DARC: Decentralized Autonomous Regulated Corporation

Xinran Wang

December 9, 2023

Abstract

Decentralized autonomous organizations (DAOs) show promise but lack governance mechanisms. This paper proposes DARC, a Decentralized Autonomous Regulated Corporation designed to facilitate oversight of DAOs. DARC operates as a virtual machine on an EVM-compatible blockchain and incorporates configurable "plugins" outlining laws and rules. Users can run programs on the DARC virtual machine to execute company operations such as equity, cash, and voting, with all actions subject to constraints and rules defined by plugins. DARC supports multi-token systems and dividends distribution, resembling corporate features. The By-law Script serves as DARC's programming language, enabling the design of DARC programs and operations, as well as the customization of plugins.

1 Introduction

Decentralized autonomous organizations (DAOs) have emerged as a novel form of internet-native organization mediated by tokenized governance systems. By leveraging cryptographic primitives from public blockchains, participation rights can be embedded directly into cryptoassets/tokens and allocated algorithmically without traditional corporate structures. However, early experiments like The DAO have exposed risks around security, flexibility, and real-world applicability.

This paper introduces DARC - the Decentralized Autonomous Regulated Corporation. DARC incorporates modular "plugins" that encode rules and policies analogous to corporate bylaws. A multi-token system allows flexible rights allocation, mimicking shares. And a virtual machine architecture enables oversight and control of operations.

By synthesizing aspects from corporate structures and DAO architectures, DARC offers a regulated and adaptable foundation for decentralized organizations seeking real-world coordination. It bridges the gap between traditional corporations and autonomous DAOs.

The paper begins by delineating the key principles and design rationale in developing DARC. It presents relevant backgrounds on Ethereum, DAO history and traditional corporate governance innovations that motivate the need for a hybrid paradigm. Next, the overall system architecture is explained followed by specifics on essential sub-components: plugins, multi-token model, sandboxed execution, voting apparatus and others. Salient examples are also provided to illustrate the flexible configurability afforded by DARC across use cases spanning corporate stocks, bonds, boards, upgrades and emergency response among others.

The paper concludes by reflecting on future directions for Decentralized Autonomous Regulated Corporations in catalyzing a new paradigm of organizational structures and economic coordination on the blockchain.

[0]In any scholarly discourse involving the Decentralized Autonomous Regulated Corporation (DARC), encompassing academic papers, journals, books, documents, conference papers, manuals, reports, analyses, slides, and any literature or outputs of equivalent nature, as well as any literature or outputs originating from research and projects of academic, commercial, educational, governmental, or other affiliations either directly or indirectly sponsored or invested in the form of DARC, and any literature or outputs of similar nature arising directly or indirectly from the utilization of any products associated with Project DARC, it is imperative for the authors or organizational entities involved in the projects, including institutions, groups, organizations, governments, and educational institutions, to meticulously include this paper in the list of cited references in the scholarly literature.

2 Principles of DARC Protocol

In designing DARC to be a regulated, programmable, customizable, profitable, and sustainable business entity, we adhere to the following principles:

2.1 Plugin as a Law

For real-world companies, compliance with a multitude of laws and regulations is essential. This compliance encompasses not only a company's By-laws but also its internal policies, agreements between the company and its employees, contracts with customers and suppliers, and agreements among shareholders. These agreements and rules define the operational procedures and boundaries of the company, forming the foundation for its healthy operation and growth. They not only determine the company's structure but also provide detailed descriptions of how the company operates and how profits are distributed.

As a purely virtual corporate entity existing solely on an EVM-compatible blockchain, within the DARC protocol, plugins serve as a core and foundational mechanism, representing a variety of by-laws, contracts, legal agreements, documents, and more. In each DARC protocol, there exists a set of plugins that form the fundamental laws governing the DARC. All operations and activities conducted within the DARC must rigorously adhere to and comply with all restrictions and conditions outlined by these plugins.

For the DARC protocol, plugins need to adhere to the following principles:

2.1.1 Design of Plugins

Each plugin consists of two crucial components: "condition" and "decision". The "condition" represents the triggering criteria that activate the plugin, while the "decision" specifies the actions to be taken when the condition is met.

The "condition" of each plugin is programmable and configurable, allowing for flexibility. When an operation is executed or when the DARC's state aligns with the specified condition, the plugin responds by implementing the corresponding "decision".

2.1.2 Levels of Plugins

For each operation, there is an associated "level". Within each DARC protocol, there are multiple plugins, each of which may have the same or different levels. When multiple plugins are triggered simultaneously for a single operation, the DARC protocol selects the plugin with the highest level and uses that plugin's decision as the final decision.

Additionally, all plugins at the same level must have the same decision type. This requirement ensures that when multiple plugins at the same level are triggered simultaneously, the final decision is consistent. This design simplifies the decision-making process and ensures that there is no ambiguity when multiple plugins of the same level are involved.

2.1.3 Immutability of Plugins

Once a plugin is added to the DARC protocol, it cannot be changed or modified. Users have the capability to enable or disable one or multiple plugins through the use of "enable operation" or "disable operation". When users wish to modify the rules represented by a plugin, they must disable the existing plugin that represents those rules and then add and enable a new plugin to replace it. This approach ensures that the integrity of the protocol is maintained while allowing for changes in the rules as needed by the users.

2.1.4 Authority of Plugins

In the DARC protocol, the plugin system holds authoritative control. For each operation, if it is rejected by the plugin system, it cannot proceed further. If the plugin system requires a vote, the operation can only be executed after undergoing the voting process. Only when the plugin system grants full approval can an operation be directly executed. This design ensures that the plugin system has the final say in the execution and validation of operations within the DARC protocol.

In the DARC Protocol, operations related to plugins include enabling plugins, disabling plugins, adding plugins, and adding and enabling a plugin. When users execute these actions associated with plugins, they are required to adhere to the regulations and constraints set by the existing plugins, just like any other operations. These operations targeted at plugins can only be executed after receiving approval from the current set of plugins.

In summary, any operations involving modifications to plugins must also receive approval from the current set of plugins. This ensures fairness and compliance with the rules and regulations established by the existing plugins within the DARC Protocol.

2.2 Program and Operations

For each DARC program, it consists of a series of operations, where each operation includes an opcode, a set of parameters, and an operator address. When an operator requests the DARC to execute a program, several scenarios may arise:

1. If one or more operations within this program receive a rejection decision from the plugin system, the entire program will be denied execution. In such a scenario, even if some of the operations meet the requirements, the program as a whole will be unable to proceed.

2. If all operations within this program do not receive rejected decisions from the plugin system, but one or more operations in this program receive a "voting needed" decision, a voting process is initiated. In this scenario, all voting items will be consolidated into a single voting process, and the DARC enters a voting state. This allows all token holders to participate in the voting. If the program is approved through the voting process, it can proceed with execution. However, if it is rejected through the voting, the project will be rejected.

3. If each operation within this program receives an "approved" decision from the plugin system, in this scenario, all operations within the program can be executed sequentially and directly.

2.3 Multi-level Token System

The DARC protocol features a multi-level token system, with each level of tokens having independent voting weights and dividend weights, where the minimum values for these weights can be set to 0. It's important to note that the voting weight and dividend weights for each level of tokens are immutable and cannot be altered. Users have the capability to initialize a new level and perform a range of operations, including minting, burning, transferring, and more for all tokens.

By assigning voting weights and dividend weights to tokens at each level, imposing limitations on token quantities, and incorporating additional plugins to restrict and design various token-related operations, multi-level tokens can serve a multitude of purposes, including common stock, bonds, board votes, A/B shares, preferred stock, common commodities, Non-Fungible Tokens (NFTs), and more.

2.4 Dividends

A company has two methods of spending money: one is through direct cash payments, typically used for purchasing, salary disbursement, bill payments, and bond redemption, among other purposes. These payments generally involve one-time or multiple transactions of fixed amounts. The other method is through dividend payments, where the company allocates a specific amount or a certain percentage of funds and distributes them to all shareholders based on dividend weights.

In the DARC protocol, the dividend mechanism distributes a dividend, which consists of X permyriad of the accumulated income from every N transactions. This dividend is distributed to all token holders based on their dividend weight.

In the DARC protocol, there is a dividend cycle counter. Every time the DARC receives a dividendable payment, this counter automatically increments by 1, and the amount of this payment is added to the dividendable fund pool. When the dividend cycle counter reaches the predefined dividend cycle N within the DARC protocol, users can, subject to approval by the plugin system, execute the "`OFFER_DIVIDENDS`" operation. This operation disburses X permyriad of the funds from the dividendable fund pool, calculates the dividends, and distributes them to the accounts of each token holder. Subsequently, both the dividendable fund pool and the dividend cycle counter are reset to zero.

2.5 Voting

In the DARC protocol, for operations that cannot be directly approved or denied by the plugin system after triggering specific conditions, a voting mechanism can be employed to make the final decision. The voting mechanism can serve various purposes, including:

1. Democratic decision-making where all token holders have one vote each, suitable for major decisions involving a large number of token holders.
2. Quick and simple decision-making for smaller groups such as boards or committees.
3. Approval processes for routine tasks involving multiple managers taking turns to approve daily affairs.
4. Weighted voting processes for different groups, including A/B classes or multiple levels of voting rights.

For each plugin, if the decision is "voting needed", the plugin must be associated with a voting item. When the condition of this plugin is triggered, and the plugin has the highest level among all the plugins with triggered conditions, DARC will select the voting item specified by that plugin as the voting rule. Once all the voting items are collected by DARC, a voting process will be initiated based on these items. All token holders who meet the criteria specified by this series of voting items are eligible to participate in the voting.

After a program undergoes evaluation by the plugin system, each operation within the program may be subject to voting as requested by one or more plugins. Each plugin will point to a specific voting item. When DARC initiates a voting process, it collects all the voting items required for the operations and proceeds to the voting phase. Assuming that the total number of collected voting items is N, each voter must submit a program containing only one voting operation. This operation must include a boolean value array with a length of N, corresponding to the voting results for the N voting items. Each time an operator submits their vote, the voting process calculates the voting weight associated with that operator for each of the N voting items. It tallies the votes separately for each of the N voting items, adding the voting weight to the respective voting result. If a user's token balance is zero for one or more voting items, their voting weight for those specific items is also considered as zero.

2.6 Emergency

In a rule-based corporate virtual machine, DARC operators may encounter various types of errors, including operational errors, incorrect parameters in operations, and various potential non-technical conflicts and disputes. DARC maintains a backdoor, known as "emergency agents", to serve as firefighters in emergency situations.

In a DARC protocol, operators have the ability to designate multiple addresses as emergency agents. In the event of an emergency, and with the permission of the plugin system, operators can call upon one or more of these emergency agents for assistance. Once an emergency agent is summoned, they gain super-administrator privileges within the DARC, granting them the authority to perform any action. This includes adding, enabling, disabling plugins, conducting token operations, and managing cash assets.

Since emergency agents possess the highest level of comprehensive administrative authority, their role can be likened to that of both a court and a firefighter. Therefore, it is essential for all members of the DARC to have complete trust in these emergency agents. Additionally, specific conditions should be established for calling upon emergency agents to prevent them from taking untimely or inappropriate actions, which could lead to damage or losses within the DARC.

3 DARC Architecture

The DARC is a virtual machine built with Solidity programming language, and can be compiled and deployed to any EVM-compatible on local devnet, testnet or mainnet without contract code size limit(EIP-170). After the DARC protocol is successfully compiled and deployed to your blockchain, user can compose a program and send it to your DARC virtual machine using darc.js(the official Node.js SDK for deploying and interacting with your DARC) or ethers.js with corresponding DARC Application Binary Interface(ABI). User can also read data and information from the deployed DARC in this way.

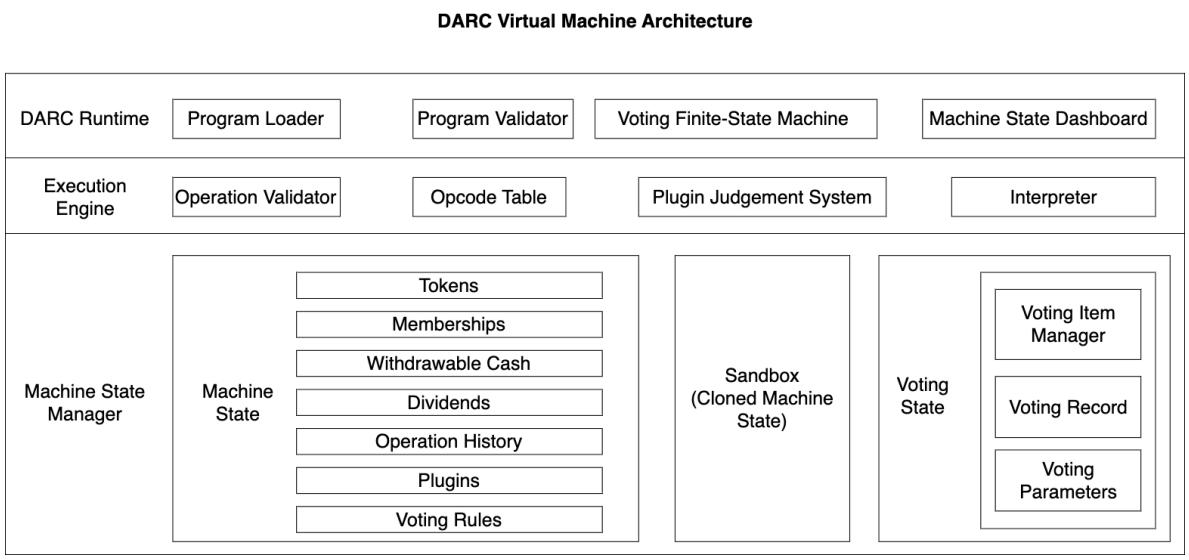


Figure 1: The architecture of DARC virtual machine.

Figure 1 shows the overview of the DARC protocol.

3.1 DARC Runtime

The first level of DARC virtual machine is the Application Binary Interface (ABI) of the smart contract that allows user to compose and pass the programs into the DARC for executing. The program loader is the the entrace for all programs that are sent from user by using darc.js or ethers.js.

3.2 Program Loader

Program loader is the module in the DARC runtime that receive the program from users. As the entrance of the DARC virtual machine, Program loader will manage the whole process of handling the program, including validating the program, executing the program, determine the program compliance with the restriction plugin system, sending the program to the sandbox, starting a voting, ending a voting, pause a voting, refund unnecessary cash (the native tokens) back to the account balance. Program loader is the basic and fundamental module that handle the whole lifecycle of executing a program, as well as the state of the DARC for voting period.

3.2.1 Program Validator

Program validator is the module that checking if the program is valid, including calculate the total consumption of native tokens for each program, checking the basic syntax and parameters of each operation. When receiving the calculated consmuption of native tokens from program validator, the program loader will compare with the actual amount of native tokens user send with the program. If the value of native tokens sent is greater than the total comsumption, the program will be executed and the remaining amount of native tokens will be sent back to the user's balance; if the value of native tokens is less than the total comsumption, the program will be rejected directly, and all the native tokens will be refund into user's balance.

3.2.2 Voting Finite-State Machine

Voting Finite-State Machine is a Finite-State Macine(FSM) that handle the lifecycle of a voting procedure, including an idle state(all program can be submitted and executed), voting state(the process that only the program with operation "vote" can be executed), and a execution pending state(a period of time that allows user to execute a pending program that is approved by previous voting process).

3.2.3 Machine State Dashboard

Machine State Dashboard is a set of interface that allows user to read all the necessary data and information from the DARC entity, including the basic information and parameters of DARC setup, the balances of multi-level token system for all token holders, the balance of withdrawable cash and dividends, the restriction plugins, the voting rules, and operation histories. All the functions in Machine State Dashboard are read-only and declared with "view" keyword, in which user can read the information without extra gas fee.

3.3 Execution Engine

The Execution Engine is the core module that check the validation via plugin system and execute each operation in both machine state and sandbox in the DARC virtual machine. It is similar like other intepreter for other Interpreted programming language, such like Java, Python, JavaScript, Perl, etc.

After a program is validated and sent from runtime program loader, it will be directly passed into the execution engine. The Operation Validator is a module that checks if each operation is valid with right parameter syntax, such as the length of parameters and the types of parameters. After each opcode and corresponding parameters from the program is checked and validated by the operation validator, it will be ready to be checked by plugin judgement system and executed inside the opcode table and interpreter.

The opcode table is the module that analyze each opeartion with opcode and parameters and execute in the intepreter. When an operation is sent into the opcode table, it will be compared and sent to the interpreter with parameters and execute in the machine state or in the sandbox.

The plugin judgement system is the core module that check each operation and make the final judgement for the program, which decide the program should be accepted and executed in the machine state, rejected, executed in the sandbox and make the decision, pending and start voting for the final decision, or rejected after executed in the sandbox. The plugin judgement system reads two arrays of restriction plugins from the machine state: before-operation plugins array and after-operation plugins array.

