and the API will set up every transaction needed for UTF on-chain and simply request the user for the sign using their private key in their wallet.

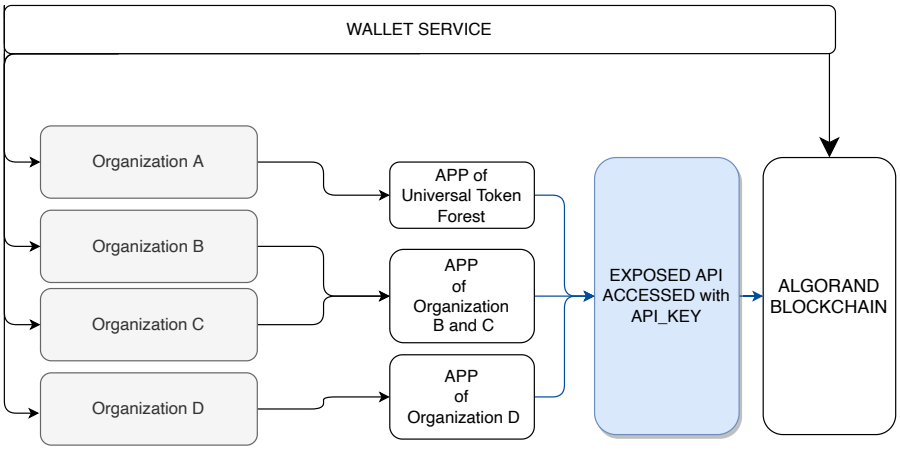

Figure 2: Example System architecture

2.2 API monetizing

As will be articulated in section 4, only basic tree-management operations are on-chain by the logic of the smart contracts. The more complex operations including keeping track of tokens can be done off-chain. Apart from a design choice to build a less complex system, this can also lead to a business model. The API is also used to monetize the system. API will be accessed using API keys associated with each organization. This gives us complete freedom in how we choose to monetize the off-chain services built on top of UTF. We can bill the users for each API invocation or periodically for API access.

3 The Data Structure

Definition 1. ***Tree****: A tree is defined as a data structure consisting of one or multiple nodes connected without a cycle. The nodes in a tree are either tree root, internal nodes, or leaves. An internal node may have one or more child nodes and is called the parent of its children. Leaves have only a connection to their parent [2].*

Definition 2. ***Forest****: A forest is a collection of one or more trees. The trees in this collection need not to be connected [2].*

Based on the definition mentioned above, Universal Token Forest comprises trees (built by tokens, hence, token trees) that can change shape and reform the forest. A visualization of UTF is depicted in figure 3. The bonds in this shape are shown using an arrow from parent to child. This bond corresponds to the relationship between product parts of the respective product.

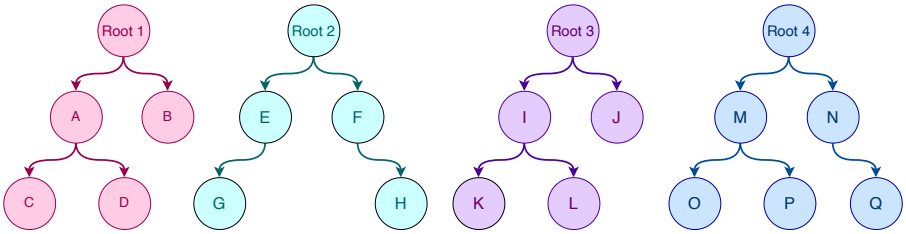

Figure 3: Simple forest of token trees denoting different products, their product parts, and the hierarchy of the physical product.

Operations on UTF are defined to reflect real-world product management requirements in a circular supply chain. Therefore, reshaping and reforming the trees in the forest corresponds to disassembling, assembling, refurbishing, or disposing of products in the real world. Figure 4 shows how a part (node K) from a product (root 3) can be separated and become part of another product (root 2).