When the program passes the operation validator, it will be first checked by all the before-operation plugins: if all of the operations are approved by all before-operation plugins and allowed to be executed in the machine state, it will be passed to the opcode table and executed in the interpreter; if at least one of the operation is rejected by any before-operation plugin, the whole program will be rejected and the transaction will be reverted; otherwise, if none of the operations are rejected, but at least one of the operation is marked by any before-operation plugin as "sandbox needed", the legality of the program is not determined yet, and the whole program needs to be executed in the sandbox.

After executed in the sandbox, the operation and the sandbox state will be checked by the after-operation plugins: if all of the operations and sandbox state are approved by all after-operation plugins, the program will be passed back to the opcode table and executed in the interpreter; if any of the operations or sandbox state is rejected by any after-operation plugin, the program will be rejected and the transaction will be reverted; otherwise, if none of the operations are rejected but at least one of the operation is marked by any after-operation plugin as "vote needed", then the program will be suspended and waiting for the final vote results, and will be allowed to executed after approved by the voting process and executed in the "execution pending time".

Since the sandbox also contains a copy of all before-operation plugins and after operation plugins, the after-operation sandbox check only uses the after-operation plugins in the current state. This is because the legality of a program should be only determined by the current machine state, including current plugin judgement system, and the machine state is allowed to be updated only by the program that is legal or approved by current machine state, current plugin judgement system and the voting result if necessary.

The workflow of the execution engine is shown in the figure 2.

3.4 Machine State Manager

Machine state manager is the module that store and manage all the state of the DARC. It contains three part: machine state, sandbox(the cloned machine state) and voting state. The machine state

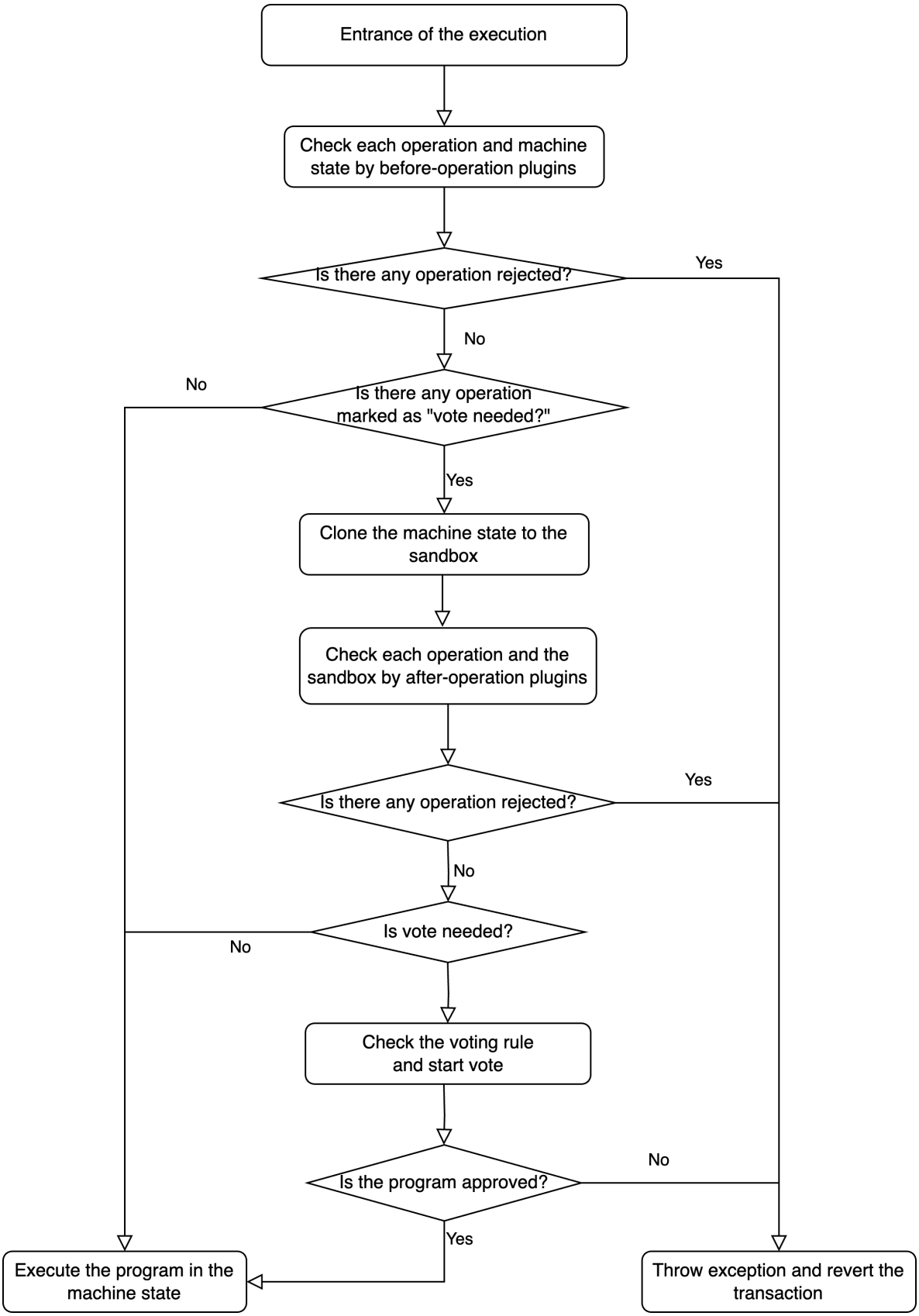


Figure 2: The workflow of execution engine.

contains all the necessary state and assets of the DARC virtual machine, including token system, memberships, withdrawable cash and dividends balance, operation history, plugins and voting rules.

3.4.1 Tokens

The token system is the most fundamental and core design in the DARC protocol. The machine state contains an array of tokens, each token contains different voting weight, dividend weight, token balance, token symbol(token information) and total supply.

The structure of each level token's information is stored in the machine state manager in following Solidity struct.

```
struct Token {
  uint256 tokenClassIndex;
  uint256 votingWeight;
  uint256 dividendWeight;
  mapping (address => uint256) tokenBalance;
  address[] ownerList;
  string tokenInfo;
  uint256 totalSupply;
  bool bIsInitialized;
}
```

The token balance is the the core mapping from address to uint256. The key is the owner address of current level token, and the value is the number of tokens owned by this owner address. Also DARC maintains an array of address, ownerList, as the full list of all addresses with at least one token in current level of token.

By maintaining the array of token struct, the machine state manager can initialize a new token level, mint tokens, burn tokens, transfer tokens and other functionalities.

3.4.2 Memberships

Membership is a a mapping from addresses to levels that enable the DARC to manage different address hierarchicially. The governance of DARC can make sure that different level of people can be assigned with different levels.

The reason that we introduce membership in DARC protocol is that users can develop different plugins to define different rules related to roles and addresses.

In the implementation of DARC Protocol, membership is defined as the following data structure:

```
/**
* The member list and internal role index number of each token owner,
* which can be used to represent the shareholders, co-founders, employees,
* board members, special agents, etc.
*/
mapping (address => MemberInfo) memberInfoMap;

/**
 * @notice The stock token owner information of the DARC protocol
 * This struct is used to store the role index number of each token owner
 */
struct MemberInfo {
  bool bIsInitialized;
  bool bIsSuspended;
  string name;
  uint256 role;
}
```

3.4.3 Withdrawable cash and dividends

Withdrawable cash and dividends are two separate balance hashmaps. Operator can withdraw native tokens(like Ethereum) from both of the balance if the balance is not zero. For withdrawable cash, the only way to add withdrawable tokens to the address balance is to execute `BATCH_ADD_WITHDRAWABLE_BALANCE` operation with target addresses and amounts of tokens. If the operation is approved, the corresponding amounts will be added to the balances. For dividends, the only way to add withdrawable tokens to the balance is to execute `OFFER_DIVIDENDS` operation without any parameters. If the operation is approved, all token holders with dividendable tokens(tokens with dividend weight larger than 0) will receive the corresponding amount of dividends.

To withdraw the native tokens from the DARC to the address, the operator can execute `WITHDRAW_CASH_TO` to transfer withdrawable cash from balance to target addresses, or execute `WITHDRAW_DIVIDENDS_TO` to transfer dividends from balance to target addresses if the operations are approved.

3.4.4 Operation History

Operation history is a log system that contains each address with its all the corresponding operations, each operation with its latest timestamp. When a program is executed and terminated successfully, the timestamps of the operations contained in the program will be updated in the operation history. With the operation history, developers can set minimum time interval between each operations for a single address or for all members in the DARC.

Rule 1 is an example about using operation history to limit the operator to execute the same operation in a period of time:

Rule 1: We provide 10 ETH per month as the salary for both employee-level and manager-level address (level 4 & 5).

For the **Rule 1**, we need to make sure that if the operator is assigned with a role level equals 4 or 5, the operation is adding withdrawable balance, the total amount of operation is less than or equal to 10 ETH(or 10000000000000000000 wei), and the operator did the same operation in more than 30 days(or 2592000 seconds), this operation will be approved and sandbox check can be skipped for this operation.

Here is an example plugin for **Rule 1** designed in By-law Script:

```
const plugin_Rule_2 =
{
    condition: (operation_name === BATCH_ADD_WITHDRAWABLE_BALANCE)
               & ( total_add_withdrawable_balance <= 10000000000000000000)
               & ( last_operation_by_operation_period >= 2592000)
               & ( (operator_membership_level === 4) | (operator_membership_level === 5) ) ,
    return_type: YES_AND_SKIP_SANDBOX,
    is_before_operation: true,
    return_level: 100,
    voting_rule_index: 0
}
```

3.4.5 Plugins

Plugins are the fundamental mechanism in DARC protocol, because all the rules and regulations need to be defined, transpiled and saved in DARC plugin system. There are two plugin arrays in the DARC protocol: before-operation plugins and after-operation plugins.

In the figure 2, each program submitted to a DARC needs to be checked by all before-operation plugins, and if any operation of the program violates any before-operation plugin, the program will be rejected.

If all operations are approved and no sandbox check needed, the DARC will execute the program directly; if one or some operations needs to be checked in the sandbox, the DARC will first copy all states from current DARC into the sandbox, then execute the whole program inside the sandbox, and double check the state of sandbox with after-operation plugins: if all the states and operations are approved by all after-operation plugins, the program will be approved and finally be executed in the real machine state; if one or some of the operations cannot be approved and need voting to make the

further decision, the program will be suspended and the plugin system will start a voting process in the DARC, and the program will be approved and executed if and only if every pending operation is approved by the voting process, otherwise the program will be aborted and back to idle state.

Each plugin is composed with following items:

- Return type: the decision of the plugin when the condition is triggered, including `YES`, `VOTE_NEEDED`, `NO`, `YES_AND_SKIP_SANDBOX` and `SANDBOX_NEEDED`.
- Level: the priority of current plugin when the condition is triggered. When an operation triggers multiple plugins, the plugin system will make the final decision with the return type from the plugin with the highest level. Since each level only allows plugins with the same return type, this will make sure that each operation will get a certain judgement result from the plugin system.
- Condition nodes: the `conditionNodes` is an array of condition nodes that represents the expression binary tree of this plugin. Each node can be a condition expression with parameters that validate a certain criteria, or a logical operators that combines two or multiple child nodes (condition expressions or logical operations) into a single condition expression.
- Voting rule index: the index number that pointing to a certain voting rule if the condition expression criteria is triggered and the return type of the plugin is `VOTE_NEEDED`. If a voting process is needed, the judgement system will traverse all `VOTE_NEEDED` plugins and create a voting process with multiple items with selected voting rules.
- Note: the notes by plugin designers for future purpose.
- Boolean flag `bIsEnabled`: the boolean flag that indicates if the plugin is enabled or not. Since the plugin is immutable in the whole life cycle, the operator cannot remove or modify the plugin after deployed successfully. The only way to disable a certain plugin is to set the boolean flag `bIsEnabled` from `True` to `False`. Also the plugin can be enabled by set it back to `True` if the operation is allowed.
- Boolean flag `bIsInitialized`: the boolean flag that indicates if the plugin is successfully initialized. If `False`, the judgement system will skip this plugin.
- Boolean flag `bIsBeforeOperation`: the boolean flat that indicates if the plugin is before-operation or after-operation plugin array. Although the before-operation plugins and after-operation plugins are stored and managed in two seperate arraies, this boolean will be double-checked when judgement system goes through each array. This field is required to make sure that each plugin object struct is aligned when constructing the plugin from By-law script and sending through the DARC program entrance.

The structure of a plugin is designed in Solidity with below struct:

```
struct Plugin {
  /**
   * the return type of the current condition node
   */
  EnumReturnType returnType;

  /**
   * the level of restriction, from 0 to the maximum value of uint256
   */
  uint256 level;

  /**
   * condition binary expression tree vector
   */
  ConditionNode[] conditionNodes;

  /**
```

```
   * the voting rule id of the current plugin if the return type is VOTE_NEEDED
   */
  uint256 votingRuleIndex;

  /**
   * the plugin note
   */
  string note;

  /**
   * the boolean that indicates whether the plugin is enabled or not
   */
  bool bIsEnabled;

  /**
   * the boolean that indicates whether the plugin is deleted or not
   */
  bool bIsInitialized;

  /**
   * the boolean that indicates whether the plugin is a before operation
   * plugin or after operation plugin
   */
  bool bIsBeforeOperation;

}
```

3.4.6 Voting Rules Array

Voting rules array is an array that stores all the voting rules for the DARC protocol. When a program is executed in the sandbox and being checked by the plugin judgement system, one or multiple operations may be pending and waiting for the voting results. During the voting process, all members are allowed to vote during the valid voting period. Each voting rule contains all the necessary requirements that starts the voting item.

3.5 Sandbox

The sandbox of DARC has exactly the same storage design as the machine state manager, including the token system, memberships, withdrawable cash, dividends, operation history, plugins and voting rules. For each program that cannot be approved by the before-operation plugins, the DARC machine state manager will first clone the machine state to the sandbox, execute the program in the sandbox, and make the decision based on the execution result and the sandbox state.

3.6 Voting State

In the machine state manager, voting state is a standalone module that managing all voting-related properties. Voting item manager contains all the voting items in history, and each voting item contains the program waiting for the final voting result, the accumulated voting power and voting records for each address.

4 By-law Script

By-law Script is a programming language with a JavaScript-like syntax specifically designed for use with the DARC protocol. Users can use By-law Script to execute a variety of tasks, including:

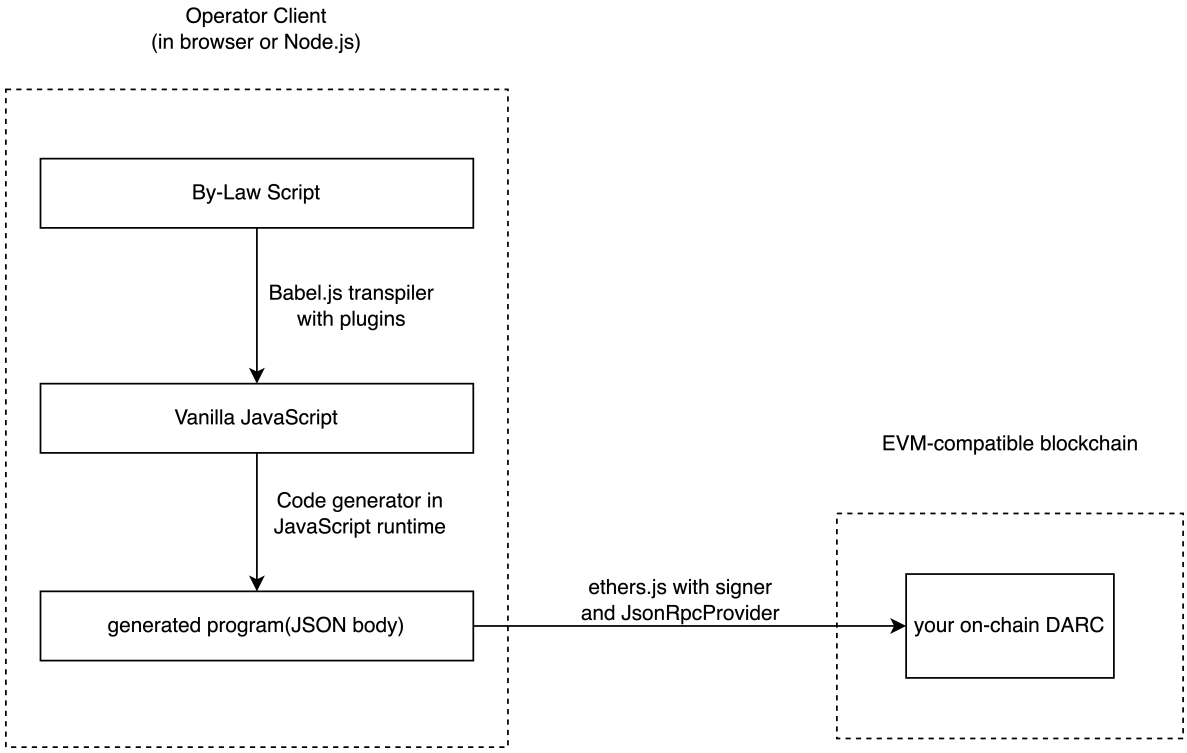


Figure 3: The process of compiling and executing By-law Script

- Operations: Users have the capability to initiate a series of operations using By-law Script. These operations encompass actions like minting and burning new tokens, token transfers, token purchases, enabling or disabling plugins, and fund withdrawals, among others.
- Plugin Design: By-law Script allows users to design plugins with binary expression trees, incorporating logical operators and DARC judgment expressions with parameters. These plugins can define a return type and, when necessary, specify a voting rule.
- Voting: By-law Script facilitates the process of voting for the current operation.

The syntax of the By-law Script closely resembles that of vanilla JavaScript, encompassing variables, constants, assignments, basic data types, functions, classes, and various other features. In addition to the fundamental JavaScript syntax, the By-law Script also supports operator overloading, which is particularly useful for describing condition nodes for plugins.

Figure 1 illustrates how the By-law Script is executed within the DARC protocol. On the operator client side, the program of the By-law Script needs to be transpiled into vanilla JavaScript. Subsequently, it is executed in the local JavaScript runtime using a code generator. After the entire program is generated within the JSON body, it takes the form of a list of operations with opcodes and parameter lists. At this point, the program is ready to be submitted from the client side as a function call to the DARC Application Binary Interface (ABI) on the EVM-compatible blockchain.

To execute the program within the DARC protocol, the DARC Software Development Kit (SDK) gains access to the address of the DARC smart contract and initiates the execution process by calling the entrance function with a wallet signature. This communication is facilitated through the JSON RPC server of the EVM-compatible blockchain.

4.1 Transpiler

After a user runs the By-law Script, the first step in the transpilation process involves converting the By-law Script into vanilla JavaScript. This initial transformation employs Babel.js for front-end syntax analysis and utilizes an operator overloading plugin to generate the resulting vanilla JavaScript code on the back end.

The primary significance of using the operator overloading plugin lies in its ability to transpile the syntax of logical operators within condition nodes designed by the user. Users create plugins to logically connect and describe various condition expressions using logical operators. When these plugins are transpiled with the operator overloading feature, each condition within a plugin is transformed into a tree-like structure expressed as an object constructed using pure functions and parameters.

Furthermore, the transpiler also supports other advanced ECMAScript syntax and syntactic sugar, making it convenient for users to work with.

4.2 Code Generator

Once a user has successfully generated vanilla JavaScript from the transpilation of By-law Script, it can be executed within a JavaScript runtime environment, whether it's in Node.js or a web browser.

The user-generated transpiled vanilla JavaScript code includes one or more operation command functions, each comprising opcodes and their corresponding parameters. Upon execution, each of these operation segments is stored in an array, ultimately forming a program object. Subsequently, the code generator leverages this program object to create a complete program JSON body, adhering to the DARC ABI entrance specifications.

Once the program JSON body has been successfully generated, users can utilize ethers.js to send this program body, following the ABI, to the DARC protocol on an EVM-compatible blockchain. This process ensures that the program is executed in its entirety within the DARC.

Figure 4 illustrates a complete compilation process. The entire By-law Script toolchain first parses the By-law Script into an expression tree, transpiles it into vanilla JavaScript, and then, within the JavaScript runtime, the code generator of vanilla JavaScript produces a condition node array for plugins in a DARC Program.

`transfer`

5 Multi-Class Token System

Multi-class token system is the core mechanism in the DARC protocol, users can design different levels of tokens with different voting weights and dividend weights. With plugins as restrictions, the multi-class token system can be used to represent different components or assets in the organization.

5.1 Common Stocks

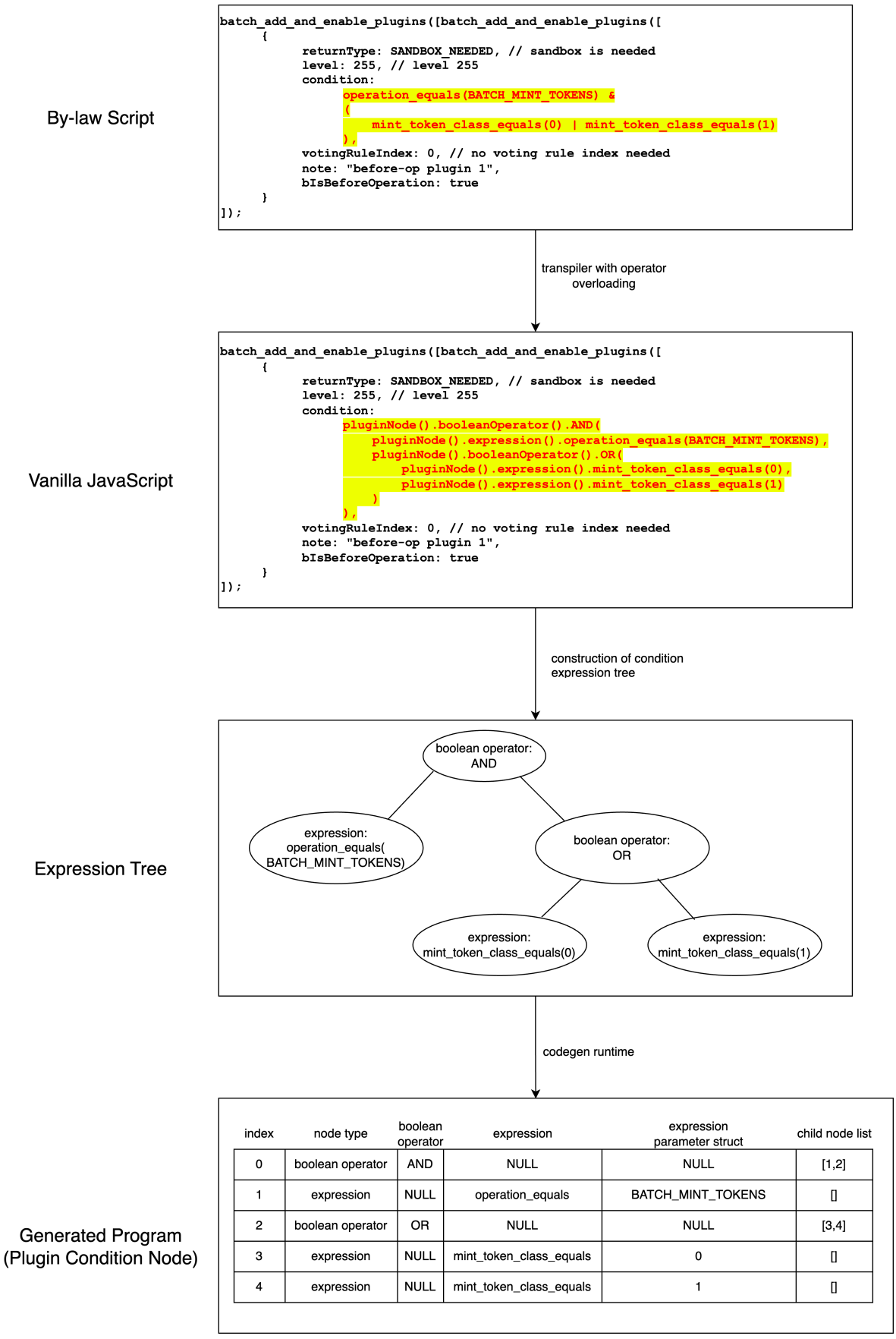
Common stocks are a central element in the mechanism of a joint stock company, and they serve as a means to raise capital and distribute ownership. Shareholders of common stock typically enjoy voting rights, allowing them to have a say in important company decisions during shareholder meetings. Additionally, common stockholders may receive dividends, which are a portion of the company's profits distributed to shareholders. This dual aspect of voting and dividends makes common stocks a key instrument for shareholders to participate in the governance and financial returns of the company.

If a token level is assigned a voting weight of 1 and a dividend weight of 1, and all essential matters in the DARC must be approved by all token holders at this level, then this token level can be considered equivalent to common stock in the DARC.

To instantiate such a token level, one must first establish it in the By-law Script, assigning a voting weight of 1 and a dividend weight of 1. Upon the successful execution of this script, the DARC protocol will initialize a level-0 token endowed with both a voting weight and dividend weight of 1.

```
batch_create_token_classes(
    ["TOKEN_0"],   // the token symbol in string
    [0],           // the level of token
    [1],           // the voting weight
    [1]            // the dividend weight
);
```

Consider about the complexity of corporation affairs, two voting rules are created for different purposes:



index	node type	boolean operator	expression	expression parameter struct	child node list
0	boolean operator	AND	NULL	NULL	[1,2]
1	expression	NULL	operation_equals	BATCH_MINT_TOKENS	[]
2	boolean operator	OR	NULL	NULL	[3,4]
3	expression	NULL	mint_token_class_equals	0	[]
4	expression	NULL	mint_token_class_equals	1	[]

Figure 4: Transpiling and Code Generation for Plugins

Voting rule 1: To approve an operation, the following conditions must be met: there must be more than 50% valid voting power in favor of the operation, and this voting needs to be in absolute majority mode, which means it must comprise at least 50% of the total token voting power. The time duration for voting rule 1 is 7 days(604800 seconds). After the program is approved, the operator needs to execute the program in 1 hour(3600 seconds).

Voting rule 2: To approve an operation, the following conditions must be met: there must be more than 75% valid voting power in favor of the operation, and this voting needs to be in relative majority mode, which means it must comprise at least 75% of the valid voting power. The time duration for voting rule 2 is 1 day(86400 seconds). After the program is approved, the operator needs to execute the program in 1 hour(3600 seconds).

The By-law Script below shows the initialization of voting rule 1 and 2. When adding a new plugin with voting rule index 1 or 2, the plugin will make sure that when the condition is triggered and the plugin is with the highest priority, a voting process will start with the selected voting rule.

```
batch_add_voting_rule([
    // add rule 1
    {
        // token level-0 is only allowed for the voting
        votingTokenClassList: [0],

        // approval threshold: 50%
        approvalThresholdPercentage: 50,

        // voting duration: 604800 seconds
        votingDurationInSeconds: 604800,

        // execution pending duration: 3600 seconds
        executionPendingDurationInSeconds: 3600,

        // is voting rule enabled or not: true
        isEnabled: true,

        // the note about the the voting rule
        note: "This is the voting rule index 1",

        // is absolute majority
        bIsAbsoluteMajority: true
    },

    // add rule 2
    {
        // token level-0 is only allowed for the voting
        votingTokenClassList: [0],

        // approval threshold: 75%
        approvalThresholdPercentage: 75,

        // voting duration: 604800 seconds
        votingDurationInSeconds: 86400,

        // execution pending duration: 3600 seconds
        executionPendingDurationInSeconds: 3600,

        // is voting rule enabled or not: true
        isEnabled: true,

        // the note about the the voting rule
```

```
        note: "This is the voting rule index 2",

        // is absolute majority
        bIsAbsoluteMajority: false
    }
]);
```

5.2 Class A/B Stocks

Class A and Class B stocks are fundamental tools in corporate finance. Class A provides voting rights and dividend advantages, typically held by founders. Class B, with limited voting rights, raises external capital while retaining control. This dual-class structure balances governance and growth needs, with implications for shareholders' decision-making and financial interests.

First initialize two token level: level-0 with voting weight 1 and dividend weight 1, and level-1 with voting weight 100 and dividend weight 1.

```
batch_create_token_classes(
    ["TOKEN_0", "TOKEN_1"],   // the token symbol in string
    [0, 1],           // the level of token
    [1, 100],           // the voting weight
    [1, 1]            // the dividend weight
);
```

Next we will mint 10000 level-0 tokens and 200 level-1 tokens. This will make the total voting power of level-0 tokens as 10000, and 20000 for level-1 tokens. In this way, the total voting power of the DARC is 30000, in which 66.7% in level-1 tokens. In this way, the co-founders and early stage investors will control the majority of the DARC for voting process which allows all level-0 and level-1 token holders.

We assume that `addr1` and `addr2` hold 10000 level-0 tokens in total, 5000 for each, and each of `addr3`, `addr4`, `addr5` and `addr5` holds 50 tokens. First mint tokens to these addresses in By-law Script.

```
batch_mint_tokens(
  [0, 0, 1, 1, 1, 1 ],
  [5000, 5000, 50, 50, 50, 50],
  [addr1, addr2, addr3, addr4, addr5, addr6]
  );
```

Next add a plugin to limit the total supply of level-0 and level-1 token as 10000 and 20000.

Voting Rule 1: To approve the operation, 90% of total voting power of both level-0 and level-1 token must vote `YES`. The voting process needs to be in the absolute majority mode. The time duration for voting rule 1 is 7 days(604800 seconds). After the program is approved, the operator needs to execute the program in 1 hour(3600 seconds).

Before-Operation Plugin Rule 1: If the operation is `BATCH_MINT_TOKENS` and the level of token mint is 0 or 1, the operation needs to be executed in the sandbox before making the decision.

After-Operation Plugin Rule 1: If the operation is `BATCH_MINT_TOKENS` and the level of token mint is 0 or 1, the operation needs to be approved by voting rule 3.

Before-Operation Plugin Rule 2: If the operation is `BATCH_DISABLE_PLUGIN` and the index is before-operation plugin 1, before-operation plugin 2 or after-operation plugin 1, reject the operation. This plugin is with the highest priority.

In before-operation plugin 1 and after-operation plugin 1, when changing the shareholder structure by minting new level-0 or level-1 tokens, the operation needs to be approved by 90% of the total voting power. This will prevent the level-1 shareholders from diluting the value of level-0 tokens and protect those level-0 token holders, mostly retail investors.

The before-operation plugin 2 is a protection for the other plugins above. When any operator is trying to disable any of the three plugins, the operation will be rejected. This will permanently locked the mechanism and guarantee that the plugins cannot be disable.

```
batch_add_voting_rules([
    // add before-operation plugin 1
    {
        // token level-0 and level-1 are allowed for the voting
        votingTokenClassList: [0, 1],

        // approval threshold: 90%
        approvalThresholdPercentage: 90,

        // voting duration: 604800 seconds
        votingDurationInSeconds: 604800,

        // execution pending duration: 3600 seconds
        executionPendingDurationInSeconds: 3600,

        // is voting rule enabled or not: true
        isEnabled: true,

        // the note about the the voting rule
        note: "This is the voting rule 1(index 0)",

        // is absolute majority
        bIsAbsoluteMajority: true
    },
]);
```

Then add three plugins to the DARC.

```
batch_add_and_enable_plugins([
    // adding before-operation plugin with index 0
    {
        returnType: SANDBOX_NEEDED, // sandbox is needed
        level: 255, // level 255
        condition:
            operation === BATCH_MINT_TOKENS &
            (
                mint_token_class === 0 | mint_token_class === 1
            ),
        votingRuleIndex: 0,  // no voting rule index needed
        note: "before-op plugin 1",
        bIsBeforeOperation: true  // the plugin will be added as before-op
    },

    // adding after-operation plugin with index 0
    {
        returnType: VOTE_NEEDED, // vote is needed
        level: 258, // level 258
        condition:
            operation === BATCH_MINT_TOKENS &
            (
                mint_token_class === 0 | mint_token_class === 1
            )
        votingRuleIndex: 0,  // voting rule 1, index = 0
        note: "after-op plugin 1",
        bIsBeforeOperation: false  // the plugin will be added as before-op
    },
```

```
    // adding before-operation plugin 1
    {
        returnType: NO, // reject
        level: 257, // level 257
        condition:
            operation === BATCH_DISABLE_PLUGIN &
            (
                disable_before_op_plugin_index === 0 |
                disable_before_op_plugin_index === 1 |
                disable_after_op_plugin_index === 0
            )
        votingRuleIndex: 0,  // no voting rule index needed
        note: "before-op plugin 2",
        bIsBeforeOperation: true  // the plugin will be added as before-op
    }
]);
```

5.3 Non-Voting Stocks

Non-voting stock, also known as preferred stock, is an ownership stake in a corporation that lacks voting rights but offers financial benefits like dividend priority, liquidation preference, and stability. It can be convertible or redeemable and is favored for its steady income. However, it provides no say in company decisions, making it a choice for income-focused investors seeking capital preservation with limited control over corporate matters. Specific terms vary among companies, so reviewing prospectuses is crucial for understanding the details.

In the DARC protocol, when a token is initialized with a dividend weight of 1 and a voting weight of 0, it can be classified as non-voting shares.

5.4 Stocks with Stamp Duty

Stocks with stamp duty are securities subject to a government tax, known as a "stamp duty", when they are bought or sold. This tax is applied to various financial instruments, including stocks and bonds, and is typically paid by the buyer or seller, depending on local regulations. People pay stamp duty when trading stocks because it serves as a source of government revenue and can also act as a tool to regulate financial markets and discourage excessive trading. The specific reasons and rates for stamp duty can vary by jurisdiction.