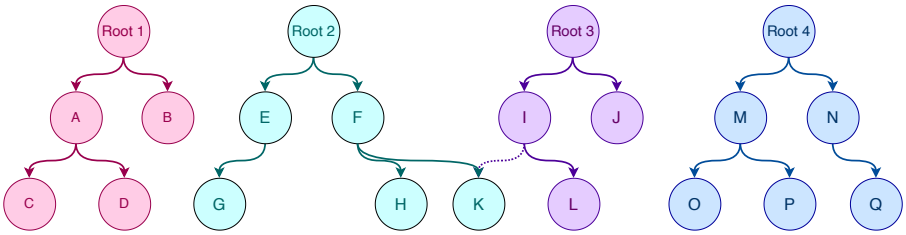

Figure 4: Breaking the bonds between one product part K and its original product token (with root 3) and build a new bond to another token tree (with root 2) under sub-product F.

Each node in figure 4 consists of an Algorand smart contract [3], and can also be associated with a particular ASA. It is worth mentioning that the root of each tree can identify stand-alone products. In this order, if a user on Algorand blockchain holds the ASA related to a tree root, it means that the user is the owner of the associated product. Figure 5 shows the parts of a typical tree node. Since each node is a smart contract, it can have logic. This logic is used for reshaping the tree structure.

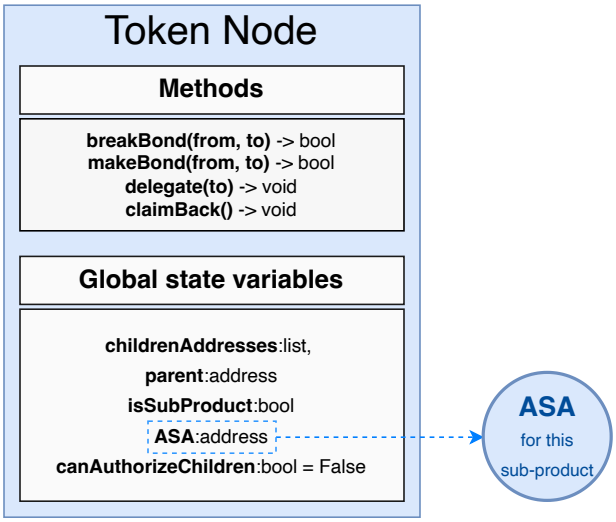

Figure 5: High-level view of each node of the tree in the system.

3.1 The Two Layer Authorization Scheme

A combination of ASA and contract logic empowers creating of a two-layer authorization scheme for UTF. As it is depicted in figure 6 one layer is the origin authorization which works by checking the owner of the ASA making the call. This layer checks if the transaction initiator is a blockchain user with the authority to perform the call. The second layer is related to the tree structure itself. The general rule in the authorization of activities is that a parent *may* be able to authorize the reformation of its children (and accordingly, the subtree that the children are a parent to). Therefore, the tree root is the most authorized node meaning that it can change any *bond* in the system. Besides the root of the token tree, any other internal (non-leave) nodes can also manipulate their children given that any of their parents in the tree enables them to do so by setting the *canAuthorizeChildren* (figure 5) to *True*.