In the DARC protocol, we have the capability to create a mechanism that compels token holders to pay a transaction fee referred to as "stamp duty" when transferring tokens of varying levels. For instance, we can designate level-0 tokens as the standard stocks within the DARC protocol, with a fixed stamp duty of 1000 wei applied to each token transfer.

First, we must implement a plugin to disable all `BATCH_TRANSFER_TOKENS` operations when the token being transferred belongs to level-0. Next, we should develop a plugin that restricts the transfer of level-0 tokens exclusively through the `BATCH_PAY_TO_MINT_TOKENS` operation, provided each token's transaction fee is greater than or equal to 1000 wei. Additionally, users are required to set the `dividendable` flag of the `BATCH_PAY_TO_MINT_TOKENS` operation to 0 (false), ensuring that the transaction fee (stamp duty) will not be considered as dividendable revenue for the DARC.

```
batch_add_and_enable_plugins([
    // adding before-operation plugin 1: disable trasnfer tokens for level-0 tokens
    {
        returnType: NO, // reject
        level: 255, // level 255
        condition:
            operation === BATCH_TRANSFER_TOKENS &
            (
                transfer_tokens_level === 0
            )
```

```
        votingRuleIndex: 0,  // no voting rule index needed
        note: "disable trasnfer tokens for level-0 tokens",
        bIsBeforeOperation: true  // the plugin will be added as before-op
    },

    // adding before-operation plugin 2:
    // allow pay to transfer tokens for level-0 tokens,
    // with the transaction fee >= 1000 wei per token
    {
        returnType: YES_AND_SKIP_SANDBOX, // allow and skip sandbox
        level: 256, // level 256
        condition:
            operation === BATCH_PAY_TO_TRANSFER_TOKENS &
            (
                pay_to_transfer_tokens_level === 0  &
                transaction_fee_per_token >= 1000   &
                pay_to_trasnfer_tokens_dividendable === 0
            )
        votingRuleIndex: 0,  // no voting rule index needed
        note: "allow pay to transfer level-0 token with transaction fee
              > 1000 and dividendable flag 0",
        bIsBeforeOperation: true  // the plugin will be added as before-op
    }
]);
```

5.5 Board of Directors

In the realm of DAOs, decisions are typically reached through voting using governance tokens. However, this process can become sluggish and inefficient when involving thousands of token holders in each decision. Furthermore, the absence of regulations or predefined rules for the pending execution of proposal smart contracts means that approved votes can potentially authorize a wide array of actions within the DAO.

Contrastingly, in traditional joint-stock companies, day-to-day operations and affairs are overseen by a Board of Directors, rather than relying on decision-making by all shareholders. Boards of Directors are known for their efficiency, driven by their expertise, strategic acumen, rapid decision-making capabilities, and their ability to maintain stability and consistency in corporate governance. While shareholder voting remains vital for accountability and representation, the Board system offers a structured and informed approach to decision-making, ensuring that the company's long-term interests are thoughtfully prioritized and effectively managed.

Below is an example of how the DARC Y experiment designs its board of directors using a multi-level token system. There are five types of tokens, each with its unique characteristics and roles within the organization, as illustrated in Table 1.

Level 0 (Class A Token): These tokens, with a total supply of 400, possess both voting and dividend weights of 1. They likely represent common stakeholders within the DARC Y protocol.

Level 1 (Class B Token): With a total supply of 60, Class B Tokens hold a substantial voting weight of 10, making them influential in decision-making. They also carry a dividend weight of 1, implying a potential share in revenue distribution.

Level 2 (Independent Director): There is only one Independent Director token, designed with no voting or dividend weight, suggesting a unique role in the governance structure.

Level 3 (Board of Directors): Comprising five tokens, the Board of Directors holds a voting weight of 1 but lacks dividend weight, indicating its primary function in decision-making.

Level 4 (Executives): With three tokens, Executives share the same characteristics as the Board of Directors, holding a voting weight of 1 and no dividend weight.

Here are some guiding principles for designing the Board of Directors mechanism:

Principle 1: The Board of Directors should consist of a maximum of 5 members, and all board decisions must receive approval from more than 2/3 of its members.

Level	Name	Total Supply	Voting Weight	Dividend Weight
0	Class A Token	400	1	1
1	Class B Token	60	10	1
2	Independent Director	1	0	0
3	Board of Directors	5	1	0
4	Executives	3	1	0

Table 1: A structure of token distribution in DARC Y

Principle 2: If an address holds over 2/3 of the Class A tokens, it can represent the interests of retail investors and may be selected as a Class A board member. In the absence of any address holding over 2/3 of these tokens, there will be no Class A representative on the board.

Principle 3: The Board of Directors should always include one independent director, who has the option to appoint their successor. The removal of the independent director from the board is not permitted.

Principle 4: The board should comprise a maximum of 3 members, with the option to add any address holding more than 5% of the Class B tokens. Existing board members can nominate a Class B board member, subject to approval by at least 2/3 of all board members.

Principle 5: Existing board members have the authority to propose the removal of a Class B board member. Such a proposal can be approved under the following conditions: (1) The address to be removed holds a quantity less than or equal to 5% of the total supply of Class B tokens, or (2) The address does not possess more than 2/3 of the total supply of Class A tokens or the independent director token, and the proposal garners more than 2/3 of the voting power within the Board of Directors.

Principle 6: Class A Token directors, Class B Token directors, and the independent director should maintain independence from token holdings. This means that (1) Class A Token directors may only possess Class A tokens and Board of Directors tokens, and they are prohibited from holding Class B tokens or independent director tokens; (2) Class B Token directors may exclusively hold Class B tokens and Board of Directors tokens, refraining from ownership of Class A tokens or independent director tokens; and (3) The independent director is solely allowed to hold the independent director token, devoid of Class A or Class B tokens.

Next, we will design the following plugins to implement the rules and processes defined above:

Before-Operation Plugin Rule 1: If an operator address holds more than 2/3 of the level-0 tokens and does not hold any level-1, level-2, or level-3 tokens, it can mint one level-3 token to itself without requiring a sandbox check. This plugin rule is designed for the nomination of Class A token board members.

Before-Operation Plugin Rule 2: When an operator address intends to transfer a level-2 token to another target address, and the target address does not hold any level-0, level-1, or level-3 tokens, this operation can be approved without a sandbox check. This plugin facilitates the transfer of the independent director position.

Before-Operation Plugin Rule 3: In cases where an operator address holds 1 level-2 token and 0 level-3 tokens and wishes to mint a level-3 token for itself, this operation can be approved without any sandbox check. This plugin serves the purpose of nominating an independent director.

Before-Operation Plugin Rule 4: Operations involving the minting or burning of level-2 tokens need to be rejected. This plugin restricts the number of independent directors.

Before-Operation Plugin Rule 5: If an operator address intends to mint 1 level-3 token and the target address holds more than 5% of the total supply of level-1 tokens, and the operator address holds 1 level-3 token, this operation must undergo a sandbox check. This plugin is utilized for the nomination of Class B token board members.

Before-Operation Plugin Rule 6: In situations where an operator address seeks to burn a level-3 token from a target address, and the target address holds less than or equal to 2/3 of the total supply of level-0 tokens, holds less than or equal to 5% of level-1 tokens, and has 0 level-2 tokens, this operation can be approved without a sandbox check. This plugin facilitates the removal of unqualified board members.

Before-Operation Plugin Rule 7: If an operator address intends to burn a level-3 token from a target address, and the target address holds less than or equal to 2/3 of the total supply of level-

0 tokens, holds more than 5% of level-1 tokens, and has 0 level-2 tokens, and the operator address possesses at least 1 level-3 token, this operation must undergo a sandbox check. This plugin is employed for the removal of Class B token board members.

After-Operation Plugin Rule 1: When an operator address plans to burn a level-3 token from a target address, and the target address holds less than or equal to 2/3 of the total supply of level-0 tokens, holds more than 5% of level-1 tokens, has 0 level-2 tokens, and the operator address possesses at least 1 level-3 token, this operation must undergo a vote by voting rule 1. This vote involves all board members and requires an absolute majority mode, with an approval rate exceeding 66%. This plugin is utilized for Class B board member elections.

And next are the plugins implemented in the By-law Script.

```
const plugin_before_op_1 = {
    returnType: YES_AND_SKIP_SANDBOX,
    level: 255,
    condition: operation === BATCH_MINT_TOKENS
               & number_of_token_mint === 1
               & level_of_token_mint === 3
               & operator_owns_num_of_token(0) > 267
               & operator_owns_num_of_token(1) === 0
               & operator_owns_num_of_token(2) === 0
               & operator_mint_to_itself ,
    votingRuleIndex: 0,
    note: "before op 1",
    bIsBeforeOperation: true
},

const plugin_before_op_2 = {
    returnType: YES_AND_SKIP_SANDBOX,
    level: 255,
    condition: operation === BATCH_TRANSFER_TOKENS
               & transfer_token_level === 2 ,
    votingRuleIndex: 0,
    note: "before op 2",
    bIsBeforeOperation: true
},

const plugin_before_op_3 = {
    returnType: YES_AND_SKIP_SANDBOX,
    level: 255,
    condition: operation === BATCH_MINT_TOKENS
               & operator_owns_num_of_token(2) === 1
               & number_of_token_mint === 1
               & level_of_token_mint === 3 ,
    votingRuleIndex: 0,
    note: "before op 3",
    bIsBeforeOperation: true
},

const plugin_before_op_4 = {
    returnType: NO,
    level: 257,
    condition: (operation === BATCH_BURN_TOKENS
               & level_of_token_burned === 2) |
               (operation === BATCH_MINT_TOKENS
               & level_of_token_mint === 2) ,

    votingRuleIndex: 0,
```

```
    note: "before op 4",
    bIsBeforeOperation: true
},

const plugin_before_op_5 = {
    returnType: SANDBOX_NEEDED,
    level: 256,
    condition: operation === BATCH_MINT_TOKENS
               & level_of_token_mint === 3
               & number_of_token_mint === 1
               & operator_owns_num_of_tokens(3) === 1
               & batch_mint_tokens_target_address_ownes_num_of_tokens(1) > 12,
    votingRuleIndex: 0,
    note: "before op 5",
    bIsBeforeOperation: true
},

const plugin_before_op_6 = {
    returnType: YES_AND_SKIP_SANDBOX,
    level: 255,
    condition: operation === BATCH_BURN_TOKENS
               & level_of_token_burned === 3
               & batch_burn_token_target_address_owns_num_of_tokens(1) <= 12
               & batch_burn_token_target_address_owns_num_of_tokens(0) <= 267
               & batch_burn_token_target_address_owns_num_of_tokens(2) === 0,
    votingRuleIndex: 0,
    note: "before op 6",
    bIsBeforeOperation: true
},

const plugin_before_op_7 = {
    returnType: SANDBOX_NEEDED,
    level: 256,
    condition: operation === BATCH_BURN_TOKENS
               & level_of_token_burned === 3
               & batch_burn_token_target_address_owns_num_of_tokens(1) >= 12
               & batch_burn_token_target_address_owns_num_of_tokens(0) <= 267
               & batch_burn_token_target_address_owns_num_of_tokens(2) === 0,
    votingRuleIndex: 0,
    note: "before op 7",
    bIsBeforeOperation: true
},

const plugin_after_op_1 = {
    returnType: VOTE_NEEDED,
    level: 253,
    condition: operation === BATCH_BURN_TOKENS
               & level_of_token_burned === 3
               & batch_burn_token_target_address_owns_num_of_tokens(1) >= 12
               & batch_burn_token_target_address_owns_num_of_tokens(0) <= 267
               & batch_burn_token_target_address_owns_num_of_tokens(2) === 0
               & operator_ownes_num_of_tokens(3) === 1,
    votingRuleIndex: 1,
    note: "after op 1",
    bIsBeforeOperation: false
},
```

```
batch_add_and_enable_plugins([
    plugin_before_op_1,
    plugin_before_op_2,
    plugin_before_op_3,
    plugin_before_op_4,
    plugin_before_op_5,
    plugin_before_op_6,
    plugin_before_op_7,
    plugin_after_op_1
]);
```

5.6 Corporate Bonds

Company bonds are debt securities issued by corporations to raise funds. Investors lend money to the company in exchange for periodic interest payments and the return of their principal amount at maturity. They provide companies with a means to secure capital for various purposes and offer investors a predictable income stream.

In the context of the DARC protocol, the commands `BATCH_PAY_TO_MINT_TOKENS` and `BATCH_BURN_TOKENS_AND_REFUND` facilitate the creation of bond tokens available for purchase by any address during a specified time frame, with a designated price for both acquisition and redemption. Two rules govern the buying and refunding of these bond tokens:

Rule 1: Between January 1, 2020, and February 1, 2020, any address can purchase no more than 100 bond tokens(level-2) using `BATCH_PAY_TO_MINT_TOKENS` at a rate of 10,000 wei per token. The total supply of bond tokens should not exceed 10000.

Rule 2: Addresses have the option to redeem bond tokens at a rate of 15000 wei per token using `BATCH_BURN_TOKENS_AND_REFUND` after January 1, 2030.

```
batch_add_and_enable_plugins([
    {
        returnType: YES_AND_SKIP_SANDBOX,
        level: 253,
        condition: operation === BATCH_PAY_TO_MINT_TOKENS
                   & pay_to_mint_tokens_price_per_token === 10000
                   & timestamp > 1577858400 // 2020-01-01-0-0-0
                   & timestamp < 1580536800 // 2020-02-01-0-0-0
                   & pay_to_mint_tokens_level === 2 // level of token
                   & number_of_token_pay_to_mint <= 100 // number to mint
                   & total_number_to_tokens(2) <= 99900, // total token supply
        votingRuleIndex: 0,
        note: "",
        bIsBeforeOperation: true
    },
    {
        returnType: YES_AND_SKIP_SANDBOX,
        level: 253,
        condition: operation === BATCH_BURN_TOKENS_AND_REFUND
                   & burn_tokens_and_refund_price_per_token === 15000
                   & timestamp > 1893477600 // 2030-01-01-0-0-0
                   & burn_tokens_and_refund_level === 2, // level of token
        votingRuleIndex: 0,
        note: "",
        bIsBeforeOperation: true
    },
]);
```

5.7 Product Tokens and Non-Fungible Tokens

When both the voting weight and dividend weight are set to zero, and we allow any address to mint and transfer tokens at specific prices, these tokens can be treated as Non-Fungible Tokens (NFTs). NFTs are unique digital assets that cannot be exchanged on a one-to-one basis with other tokens. In this context, if we further restrict the total supply of certain token levels to just one, it enhances the uniqueness and rarity of these tokens, making them truly Non-Fungible Tokens. Each token becomes a distinct digital collectible, often used for representing ownership of unique digital or physical assets in a blockchain-based system.

6 Plugins

6.1 Design of Plugins

Plugin is the law in the DARC Protocol, and all programs and operations within DARC must adhere to the restrictions imposed by all plugins. For an individual plugin, it is required to follow the logic outlined in the pseudo code below:

```
if plugin.condition:
    return plugin.returnType
```

For the DARC protocol, the main difference between before-operation plugins and after-operation plugins lies in their return types. For before-operation plugins, as they determine whether a certain operation should be executed directly, rejected outright, or entered into a sandbox, they have three distinct return types that serve as their final decisions:

1. `NO`. When the condition of a before-operation plugin is triggered, the plugin's decision for that operation is `NO`. This decision indicates that the plugin believes the operation violates its rules, and therefore, it is rejected outright before entering the sandbox for execution.

2. `SANDBOX_NEEDED`. When the condition of a before-operation plugin is triggered, the plugin's decision for that operation is `SANDBOX_NEEDED`. This decision indicates that the plugin cannot determine whether the operation should be accepted or rejected. The plugin is aware that the operation needs to be evaluated in the sandbox by after-operation plugins, and thus, the decision is made to let the operation proceed to the sandbox for further evaluation.

3. `YES_AND_SKIP_SANDBOX`. When the condition of a before-operation plugin is triggered, the plugin's decision for that operation is `YES_AND_SKIP_SANDBOX`. This decision indicates that the plugin has determined that the operation should be approved and does not require execution in the sandbox. Therefore, the operation can proceed directly without going through the sandbox.

For after-operation plugins, since the program has been executed in the sandbox and voting can commence, these plugins can have three return types as their final decisions:

1. `NO`. When the condition of an after-operation plugin is triggered, the plugin's decision for the operation is `NO`. This decision indicates that the plugin believes the operation violates its rules and should be rejected outright.

2. `VOTING_NEEDED`. When the condition of an after-operation plugin is triggered, the plugin's decision for the operation is VOTING_NEEDED. This decision indicates that the plugin believes the operation requires a vote, and the operation needs to initialize a voting item based on the voting rule specified by this plugin.