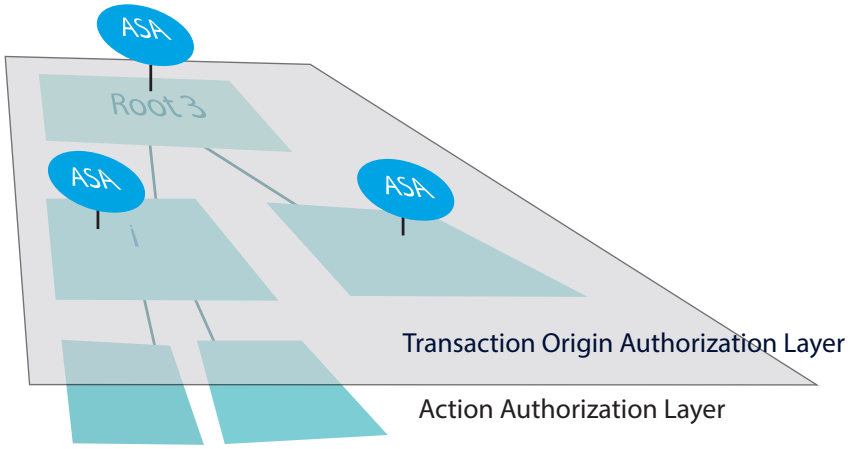

Figure 6: Layered authorization system

3.2 Breaking a bond

Breaking a bond should always be initiated from a parent node ASA owner to the child node. Figure 7 bond-breaking works. Initially, the transaction is issued by the ASA holder of Root 3 (the parent). This call is shown by the dotted red line 1. Then, dotted red line 2 shows the next call in the chain of calls that asks the parent to remove its bond with the child. After the result of that call returns to the child I (red dotted line 3), I will also set its parent field to "NULL". If breaking of a bond is not followed by a make bond, the newly separated child's fields can be later changed to mark it as the root of a new tree. This is not enforced on-chain as UTF focuses on offering a data structure and its primary tree management operations.
The pseudocode of the *breakBond* function logic is demonstrated in listing 1. The operation of the three arrows in figure 7 is performed in a distributed way on both the parent and child side. Line $2-5$ of listing 1 match arrow 3 in figure 7. Root 3 after breaking the bond, will return *True* to the caller (here, token node I) which enables it to break its bond with the parent. This is also demonstrated in lines $6-12$ in listing 1. The *call origin* (in line 2) points to the immediate caller of the function.

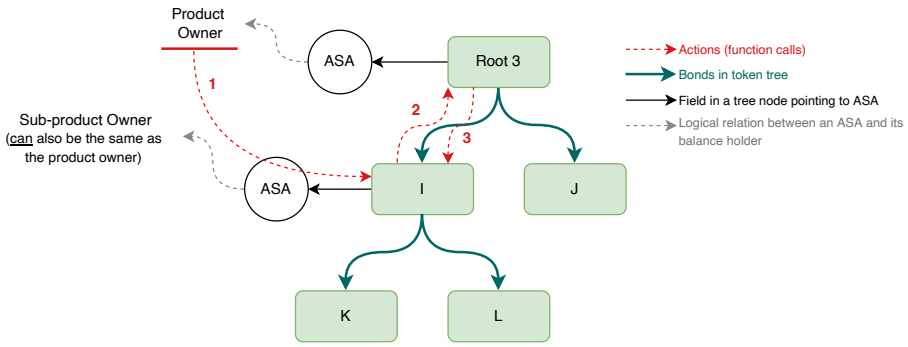

Figure 7: Break bond action (breaking mutual bond: parent to child and vice versa)

Algorithm 1 Break a bond function

```
1:  function BREAKBOND(from, to)
2:      if (call origin == any immediate child) AND (from == call origin AND
    to == this.address) then
3:          this.childrenAddresses.remove(from)
4:          return True
5:      end if
6:      if (call origin == ASA owner of any parent with sub-tree authorization
    ability) OR (call origin == tree root) then
7:          result = call breakBond(this.address, parent) on this.parent
8:          if result == True then
9:              this.parent = NULL
10:             return True
11:         end if
12:     end if
13:     return False
14: end function
```

3.3 Making a bond

Making a bond can be translated into two following operations order:

1. Adding a child to the parent in the child.
2. Changing parent from NULL to a certain address.

Furthermore, if the new child is part of another token tree (the parent is not NULL yet), before making a bond, that already existing bond should be broken. This is ensured in lines $9 - 12$ in listing 2. The first operation of the above list is performed in lines $3 - 8$ where apart from authorizing the call origin user,

cycle creation is also avoided (*if* condition in line 4). The second operation is performed through lines $9-15$ 2. This is done by first calling *makeBond* function in the parent and seeing if the parent accepts this token node as a child. If it does (lines $11-13$), the parent field will also be changed in the child.

Algorithm 2 Make a bond function

```
1: function MAKEBOND(from, to)
2:     isFromAbove:bool = ((call origin == ASA owner of any parent) AND
   (parent.canAuthorizeChildren == True)) OR (call origin == tree root)
3:     if ((call origin == ASA owner of the current node) AND
   (this.canAuthorizeChildren == True)) OR isFromAbove then
4:         if (from == this.address) AND (to != this.parent) AND (to is not
   in this.childrenAddresses) then
5:             this.childrenAddresses.append(to)
6:             return True
7:         end if
8:     end if
9:     if (from == this.address) AND (this.parent == NULL) AND (ASA
   owner of the current node is the same as ASA owner of the 'to') then
10:        result = call makeBond(to, this.address) on 'to' node
11:        if result == True then
12:            this.parent = to
13:        end if
14:        return True
15:    end if
16:    return False
17: end function
```

3.4 Deletation

We define delegation as an organization or user on blockchain being authorized to change the structure of the token tree (or part of it). This ability, however, should be temporary and the actual owner of the product or product part should be able to take control back whenever they wish.

Delegation in UTF is done by transferring the ASA of that product or product part to another user (delegate address), and at the same time setting the *isDelegated* field in the root (or sub-product) to True. The mentioned actions are done in lines $3-5$ in listing 3. One important design choice is that you cannot explicitly delegate the root node. The reason is that taking back the control of a fully delegated tree would not be possible because of the *claimBack* mechanism. The workaround is to have a child below the root and delegate that child instead of the root.

Algorithm 3 Delegation

```
1: function DELEGATE(to)
2:     isFromAboveNotRoot:bool = (call origin == ASA owner of any parent)
   AND (node pertaining to the owner ASA.canAuthorizeChildren == True)
   AND (call origin != tree root)
3:     if isFromAboveNotRoot then
4:         this.isDelegated = True
5:         transfer(this.ASA, to)
6:     end if
7: end function
```

3.5 Claiming a Delegated Token Back

Taking back the control of the delegated sub-tree is easy since the control of any sub-tree is through its parent(s), therefore, the ASA holder of parent(s) can request to take the delegated child node back at any time they want. This is demonstrated in listing 4.

Algorithm 4 Taking control back from the delegate

```
1: function CLAIMBACK
2:     isFromAbove:bool = ((call origin == ASA owner of any parent) AND
   (parent.canAuthorizeChildren == True)) OR (call origin == tree root)
3:     if isFromAbove then
4:         this.isDelegated = False
5:         transfer(this.ASA, this.owner)
6:     end if
7: end function
```

4 Discussion

There might be specific technical challenges in the way of implementing this. An example is being forced to perform consecutive calls to functions in which case we might face limitations regarding maximum call depth.
There might be cases where we only need to read the data from the blockchain. In this case, it is possible to eliminate the problem by merely consecutively reading transactions and blocks to get the tree structure we want. Using this method instead of direct on-chain recursive calls eliminates the limitations related to the maximum depth of calls.

4.1 Off-chain Operations

With the minimalistic design of UTF, it is possible to let more specific requirements of the system be defined and implemented by potential clients, organizations, or even as part of the API described in section 2.1. In the next subsections, we will go through some of these domain-specific requirements.

4.1.1 Structural Violations Monitoring

Operations such as ensuring a particular operation do not violate the token tree structure do not need to be done on-chain. Such operations can be done as part of the API (on-demand), periodically, or before issuing on-chain certain transactions such as makeBond which might add cycles to the tree.

4.1.2 Product Passport

Without having a proper way to retrieve product information and history, UTF would not be very beneficial. There are two ways to integrate product (token) information and history into UTF:

1. Track smart ASAs of organizations, contract accounts, and transactions using an off-chain application. The whole life-long information of the token, hence the product will be available. Information includes origin, current situation, buyers, sellers, recyclers, and refurbishers. These can be added as a field to the smart contract of the root node of the product. Information regarding the product parts can also be added to the tree structure similarly based on the description of tree operations in section 3.

2. Another interesting feature of a blockchain-based system is that merely tracking the transactions that happen on the tokens, or the token trees reveal their buy, sell, and delegation history. This tracking is similar to what blockchain indexers do with more specialized functionality.

5 Conclusion

Universal Token Forest (UTF) is a tree structure that can store hierarchical data on-chain. It uses a two-layer authorization scheme based on Algorand smart contracts logic and Algorand Standard Assets (ASAs). This scheme of authorization helps separate the tree management actions (that are always done in parent-to-child order) from the token (product) owner authorization. We argue that such a separation helps the tree modification to be more granularly controllable. The granular modification capability helps create tokens that match products with sub-products in a hierarchy. These sub-products (sub-trees in the token tree) can then be sold, delegated, or disposed of. This scheme would reflect how the real-world products or product parts would be refurbished, and reused multiple times in the context of a circular economy.

References

[1] M. Eshghie, L. Quan, G. A. Kasche, F. Jacobson, C. Bassi, and C. Artho, "CircleChain: Tokenizing Products with a Role-based Scheme for a Circular Economy," May 2022. arXiv:2205.11212 [cs].

[2] "Tree." https://xlinux.nist.gov/dads/HTML/tree.html.

[3] "Smart contract details - Algorand Developer Portal." https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/.