3. `YES`. When the condition of an after-operation plugin is triggered, the plugin's decision for the operation is `YES`. This decision indicates that the plugin believes, based on its rules, the operation should be allowed to proceed.

Each plugin has a condition node array, where condition nodes are stored in sequence. The root node corresponds to the node at index 0, which is the first node. The condition node array follows the following principles:

1. The type of each node can be a boolean operator or an expression;
2. For boolean operators, the type must be set as one of AND, OR, or NOT;
3. For AND and OR operators, at least two valid child node indices must be specified in the child node list;
4. For the NOT operator, a unique child node index must be specified in the child node list;
5. For expression nodes, valid condition expression parameters consistent with the expression must be set;
6. For expression nodes, the length of their child node list must be 0, meaning no child nodes are allowed.

Figure 5 is an example illustrating how a condition expression binary tree is serialized into a condition node array.

Additionally, a plugin needs to set two parameters: one is the 'level', representing the priority of the plugin within the entire plugin system. For the same operation, the judgment system traverses all plugins, and it is possible that at least two or more plugins are triggered. In such cases, if the levels of the plugins are different, the judgment system will consider the plugin with the higher level as the final determination.

The other parameter is the 'voting rule index,' which points to a specific index in the voting rule array. When the final decision of a plugin is `VOTING_NEEDED`, the plugin automatically requests the DARC protocol to use the voting rule indicated by the voting rule index for initializing the voting item. If the return type of the plugin is not `VOTING_NEEDED`, the voting rule index will be ignored.

6.2 Plugins and the Judgement System

For the DARC protocol, the judgment system needs to undergo two assessments: one through before-operation plugins and another after the program has run completely in the sandbox, followed by evaluation through after-operation plugins. The reason for this design is that, without a sandbox and relying solely on a set of plugins for judgment, it becomes challenging to anticipate the behavior of the program. As a result, it is not possible to protect against modifications to special states in the DARC protocol.

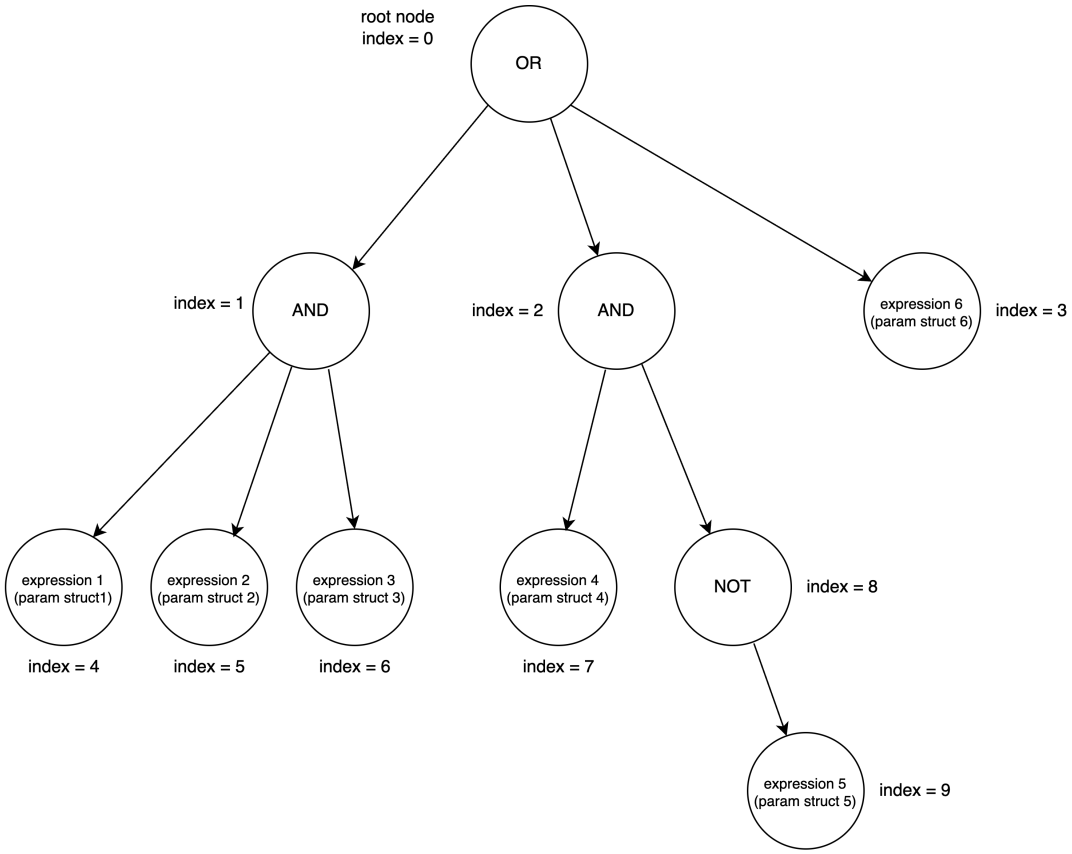
For example, in a DARC instance where shareholder X is required to permanently hold 15% voting rights and 10% dividend rights, when designing a plugin, it is impossible to predict the state of the DARC instance after the execution of operations like mint tokens or burn tokens. This uncertainty poses challenges in ensuring that shareholder X maintains permanent ownership of 15% voting rights and 10% dividend rights. Only by running the operation in the sandbox and then re-evaluating the state of the sandbox, can such modifications be prevented.

In another scenario, if there is a need to ensure that a DARC instance reserves 10,000 native tokens permanently before January 1, 2035, the correct detection and prevention of operations such as paying dividends or withdrawing cash can only be guaranteed by executing these operations in the sandbox. After all operations are completed in the sandbox, the judgment system performs a second evaluation through after-operation plugins. Without a sandbox and relying solely on plugins, the design of such a mechanism would be overly complex.

If there are only after-operation plugins and a sandbox without before-operation plugins, it would not be feasible. This is because the running cost of the sandbox is very high. It not only requires the program to run completely in the sandbox but also involves initializing the sandbox by fully replicating the internal state of the entire DARC instance. This process incurs a significant amount of gas fees.

For the majority of simple operations that can be approved without the need for running in the sandbox, it is more cost-effective to have rules established directly in before-operation plugins. These rules can leverage factors such as the operation, operator, timestamp, etc., for a straightforward approval or rejection, preventing them from entering the sandbox and saving costs.

For before-operation plugins, whenever a program is submitted to the DARC protocol, the judgment system sequentially checks each operation. For each operation, the judgment system traverses each before-operation plugin and obtains a single judgment result. Finally, it aggregates the results for all operations to determine the overall result for the entire program. This decision dictates whether



Node Array Index	0	1	2	3	4	5	6	7	8	9
Boolean Operator	OR	AND	AND	NULL	NULL	NULL	NULL	NULL	NOT	NULL
Expression	NULL	NULL	NULL	exp6	exp1	exp2	exp3	exp4	NULL	exp5
Parameter Struct	NULL	NULL	NULL	param6	param1	param2	param3	param4	NULL	param5
Child Node Index	[1,2,3]	[4,5,6]	[7,8]	[]	[]	[]	[]	[]	[9]	[]

Figure 5: Condition Nodes and Expression Tree

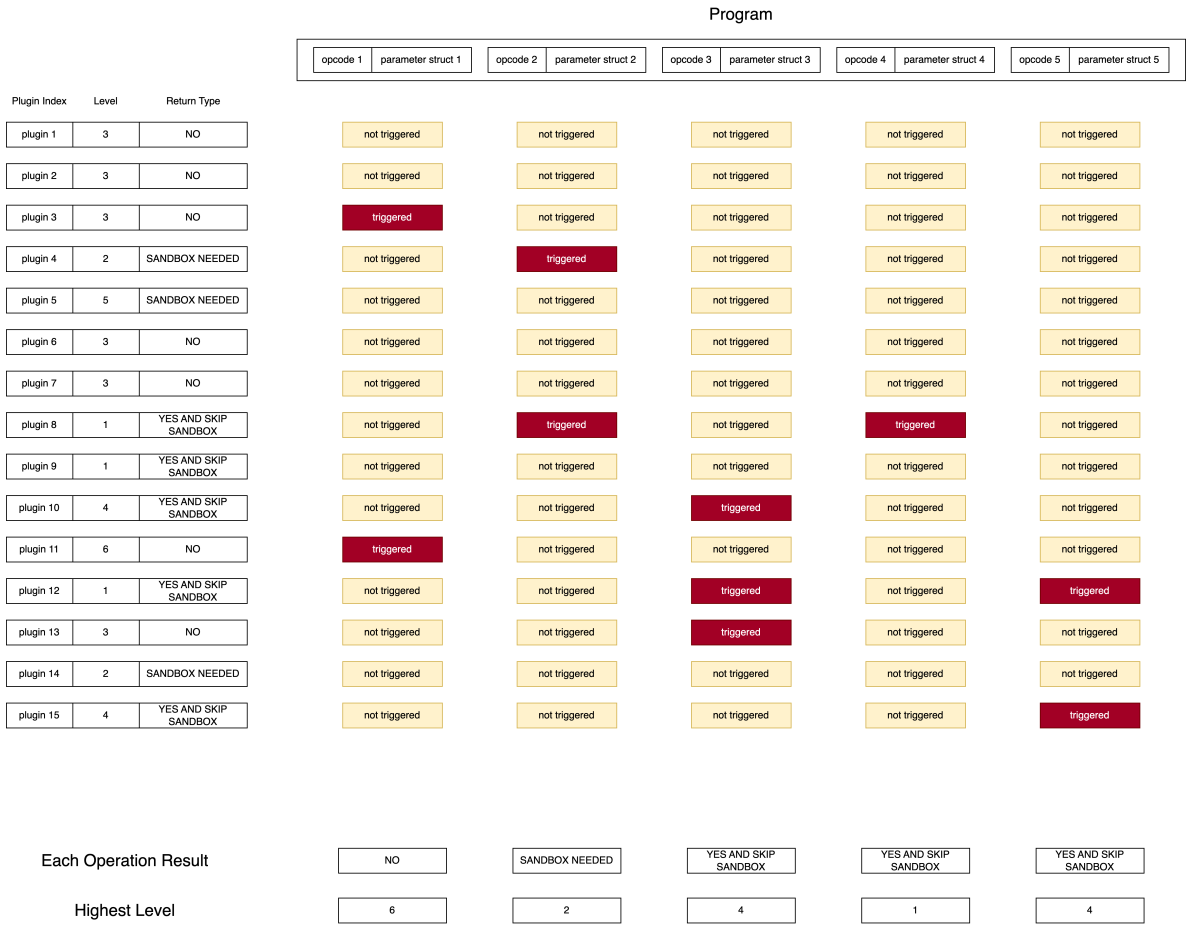


Figure 6: Judgement on program with before-operation plugins

the program needs to run in the sandbox (`SANDBOX_NEEDED`), be rejected outright (`NO`), or run directly without the need for the sandbox (`YES_AND_SKIP_SANDBOX`). For before-operation judgment, the following rules are followed:

1. If any operation is judged by the judgment system as `NO`, the entire program is rejected with a result of `NO`.

2. If none of the operations is judged by the judgment system as `NO`, and at least one operation is judged as `SANDBOX_NEEDED`, the entire program is required to run in the sandbox, and the result is `SANDBOX_NEEDED`.

3. If all operations are judged by the judgment system as `YES_AND_SKIP_SANDBOX`, the entire program is approved with a result of `YES_AND_SKIP_SANDBOX`, the entire program can skip the sandbox.

Figure 6 illustrates how individual operations within a program are judged by before-operation plugins, resulting in judgment outcomes.

For after-operation plugins, once a program has completed its full execution in the sandbox, the judgment system sequentially examines each operation. For each operation, the judgment system traverses each after-operation plugin, obtaining an individual judgment result. Ultimately, the results for all operations are aggregated to determine the final program result, deciding whether the program needs approval through voting (`VOTING_NEEDED`), should be rejected outright (`NO`), or can proceed directly (`YES`). The following rules apply to after-operation judgment:

1. If any operation is judged by the judgment system as `NO`, the entire program is rejected with a result of `NO`.

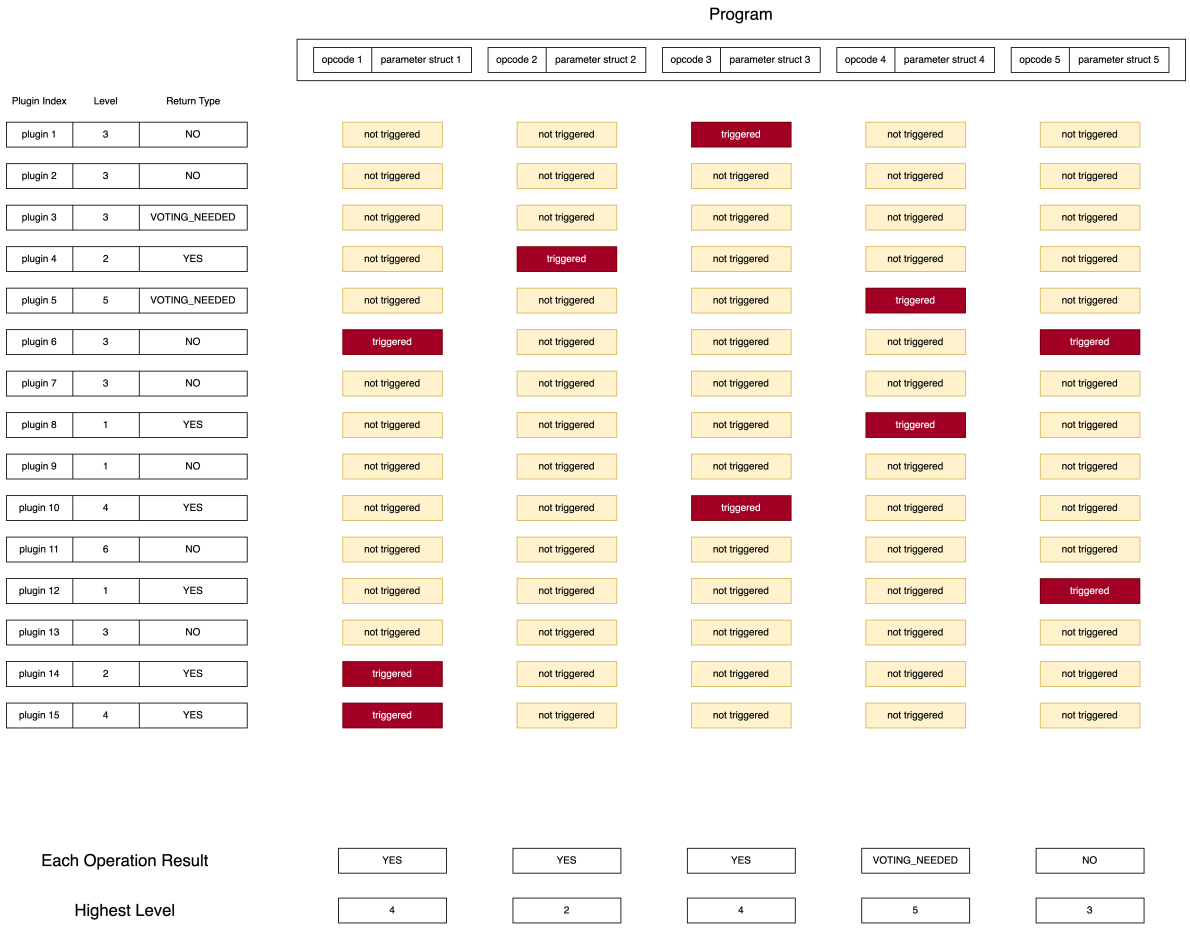


Figure 7: Judgement on program with after-operation plugins

2. If none of the operations have been determined as `NO` by the judgement system, and at least one operation has been determined as `VOTING_NEEDED`, the final judgement for the entire program is `VOTING_NEEDED`. This program will be placed into the pending program category, and the DARC protocol must initiate the voting system to decide on approval or rejection.

3. If all operations are judged by the judgment system as `YES`, the entire program is approved with a result of `YES`, the entire program can be executed directly.

Figure 6 illustrates how individual operations within a program are judged by after-operation plugins, resulting in judgment outcomes.

7 Voting

7.1 Voting Process and States

As shown in figure 8, you can consider each DARC protocol as a finite-state machine (FSM) with three states:

1. Idle state: In this state, the DARC protocol can accept any program. When a program is accepted and executed, the DARC protocol transitions back to the idle state. Similarly, if a program is rejected, the DARC protocol returns to the idle state. When a program contains an operation that requires voting, the DARC protocol transitions to the voting state.

2. Voting state: During the voting state of the DARC Protocol, all users are allowed and exclusively permitted to execute programs containing a single vote operation. If a program contains multiple operations, the program will be rejected. Similarly, if a program contains a single operation that is not

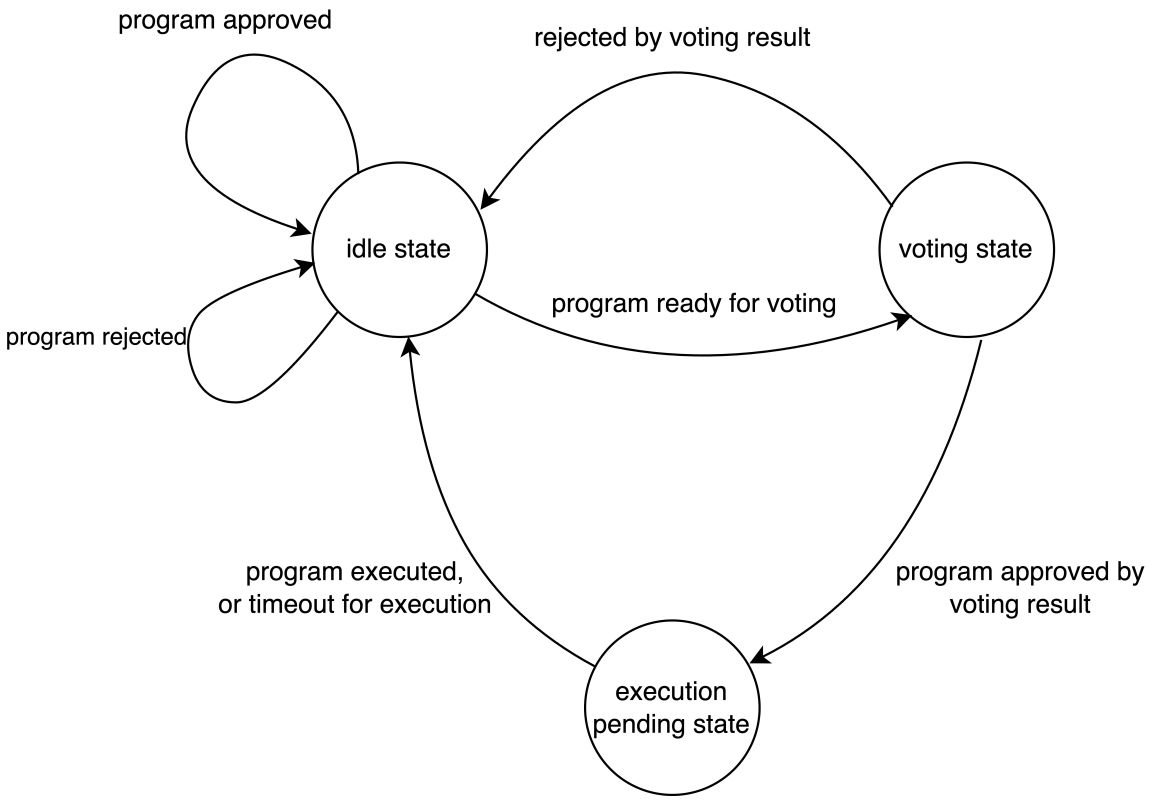


Figure 8: The finite-state machine of DARC

a vote, the program will also be rejected. If the current vote is approved, the system will automatically transition to the execution pending state. Additionally, the voting state has a minimum time limit. If, within this time limit, the supporting votes for each voting item do not collectively exceed the defined threshold, the current voting round is considered unsuccessful, and the system will automatically return to the idle state. Only when all voting items receive approval within the specified time limit will the program be approved by the DARC protocol.

3. Execution pending state: When a program is voted upon and approved by the DARC protocol, it can be executed in the execution pending state. The execution pending state also has a minimum time limit. If the program is executed within this limit, it will be successfully executed, and the system will automatically revert to the idle state. If execution does not occur within this limit, the program will fail to execute, and the DARC protocol will discard the program, automatically returning to the idle state.

7.2 Voting Rules

Each voting rule is composed with following items:

- Voting token class list: a list of token class indices that are allowed to vote for this voting item. It contains at least one valid token class index number. All tokens with indices listed in this array are considered as valid voting tokens. All token holders that holds any tokens that are included in this array are allowed to vote for this item.
- Approval threshold percentage: the approval threshold in percentage.
- Boolean flag `bIsAbsoluteMajority`: a boolean flag that indicates if the voting mode is by absolute majority or relative majority.
- Voting duration(in seconds): the maximum duration for this voting process in seconds. If the voting process takes more time than this parameter, the voting process will be terminated and the vote result will be set as failure.

- Executing pending duration(in seconds): the maximum duration for the operator to execute an approved program after a voting process finishes. If the program is approved, the operator must execute this program in a certain period of time, otherwise the program will be aborted and the DARC will be reset to idle state.
- Boolean flag `isForceStopAllowed`: If this flag is set as `True`, some operators can stop this voting process manually.
- Boolean flag `isEnabled`: a boolean flag that must be set as `True` if the voting rule is initialized and enabled successfully.
- Note: a string stored in the voting rule with extra information, comments or external URLs about this voting rule.

```
struct VotingRule {

  /**
   * the voting token class index list
   */
  uint256[] votingTokenClassList;

  /**
   * the approval threshold percentage of the voting policy
   */
  uint256 approvalThresholdPercentage;

  /**
   * the voting duration of the voting policy in seconds
   */
  uint256 votingDurationInSeconds;

  /**
   * the execution pending duration of the voting policy in seconds
   */
  uint256 executionPendingDurationInSeconds;

  /**
   *  the forced stop during the voting duration is allowed or not
   */
  bool isForcedStopAllowed;

  /**
   * the voting policy is enabled or not
   */
  bool isEnabled;

  /**
   * the note of the voting policy
   */
  string note;

  /**
   * the voting policy is absolute majority or relative majority.
   */
  bool bIsAbsoluteMajority;
}
```

7.3 Voting Types

Typically, in the voting system of DARC protocol, there are two types of voting methods:

- Absolute Majority: In absolute majority, the approval of a vote requires reaching a fixed threshold. Expressed as a percentage of the total voting weight, if the total weight of approval votes exceeds a predetermined threshold, the vote is considered passed. For example, with a total weight of 1000 and a threshold set at 70%, the vote is deemed successful only if the total weight of approval votes exceeds 700.

- Relative Majority: In relative majority, the approval of a vote is relative to the percentage of the total voting weight. In this case, the total voting weight can be a percentage of the actual cast votes, rather than a predefined absolute value. For instance, if the total weight is 1000 but only 300 weight is cast in the vote, relative majority requires that the approved vote weight exceeds 70% of the cast votes (0.7 * 300 = 210) for the vote to be considered successful.

We present the options of "absolute majority" and "relative majority" in voting due to the distinct emphases these two modes offer when balancing the outcomes of the vote.

With regards to the Absolute Majority, its advantage is that the approved voting weight only needs to constitute a proportion of the total possible vote, offering more flexibility. It is applicable in situations where a significant majority is sufficient to determine the outcome in order to promote broader consensus. Its application scenario is suitable for important decisions where it is crucial to secure support from a substantial majority. Using an absolute majority as the criterion serves to promote broader consensus.

As for the Relative Majority, its advantage is that the approved voting weight simply needs to constitute the largest share of the current vote, providing greater expediency and flexibility. It is thus applicable in situations where a relative majority is sufficient to determine the outcome. Its application scenario is suited for scenarios where a more flexible and expedited decision-making process is required. A relative majority can achieve consensus more quickly without needing to secure a fixed percentage of the total possible voting weight.

By providing both absolute and relative majority options, we enhance the adaptability of voting rules to serve the diverse governance needs of various protocols and organizations. Opting for the appropriate voting mode facilitates better fulfilling the consensus and efficiency requirements of particular decision scenarios.

7.4 Voting Mechanism

For each program, after receiving approval from the before-operation plugin and completing its full execution within the sandbox, the judgment system, guided by all after-operation plugins, evaluates each operation. If the judgments from all plugins at the highest level for a particular operation are marked as "VOTING_NEEDED", all the voting rules associated with that plugin are collected. Once the judgment system has compiled all the voting rules corresponding to each operation within the program, it generates a voting item array, assigning each rule to its respective entry.

For each operation, there may be one or more distinct plugins simultaneously triggered, all having a return type of "VOTING_NEEDED". In such cases, the voting rules associated with each plugin pointing to that operation will be gathered in order. On the other hand, for each program, different operations may be successively triggered by the same plugins. In the voting item array, a new voting item will be generated for each operation, pointing to the respective voting rule.

Figure 9 is an example illustrating how the DARC protocol initializes a voting item array based on judgment results.

Once a voting item array is initialized, as a finite state machine, the DARC protocol transitions into the voting state. Assuming there are N voting items in the voting item array, all operators will be allowed to cast their votes, submitting a boolean array of length N. Each boolean value represents the operator's support or opposition to the corresponding voting item. For each operator, in every voting state, they can and must vote only once, ensuring that the parameter in the vote operation is a boolean array of length N. If an operator attempts to execute a vote operation more than once or provides a boolean array of a length other than N, the voting program will be automatically rejected.

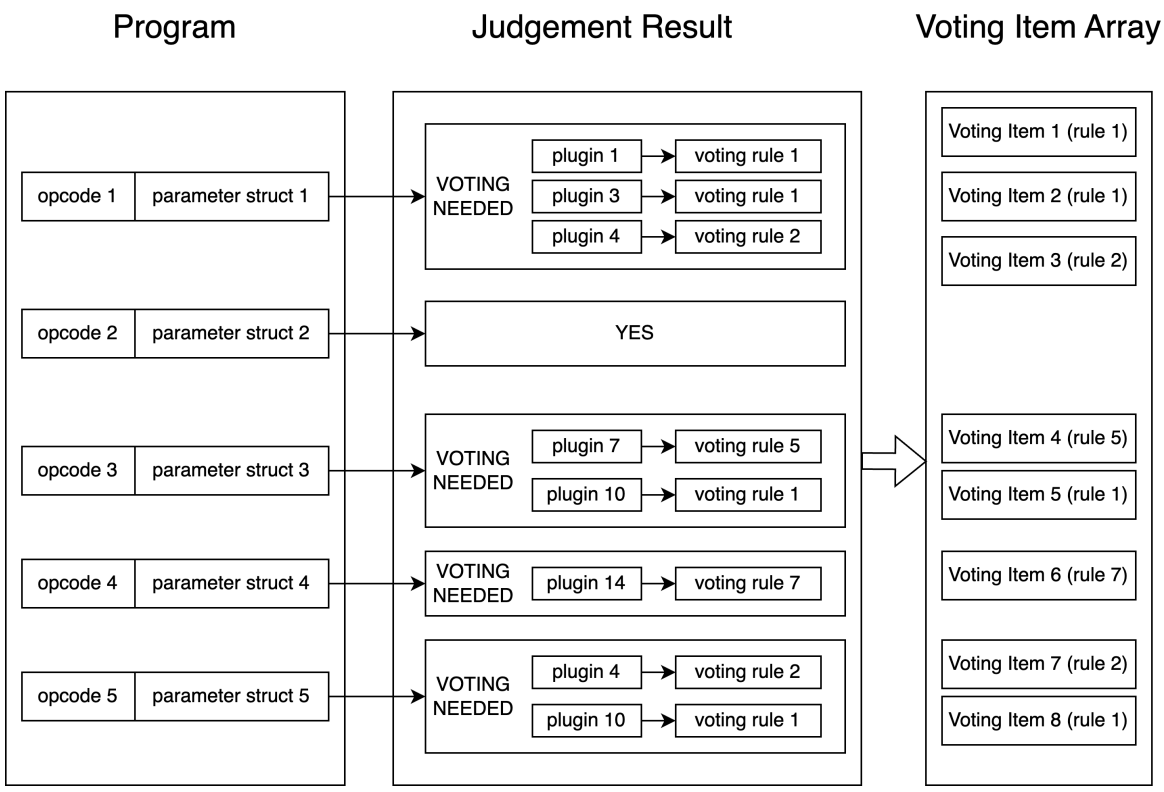


Figure 9: Voting Rules and Items

In the DARC protocol, each voting item has two counters: one for "YES" votes and another for "NO" votes. Whenever an operator performs a valid voting operation, each voting item calculates the total voting weight of that operator in the corresponding voting rule. In a given voting item, each voting rule defines a set of allowed token levels $C = [c_1, c_2, ..., c_n]$, where n is the number of token levels. Each token level c_i has a corresponding voting weight denoted as v_i. Suppose during the voting process, an operator chooses to vote on this voting item, the operator's voting weight W is calculated by the following formula:

$$W = \sum_{i=1}^{n} w_i v_i$$

In this formula, W represents the total voting weight of the operator for this voting item, and the summation goes through all token levels c_i allowed for voting. For each level i, v_i is the quantity of tokens at that level, and w_i is the corresponding voting weight for level i.

After each operator submits a valid vote operation, each voting item will increment the corresponding voting weight into the YES or NO counters. If the operator holds zero tokens in the votable token set for a particular voting item, neither the YES nor NO counters will be affected. Upon the completion of the voting process, the DARC protocol will determine whether the voting has concluded based on the following criteria:

1. When there is at least one voting item among all voting items operating in absolute majority mode, and this voting item has received a sufficient number of NO votes, the entire voting process will terminate prematurely, resulting in a voting failure. Subsequently, the DARC protocol transitions into the idle state. For instance, if a voting item has an approval threshold of 66%, operates in absolute majority mode, and has collected over 34% of NO votes before the deadline, the entire voting process will immediately conclude, resulting in a voting failure and transitioning into the idle state.

2. When all voting items exclusively operate in absolute majority mode, and each voting item

has garnered YES votes surpassing the approval threshold, the entire voting process concludes prematurely, indicating a successful vote. Subsequently, the DARC protocol transitions into the execution pending state.

3. When the voting state does not meet criteria 1 and 2, the DARC protocol will wait until the deadline expires for assessment. If, post-deadline, each voting item in absolute majority mode has received equal to or more than the total voting weight multiplied by the threshold in YES votes, and for each voting item in relative majority mode, the item has garnered equal to or more than the submitted voting weight multiplied by the threshold in YES votes, the vote is approved, and the DARC protocol transitions to the execution pending state. If any voting item fails to meet this criterion, the vote is deemed unsuccessful, and the DARC protocol reverts to the idle state.

After entering the execution pending state, the program stored in the current voting state has been approved and is ready for execution. The program must be completed before the new execution pending deadline. Upon successful execution, the DARC protocol transitions to the idle state. However, if an error or exception occurs during execution or if no one executes the program before the deadline, the DARC protocol also reverts to the idle state, and the program becomes unable to proceed with execution.

8 Memberships

In every company, in addition to members such as the board of directors, shareholders, executives, there are also numerous employees, contractors, interns, customers, suppliers, and so on. They may not hold any first-level tokens or company shares, but their payroll, orders, subscription fees, and other functionalities require bulk management. Therefore, membership is another set of convenient tools for use in the DARC protocol.

Table 2 is an example of a hierarchical membership structure of the DARC X, a consulting company. In this table, we define 6 levels: co-founder, substantial shareholder, executive, manager, employee and departed employee. And we can define some plugins for DARC X based on this membership table.

Rule 1: We need to limit the co-founders from selling a large amount of stock without notifications to the executives. If the operator is an address with a "co-founder" role and operation is selling more than 100000 level-0 tokens, this operation needs to be approved by all board members.

Rule 2: We provide 10 ETH per month as the salary for both employee-level and manager-level address (level 4 & 5).

Level	Role
1	Co-founder
2	Substantial Shareholder
3	Executive
4	Manager
5	Employee
6	Departed Employee

Table 2: A hierarchical membership structure

Now we analyze the rules above and design plugins in By-law Script.

For **Rule 1**, we need to make sure that if the current operation is "transfer token", the level of token is 0, and the number of token is more than 100000, and the address is in the membership table with a role level equals to 1, the program will be suspended and waiting for a voting process. The voting process is defined and saved as **voting rule 1**, which requires all the holders of level-3 token to vote in 1 hour. There are 5 level-3 tokens in total, and each board member holds 1 token. If all the 5 board members vote yes in 1 hour, this operation will be approved. Otherwise, the whole program will be rejected and the operation will fail.

Here is an example plugin for **Rule 1** designed in By-law Script:

Address	Role	Name	IsSuspended
0x0AC..03	1 (Co-founder)	Ann	No
0x156..21	2 (Substantial Sharehoulder)	Banana Capital	No
0x918..1B	3 (Executive)	Tom	No
0x4E1..90	4 (Manager)	Jack	No
0x510..0B	5 (Employee)	Bob	No
0x113..C7	6 (Departed Employee)	Tim	No

Table 3: A membership table of DARC X

```
const plugin_Rule_1 =
{
    condition: (operation_name === BATCH_TRANSFER_TOKENS)
               & ( transfer_token_level === 0 )
               & ( transfer_token_amount > 100000)
               & ( operator_membership_level === 1) ,
    return_type: VOTE_NEEDED,
    is_before_operation: false,
    return_level: 100,
    voting_rule_index: 1
}
```

For **Rule 2**, we need to make sure that if the operator is assigned with a role level equals 4 or 5, the operation is adding withdrawable balance, the total amount of operation is less than or equal to 10 ETH(or 10000000000000000000 wei), and the operator did the same operation in more than 30 days(or 2592000 seconds), this operation will be approved and sandbox check can be skipped for this operation.

Here is an example plugin for **Rule 2** designed in By-law Script:

```
const plugin_Rule_2 =
{
    condition: (operation_name === BATCH_ADD_WITHDRAWABLE_BALANCE)
               & ( total_add_withdrawable_balance <= 10000000000000000000)
               & ( last_operation_by_operation_period >= 2592000)
               & ( (operator_membership_level === 4) | (operator_membership_level === 5) ) ,
    return_type: YES_AND_SKIP_SANDBOX,
    is_before_operation: true,
    return_level: 100,
    voting_rule_index: 0
}
```

9 Dividends

The dividendable fund pool will contain a fraction of X per ten thousand, which can be used for dividend distribution. For the total amount T in the dividendable fund pool and the sum of dividend weights of various token levels within the current DARC protocol, we can determine the dividend amount for each dividend unit as follows:

$$u = \frac{XT}{10000W}$$

Here, u represents the dividend amount for each dividend unit, T is the total amount in the dividendable fund pool, and W represents the sum of dividend weights for all dividendable tokens across different token levels within the current DARC protocol.

For each user, assuming that the sum of dividend weights from various token levels that user i holds is D_i, the total dividend amount U_i that user i can receive in this round is given by:

$$U_i = \frac{D_i}{10000} u$$

For the DARC protocol, three dividend distribution methods can be employed:

Dividend based on the Number of Dividendable Transactions: The first approach involves distributing dividends based on the quantity of dividendable transactions. This is achieved by setting the dividend transaction cycle counter, N, to a reasonable value. Upon receiving at least N dividendable transactions, the instruction `OFFER_DIVIDENDS` becomes executable. Upon execution, the dividendable fund pool and dividendable transaction cycle counter are reset to zero, initiating a new count. This process repeats, waiting for the next N transactions and subsequent dividend distribution operation.

Time-Based Dividend Distribution: The second method entails dividend distribution based on time. Setting N to 1, an additional plugin can be introduced, allowing `OFFER_DIVIDENDS` to be invoked no less than S seconds after the previous call within the entire DARC instance. In this manner, dividends can be scheduled every 2 weeks, 4 weeks, 3 months, 6 months, or 1 year. This approach aligns more closely with traditional corporate dividends.

Dividend Distribution Based on the Total Amount in the Dividendable Fund Pool: The third method involves distributing dividends based on the total amount in the dividendable fund pool. With N set uniquely, an additional plugin enables the execution of `OFFER_DIVIDENDS` whenever the amount in the dividendable fund pool exceeds a specified threshold. Following this design, a dividend distribution can be triggered after obtaining Y dividendable native tokens.

Users can choose any of the above methods or design plugins to create alternative, practical, and rational dividend distribution patterns based on different DARC organizational structures and methods.

It is crucial to note that performing `OFFER_DIVIDENDS` for every received dividendable transaction may result in significant gas fee wastage. This is due to the potential time complexity of $O(MNP)$, where M represents the number of token levels, N is the number of tokens in each level, and P is the total number of token holders. Therefore, implementing an efficient dividend distribution mechanism to save on gas fees is imperative.

Additionally, funds in the dividendable fund pool are not locked within the smart contract by the DARC protocol. The protocol does not safeguard dividends in cash, and `OFFER_DIVIDENDS` only performs a calculation, storing each token holder's upcoming dividends in a withdrawable dividend balance. When the operator withdraws dividends, it is impossible if the DARC protocol lacks a sufficient quantity of native tokens. To protect the dividend rights of specific or all token holders, additional plugins must be designed.

10 Emergency Agent

In the DARC protocol, users frequently encounter various issues, such as:

1. Setting incorrect plugin parameters, triggering conditions, or performing erroneous token operations, rendering recovery impossible.
2. Locking certain critical operations in the DARC protocol, resulting in the permanent unavailability of functionalities within the protocol.
3. Human disputes leading to the organization's inability to operate, which cannot be resolved through plugins but may be addressed through manual litigation using documents, texts, evidence, etc.
4. Identifying vulnerabilities or facing attacks in the DARC protocol, necessitating an urgent pause and recovery of the protocol.
5. Addressing other potential technical issues or disputes that may arise.

In the DARC protocol, users can designate one or more emergency agents for emergency situations. In the event of unforeseen circumstances, users can invite emergency agents to intervene. An emergency agent functions as a super administrator with the authority to execute any operation within the DARC protocol without being subject to any limitations imposed by plugins. This role is crucial for addressing urgent or unresolved issues within the DARC protocol.

1. `addEmergency(emergencyAgentAddress)`: This command is used to add an emergency agent by providing the address of the emergency agent. Once added, the emergency agent gains superadmin privileges, allowing them to execute any operation in the DARC protocol without being restricted by plugins.

2. `callEmergency(emergencyAgentAddress)`: This command is employed to invoke an emergency agent by specifying the address of the emergency agent to be called. Upon invocation, the emergency agent can take necessary emergency measures to address unforeseen situations and perform operations to ensure the normal functioning of the DARC.

3. `endEmergency()`: This command is utilized to conclude the emergency state. Once the emergency situation is resolved or addressed, users can use this command to end the emergency state and restore normal DARC protocol operations.

For the DARC protocol and emergency agents, issues such as problem resolution, parameter modifications, asset transfers, evidence acquisition methods, and payment for service fees are not discussed in this document. These aspects constitute the core topics of discussions between the DARC protocol's key stakeholders, executives, and emergency agents.

11 Upgradablity

Once a compiled smart contract binary is deployed on an EVM-compatible blockchain, it becomes immutable, and any modifications to the binary are not possible. OpenZeppelin [Ope] utilizes a proxy-based approach for updates. The specific method involves deploying a proxy smart contract along with all the implementation smart contracts and setting an admin address. When there is a need to update the implementation contract, the admin can directly modify the proxy, redirecting the proxy to the new implementation contract address.

For the DARC protocol, the proxy upgrade pattern is not applicable because it requires setting an admin in the proxy. The admin in the DARC protocol would have complete control and modification rights, potentially overwriting all existing assets, plugins, and information, resulting in the admin's power surpassing that of all token holders at various levels. In a scenario where there is already an established corporate structure in a company, having a super administrator with the authority to modify all legal aspects, deal with all company shares, and manage assets would be a dangerous proposition.

The program entrance serves as the unified and exclusive entry point for the DARC protocol. When upgrading from an old DARC instance X to a new DARC instance Y, the process is streamlined. This involves configuring the upgrade address for the old DARC instance X and setting up the new DARC instance Y to accept all delegate programs from DARC instance X. With these adjustments, the upgrade is smoothly executed. After the upgrade, users can seamlessly continue executing programs in DARC instance X, and these programs will be delegated and executed within the new DARC instance Y.

In the DARC protocol, there are three operations related to upgrades:

- `upgradeToAddress(targetAddress)`: Upgrades the current DARC instance to the targetAddress. All programs submitted to the current DARC instance will be delegated and executed in the DARC instance at the targetAddress.

- `confirmUpgradedFromAddress(sourceAddress)`: Allows the current DARC instance to upgrade from the DARC instance at sourceAddress. When a program is proxied and submitted from sourceAddress, it is permitted to run in the current DARC instance with the operator identity of the program submitter.

- `upgradeToTheLatest()`: If the current DARC instance A has been upgraded to the new DARC instance B, and the new DARC instance B has also been upgraded to the updated DARC instance C, executing this function will directly upgrade the DARC instance to DARC instance C. In other words, programs executed in this DARC instance A will be directly delegated and executed in DARC instance C, bypassing DARC instance B. This operation will be interpreted differently. Assuming a DARC instance A has already undergone an upgrade to DARC instance B, in such

a scenario, if a user executes a program in A with only one `upgradeToTheLatest()` operation, the program will be executed in DARC instance A instead of being delegated to DARC instance B.

With the three operations mentioned above, we can discuss two scenarios for upgrading the DARC protocol:

In the first scenario, where the user has already deployed My DARC 1, and My DARC 1 has not been upgraded to any other DARC, the following three steps are necessary:

1. Deploy the new version of My DARC 2 to the blockchain and complete all configurations.
2. Execute `confirmUpgradedFromAddress(MyDARC1Addr)` in My DARC 2, allowing My DARC 2 to accept programs and act as a proxy for My DARC 1.
3. Execute `upgradeToAddress(MyDARC2Addr)` in My DARC 1, completing the upgrade from My DARC 1 to My DARC 2.

Figure 10 illustrates the entire process of upgrading directly from My DARC 1 to My DARC 2.

In the second scenario, where the user has deployed My DARC 1, My DARC 1 has already been upgraded to My DARC 2, and the user intends to further upgrade to My DARC 3, the following three steps are required:

1. Deploy the new version of My DARC 3 on the blockchain and complete all configurations.
2. Execute `confirmUpgradedFromAddress(MyDARC1Addr)` in My DARC 3, enabling My DARC 3 to accept programs and proxies from My DARC 1.
3. Execute `upgradeToAddress(MyDARC2Addr)` in My DARC 1. Since this program will run in My DARC 2 under the proxy of My DARC 1, the `upgrade_address` of My DARC 2 will point to the address of My DARC 3.
4. Execute `upgradeToTheLatest()` in My DARC 1, directing My DARC 1 to the address of My DARC 3 based on the address of My DARC 2, completing the upgrade from My DARC 1 to My DARC 3.
5. Finally, close My DARC 2.

Figure 11 illustrates the entire process of upgrading directly from My DARC 1 to My DARC 3.

References

[Ope] "Proxy Upgrade Pattern", OpenZeppelin Docs, https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies.

Appendix 1: Reference Design of Instruction Opcodes

```
enum EnumOpcode {

/**
 * @notice Invalid Operation
 * ID: 0
 */
UNDEFINED,

/**
 * @notice Batch Mint Token Operation
 * @param ADDRESS_2DARRAY[0] address[] toAddressArray: the array of the address to mint
new token to
```

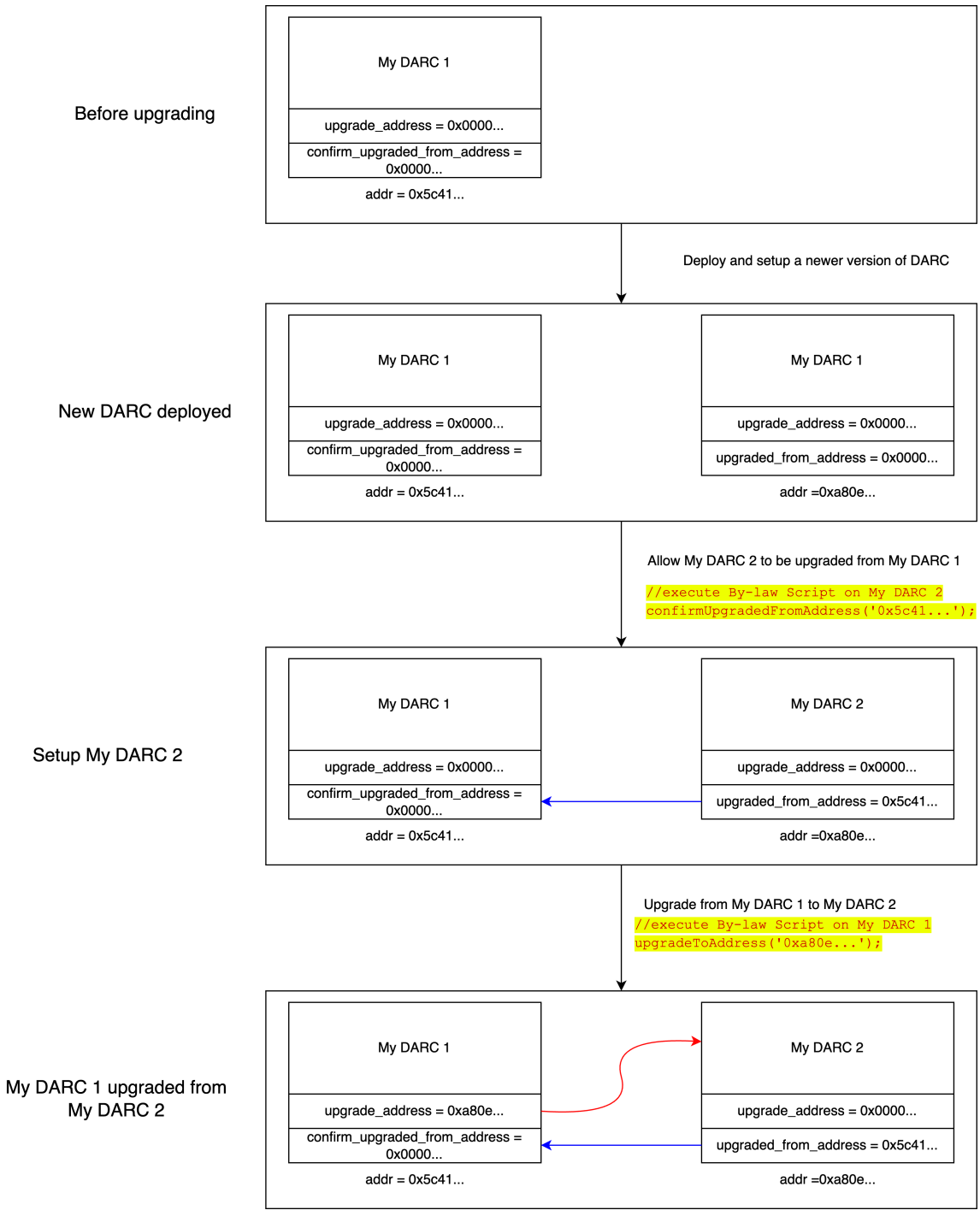


Figure 10: Upgrading from My DARC 1 to My DARC 2

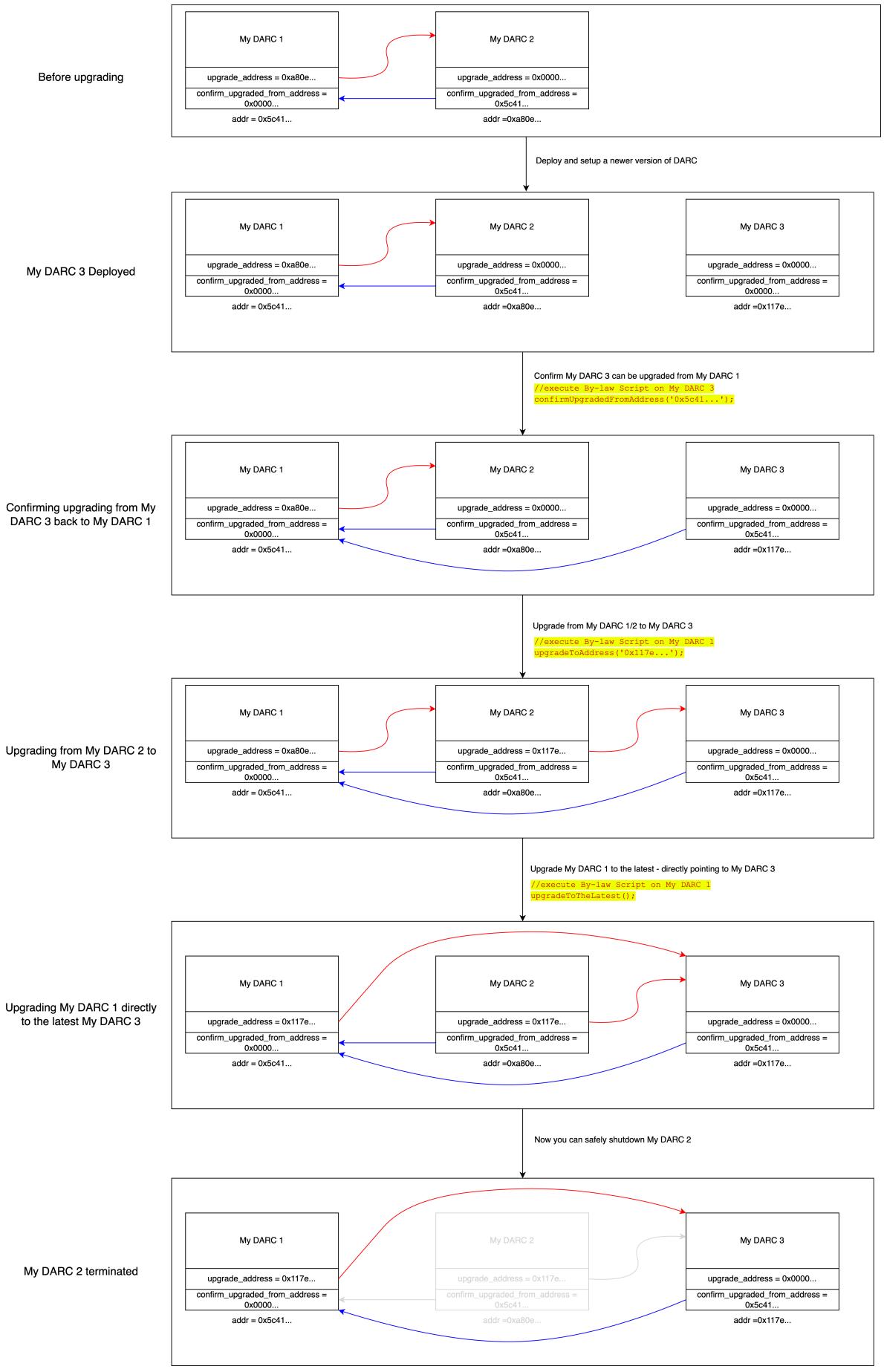


Figure 11: Upgrading from My DARC 1 to My DARC 2 to My DARC 3

```
   * @param UINT256_2DARRAY[0] uint256[] tokenClassArray: the array of the token class
index to mint new token from
   * @param UINT256_2DARRAY[1] uint256[] amountArray: the array of the amount of the token
to mint
   *
   * ID: 1
   */
  BATCH_MINT_TOKENS,

  /**
   * @notice Batch Create Token Class Operation
   * @param STRING_ARRAY[] string[] nameArray: the array of the name of the token class
to create
   * @param UINT256_2DARRAY[0] uint256[] tokenIndexArray: the array of the token index
of the token class to create
   * @param UINT256_2DARRAY[1] uint256[] votingWeightArray: the array of the voting weight
of the token class to create
   * @param UINT256_2DARRAY[2] uint256[] dividendWeightArray: the array of the dividend
weight of the token class to create
   *
   * ID:2
   */
  BATCH_CREATE_TOKEN_CLASS,

  /**
   * @notice Batch Transfer Token Operation
   * @param ADDRESS_2DARRAY[0] address[] toAddressArray: the array of the address to transfer
token to
   * @param UINT256_2DARRAY[0] uint256[] tokenClassArray: the array of the token class
index to transfer token from
   * @param UINT256_2DARRAY[1] uint256[] amountArray: the array of the amount of the token
to transfer
   *
   * ID:3
   */
  BATCH_TRANSFER_TOKENS,

  /**
   * @notice Batch Transfer Token From Addr A to Addr B Operation
   * @param ADDRESS_2DARRAY[0] address[] fromAddressArray: the array of the address to
transfer token from
   * @param ADDRESS_2DARRAY[1] address[] toAddressArray: the array of the address to transfer
token to
   * @param UINT256_2DARRAY[0] uint256[] tokenClassArray: the array of the token class
index to transfer token from
   * @param UINT256_2DARRAY[1] uint256[] amountArray: the array of the amount of the token
to transfer
   *
   * ID:4
   */
  BATCH_TRANSFER_TOKENS_FROM_TO,

  /**
   * @notice Batch Burn Token Operation
   * @param UINT256_2DARRAY[0] uint256[] tokenClassArray: the array of the token class
index to burn token from
```

```
   * @param UINT256_2DARRAY[1] uint256[] amountArray: the array of the amount of the token
to burn
   *
   * ID:5
   */
  BATCH_BURN_TOKENS,

  /**
   * @notice Batch Burn Token From Addr A Operation
   * @param ADDRESS_2DARRAY[0] address[] fromAddressArray: the array of the address to
burn token from
   * @param UINT256_2DARRAY[0] uint256[] tokenClassArray: the array of the token class
index to burn token from
   * @param UINT256_2DARRAY[1] uint256[] amountArray: the array of the amount of the token
to burn
   *
   * ID:6
   */
  BATCH_BURN_TOKENS_FROM,

  /**
   * @notice Batch Add Member Operation
   * @param ADDRESS_2DARRAY[0] address[] memberAddressArray: the array of the address
to add as member
   * @param UINT256_2DARRAY[0] uint256[] memberRoleArray: the array of the role of the
member to add
   * @param STRING_ARRAY string[] memberNameArray: the array of the name of the member
to add
   *
   * ID:7
   */
  BATCH_ADD_MEMBERSHIP,

  /**
   * @notice Batch Suspend Member Operation
   * @param ADDRESS_2DARRAY[0] address[] memberAddressArray: the array of the address
to suspend as member
   *
   * ID:8
   */
  BATCH_SUSPEND_MEMBERSHIP,

  /**
   * @notice Batch Resume Member Operation
   * @param ADDRESS_2DARRAY[0] address[] memberAddressArray: the array of the address
to reinstate as member
   *
   * ID:9
   */
  BATCH_RESUME_MEMBERSHIP,

  /**
   * @notice Batch Change Member Role Operation
   * @param ADDRESS_2DARRAY[0] address[] memberAddressArray: the array of the address
to change role of as member
   * @param UINT256_2DARRAY[0] uint256[] memberRoleArray: the array of the role of the
```

```
member to change
   *
   * ID:10
   */
  BATCH_CHANGE_MEMBER_ROLE,

  /**
   * @notice Batch Change Member Name Operation
   * @param ADDRESS_2DARRAY[0] address[] memberAddressArray: the array of the address
to change name of as member
   * @param STRING_ARRAY string[] memberNameArray: the array of the name of the member
to change
   *
   * ID:11
   */
  BATCH_CHANGE_MEMBER_NAME,

  /**
   * @notice Batch Add Emergency Agent Operation
   * @param Plugin[] pluginList: the array of the plugins
   * ID:12
   */
  BATCH_ADD_PLUGIN,

  /**
   * @notice Batch Enable Plugin Operation
   * @param UINT256_ARRAY[0] uint256[] pluginIndexArray: the array of the plugins index
to enable
   * @param BOOL_ARRAY bool[] isBeforeOperationArray: the array of the flag to indicate
if the plugin is before operation
   * ID:13
   */
  BATCH_ENABLE_PLUGIN,

  /**
   * @notice Batch Disable Plugin Operation
   * @param UINT256_ARRAY[0] uint256[] pluginIndexArray: the array of the plugins index
to disable
   * @param BOOL_ARRAY bool[] isBeforeOperationArray: the array of the flag to indicate
if the plugin is before operation
   * ID:14
   */
  BATCH_DISABLE_PLUGIN,

  /**
   * @notice Batch Add and Enable Plugin Operation
   * @param Plugin[] pluginList: the array of the plugins
   * ID:15
   */
  BATCH_ADD_AND_ENABLE_PLUGIN,

  /**
   * @notice Batch Set Parameter Operation
   * @param MachineParameter[] parameterNameArray: the array of the parameter name
   * @param UINT256_2DARRAY[0] uint256[] parameterValueArray: the array of the parameter
value
```

```
   * ID:16
   */
  BATCH_SET_PARAMETER,

  /**
   * @notice Batch Add Withdrawable Balance Operation
   * @param address[] addressArray: the array of the address to add withdrawable balance
   * @param uint256[] amountArray: the array of the amount to add withdrawable balance
   * ID:17
   */
  BATCH_ADD_WITHDRAWABLE_BALANCE,

  /**
   * @notice Batch Substract Withdrawable Balance Operation
   * @param address[] addressArray: the array of the address to substract withdrawable
balance
   * @param uint256[] amountArray: the array of the amount to substract withdrawable balance
   * ID:18
   */
  BATCH_SUBSTRACT_WITHDRAWABLE_BALANCE,

  /**
   * @notice Batch Add Voting Rules
   * @param VotingRule[] votingRuleList: the array of the voting rules
   * ID:19
   */
  BATCH_ADD_VOTING_RULE,


  /**
   * @notice Batch Pay to Mint Tokens Operation
   * @param ADDRESS_2DARRAY[0] address[] addressArray: the array of the address to mint
tokens
   * @param UINT256_2DARRAY[0] uint256[] tokenClassArray: the array of the token class
index to mint tokens
   * @param UINT256_2DARRAY[1] uint256[] amountArray: the array of the amount to mint
tokens
   * @param UINT256_2DARRAY[2] uint256[] priceArray: the price of each token class to
mint
   * @param UINT256_2DARRAY[3] uint256[1] dividendableFlag: the flag to indicate if the
payment is dividendable. 1 for yes (pay for purchase), 0 for no (pay for investment)
   * ID:20
   */
  BATCH_PAY_TO_MINT_TOKENS,

  /**
   * @notice Pay some cash to transfer tokens (can be used as product coins)
   * @param ADDRESS_2DARRAY[0] address[] toAddressArray: the array of the address to transfer
token to
   * @param UINT256_2DARRAY[0] uint256[] tokenClassArray: the array of the token class
index to transfer token from
   * @param UINT256_2DARRAY[1] uint256[] amountArray: the array of the amount of the token
to transfer
   * @param UINT256_2DARRAY[2] uint256[] priceArray: the price of each token class to
transfer
   * @ param UINT256_2DARRAY[3] uint256[1] dividendableFlag: the flag to indicate if the
```

```
payment is dividendable. 1 for yes (pay for purchase), 0 for no (pay for investment)
   * ID:21
   */
  BATCH_PAY_TO_TRANSFER_TOKENS,

  /**
   * @notice Add an array of address as emergency agents
   *  (can be used as product NFTs with a new unique token class)
   * @param ADDRESS_2DARRAY[0] address[] The array of the address to add as emergency
agents
   * ID:22
   */
  ADD_EMERGENCY,

  /**
   * @notice withdraw cash from the contract's cash balance
   * @param address[] addressArray: the array of the address to withdraw cash to
   * @param uint256[] amountArray: the array of the amount of cash to withdraw
   * ID:23
   */
  WITHDRAW_CASH_TO,

  /**
   * @notice Call emergency agents to handle emergency situations
   * @param UINT256_2DARRAY[0] address[] addressArray: the array of the emergency agents
index to call
   * ID:24
   */
  CALL_EMERGENCY,


  /**
   * @notice Call a contract with the given abi
   * @param address contractAddress: the address of the contract to call
   * @param bytes abi: the abi of the function to call
   * ID:25
   */
  CALL_CONTRACT_ABI,

  /**
   * @notice Pay some cash
   * @param uint256 amount: the amount of cash to pay
   * @param uint256 paymentType: the type of cash to pay, 0 for ethers/matic/original
tokens
   *  1 for USDT, 2 for USDC (right now only 0 is supported), 3 for DAI ...
   * @param uint256 dividendable: the flag to indicate if the payment is dividendable,
   * 0 for no (pay for investment), 1 for yes (pay for purchase)
   * ID:26
   */
  PAY_CASH,

  /**
   * @notice Calculate the dividends and offer to token holders
   *  by adding the dividends to the withdrawable balance of each token holder
   *
   * ID:27
```

```
   */
  OFFER_DIVIDENDS,

  /**
   * @notice Withdraw dividends from the withdrawable dividends balance
   * @param address[] addressArray: the array of the address to withdraw dividends to
   * @param uint256[] amountArray: the array of the amount of dividends to withdraw
   * ID:28
   */
  WITHDRAW_DIVIDENDS_TO,

  /**
   * @notice Set the approval for all transfer operations by address
   * @paran address: the address to set approval for all transfer operations
   * ID:29
   */
  SET_APPROVAL_FOR_ALL_OPERATIONS,


  /**
   * @notice Batch Burn tokens and Refund
   * @param UINT256_2D[0] uint256[] tokenClassArray: the array of the token class index
to burn tokens from
   * @param UINT256_2D[1] uint256[] amountArray: the array of the amount of the token
to burn
   * @param UINT256_2D[2] uint256[] priceArray: the price of each token class to burn
   * ID:30
   */
  BATCH_BURN_TOKENS_AND_REFUND,

  /**
   * @notice Add storage IPFS hash to the storage list permanently
   * @paran STRING_2DARRAY[0] address: the address to set approval for all cash withdraw
operations
   * ID:31
   */
  ADD_STORAGE_IPFS_HASH,


  /**
   * Below are two operations than can be used during voting pending process
   */

  /**
   * @notice Vote for a voting pending program
   * @param bool[] voteArray: the array of the vote for each program
   * ID:32
   */
  VOTE,

  /**
   * @notice Execute a program that has been voted and approved
   * ID:33
   */
  EXECUTE_PROGRAM,
```

```
  /**
   * @notice Emergency mode termination. Emergency agents cannot do anything after this
operation
   * ID:34
   */
  END_EMERGENCY,

  /**
   * @notice Upgrade the contract to a new contract address
   * @param ADDRESS_2DARRAY[0][0] The address of the new contract
   * ID:35
   */
  UPGRADE_TO_ADDRESS,

  /**
   * @notice Accepting current DARCs to be upgraded from the old contract address
   * @param ADDRESS_2DARRAY[0][0] The address of the old contract
   * ID:36
   */
  CONFIRM_UPGRAED_FROM_ADDRESS,

  /**
   * @notice Upgrade the contract to the latest version
   * ID:37
   */
  UPGRADE_TO_THE_LATEST
}
```

Appendix 2: Reference Design of Program and Operation

```
/**
  * The parameter(s) or operand(s) of the operation
  */
struct Param {
  uint256[] UINT256_ARRAY;
  address[] ADDRESS_ARRAY;
  string[] STRING_ARRAY;
  bool[] BOOL_ARRAY;
  VotingRule[] VOTING_RULE_ARRAY;
  Plugin[] PLUGIN_ARRAY;
  MachineParameter[] PARAMETER_ARRAY;
  uint256[][] UINT256_2DARRAY;
  address[][] ADDRESS_2DARRAY;
}

/**
  * The operation to be executed, including the operator address, the opcode and the parameters
  */
struct Operation {
  address operatorAddress;
  EnumOpcode opcode;
  Param param;
}
```

```
/**
  * The program to be executed, including the operator address and the operation array
  */
struct Program {
  address programOperatorAddress;

  /**
   * @notice operations: the array of the operations to be executed
   */
  Operation[] operations;
}
```

Appendix 3: Reference Design of Plugin

```
/**
  * The condition node types
  */
enum EnumConditionNodeType { UNDEFINED, EXPRESSION, LOGICAL_OPERATOR, BOOLEAN_TRUE, BOOLEAN_FALSE}

/**
  * The logical operator types
  */
enum EnumLogicalOperatorType {UNDEFINED, AND, OR, NOT }


enum EnumReturnType {

  /**
   * The default value. The plugin system will return UNDEFINED if no plugin is triggered.
   * Both BEFORE and AFTER operation plugin system may return UNDEFINED.
   */
  UNDEFINED,


  /**
   * The operation is approved but must be executed in sandbox to check if the operation
   * is valid in the current machine state.
   * Only BEFORE operation plugin system may return SANDBOX_NEEDED.
   */
  SANDBOX_NEEDED,

  /**
   * The operation is disapproved and should be rejected at this level.
   * Both BEFORE and AFTER operation plugin system may return NO.
   */
  NO,

  /**
   * The decision is pending and a voting item should be created at this level.
   * Only AFTER operation plugin system may return VOTE_NEEDED.
   */
  VOTE_NEEDED,

  /**
```

```
   * The operation is approved and should skip the sandbox check.
   * Only BEFORE operation plugin system may return YES_AND_SKIP_SANDBOX.
   */
  YES_AND_SKIP_SANDBOX,

  /**
   * The operation is finally approved at this level.
   * Only AFTER operation plugin system may return YES.
   */
  YES
}

/**
  * The condition node expression parameters
  */
struct NodeParam {
  uint256[] UINT256_ARRAY;
  address[] ADDRESS_ARRAY;
  string[] STRING_ARRAY;
  uint256[][] UINT256_2DARRAY;
  address[][] ADDRESS_2DARRAY;
  string[][] STRING_2DARRAY;
}

/**
  * The condition node struct
  */
struct ConditionNode {
  /**
   * current condition node index
   */
  uint256 id;

  /**
   * the type of current condition node
   */
  EnumConditionNodeType nodeType;

  /**
   * the logic operator of the current condition node
   */
  EnumLogicalOperatorType logicalOperator;

  /**
   * the condition expression of the current condition node
   */
  EnumConditionExpression conditionExpression;

  /**
   * a list of the child nodes of the current condition node
   */
  uint256[] childList;

  /**
   * The array of the EXPRESSION node parameters
   */
```

```
  NodeParam param;
}

/**
  * The struct of the plugin
  */
struct Plugin {
  /**
   * the return type of the current condition node
   */
  EnumReturnType returnType;

  /**
   * the level of restriction, from 0 to the maximum value of uint256
   */
  uint256 level;

  /**
   * condition binary expression tree vector
   */
  ConditionNode[] conditionNodes;

  /**
   * the voting rule id of the current plugin if the return type is VOTE_NEEDED
   */
  uint256 votingRuleIndex;

  /**
   * the plugin note
   */
  string note;

  /**
   * the boolean that indicates whether the plugin is enabled or not
   */
  bool bIsEnabled;

  /**
   * the boolean that indicates whether the plugin is deleted or not
   */
  bool bIsInitialized;

  /**
   * the boolean that indicates whether the plugin is a before operation
   * plugin or after operation plugin
   */
  bool bIsBeforeOperation;

}
```