Audit Report

Produced by CertiK

for 

Oct 18, 2019

CertiK Audit Report For V Systems

Request Date: 2019-09-19
Revision Date: 2019-10-18
Platform Name: VSYS Chain

CERTIK

Contents

Disclaimer

This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and V Systems(the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement"). This Report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This Report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, has developed a proprietary Formal Verification technology to apply rigorous and complete mathematical reasoning against code. This process ensures algorithms, protocols, and business functionalities are secured and working as intended across all platforms.

CertiK differs from traditional testing approaches by employing Formal Verification to mathematically prove blockchain ecosystem and smart contracts are hacker-resistant and bug-free. CertiK uses this industry-leading technology together with standardized test suites, static analysis, and expert manual review to create a full-stack solution for our partners across the blockchain world to secure 6.2B in assets.

For more information: https://certik.org/

Executive Summary

This report has been prepared as the product of the Smart Contract Audit request by V Systems. This audit was conducted to discover issues and vulnerabilities in the source code of V Systems's Smart Contracts. Utilizing CertiK's Formal Verification Platform, Static Analysis, and Manual Review, a comprehensive examination has been performed. The auditing process pays special attention to the following considerations.

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessment of the codebase for best practice and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line by line manual review of the entire codebase by industry experts.

Vulnerability Classification

For every issue found, CertiK categorizes them into 3 buckets based on its risk level:

Critical

The code implementation does not match the specification, or it could result in loss of funds for contract owner or users.

Medium

The code implementation does not match the specification at certain conditions, or it could affect the security standard by lost of access control.

Low

The code implementation is not a best practice, or use a suboptimal design pattern, which may lead to security vulnerabilies, but no concern found yet.

Vulnerability Details

Critical

No issues found.

Medium

No issues found.

Low

File	Issues
Contract.scala	No issues found
ContractPermitted.scala	No issues found
DataEntry.scala	No issues found
DataType.scala	No issues found
ExecutionContext.scala	No issues found
ContractApiRoute.scala	No issues found
ContractBroadcastApiRoute.scala	No issues found
ExecuteContractFunctionRequest.scala	No issues found
RegisterContractRequest.scala	No issues found
SignedExecuteContractFunctionRequest.scala	No issues found
SignedRegisterContractRequest.scala	No issues found
ExecuteContractFunctionTransaction.scala	No issues found
RegisterContractTransaction.scala	No issues found
AssertOpcDiff.scala	No issues found
CDBVOpcDiff.scala	No issues found
CDBVROpcDiff.scala	No issues found
LoadOpcDiff.scala	No issues found
OpcDiff.scala	No issues found
OpcDiffer.scala	No issues found
OpcFuncDiffer.scala	No issues found
ReturnOpcDiff.scala	No issues found
TDBAOpcDiff.scala	No issues found
TDBAROpcDiff.scala	No issues found
TDBOpcDiff.scala	No issues found
TDBROpcDiff.scala	No issues found
ContractAccount.scala	No issues found
RegisterContractTransactionDiff.scala	No issues found
ExecuteContractFunctionTransactionDiff.scala	No issues found

Introduction

V SYSTEMS, a distributed database project using cutting edge blockchain technology that allow all economic systems can be build their app on top of the platform.
Certik was chosen by VSYSTEMS team for reviewing their Non-Turing-Complete, smart contract technology development. The development is planned into three phrase:

1. Token creation, distribution, and issuance
2. Token trading and management
3. Optimize the performance

The phase Token creation, distribution, and issuance, is conducted under this audit review.
Below reference, taking from VSYS Smart Contract Wiki page:

> Considering the technology development and industrial needs for smart contracts, V SYSTEMS will temporarily adopt the non-Turing-complete scripting language, so that smart contracts can be secure, resource-efficient, and easy to use and manage. In the near future, a Turing-complete model will eventually be adopted by V SYSTEMS.
>
> - Smart contract ownership cannot be transferred, but the token issue right can be transferred. The contract creator has the final right to interpret the token issue right.
> - The smart contract itself cannot be modified. It is a simple consensus and cannot be modified at will, but the parameters of some contracts can be changed. The contracts with modifiable parameters are relatively weak in consensus. These parameter revisions will provide choices and an advanced notice.

Scope of Audit:

CertiK was chosen by V Systems to audit the design and implementation of its smart contract technology based on VSYS chain. To ensure comprehensive protection,the source code has been manually reviewed by our smart contract experts and engineers. That end-to-end process ensures proof of stability as well as a hands-on, engineering-focused process to close potential loopholes and recommend design changes in accordance with the best practices in the space.

Source Code SHA-256 Checksum

(commit aa95a9a50d08bd58767936a5188a570b95e3e73d)
blockchain/contract/

- **Contract.scala**
  `ea17f9d3a61d7fb4f3532a34eb1766daf8cf00113fbf3a95e050f7dd8b9528a5`
- **ContractPermitted.scala**
  `a1e03d3ba33b0448ca99fd2d348040536b4f738741bb7708eaaac909f71df67e`

- **DataEntry.scala**
  7d9d86764f388f2fd2a6e89ef4878a050f7246dbe21ad908b97b43638721b2d3
- **DataType.scala**
  587372c2832e5b367b8d559482f629d88582cb30da4e37715b01da3641b2b019
- **ExecutionContext.scala**
  cb8206a24c19c2bdbf320760a469c52ba9d166b289285d00ff112fabd903b482

api/http/contract/

- **ContractApiRoute.scala**
  3a75d8353f30eb7b3a7f097d839c6f3b3f4c430e45711172cdf2091c671158c1
- **ContractBroadcastApiRoute.scala**
  aabac651e6cafef44d658f58a02c0a363bdf80734aa557a58d8ec1219c6045d9
- **ExecuteContractFunctionRequest.scala**
  6e2957d58a82affede23db454d39b9065d35a56915ba849e9b3f9d7de566b595
- **RegisterContractRequest.scala**
  97838059cbc3f2783071f9d5c06de8dd47bda610461ca300c69e889a21ac0c19
- **SignedExecuteContractFunctionRequest.scala**
  e2fdcfc9c9644bb7135d79475ed595a331433fa15ed52df9dd7be72a1ee0cb76
- **SignedRegisterContractRequest.scala**
  e64b3dbad7067703ac63682d0a59b927ad7691a56d71bb99cc356a1cb7a7050f

blockchain/transaction/contract/

- **ExecuteContractFunctionTransaction.scala**
  4b4f9d170ffe3ef6d4edf9f9dd58ade9ecc8ac627287651f09c21233e6db48eb
- **RegisterContractTransaction.scala**
  8fc701c4ddbc3cef85378b810c8a15b1587afb182b95a7eb216d62f4fd924845

blockchain/state/opcdiffs/

- **AssertOpcDiff.scala**
  eea7670f42afb32283e430925e70eb787fe9f70be0dba743dce8e3fdb5c14d29
- **CDBVOpcDiff.scala**
  02e998f4407ae5b316d0f01b740f7c2b68adc59a549fd3ec6992f9f266593d49
- **CDBVROpcDiff.scala**
  b3fe5ae68095a1dcaf3dc3394e7798382b5ab88369bc7f37d959b6b4ea48cda1
- **LoadOpcDiff.scala**
  6a6ac9dfb431dcf940f3d9e2d97df3c97be82433e8182c9a52365bd1587236b7
- **OpcDiff.scala**
  827c5e725229b1d38585ba02846e6da1a99d04e4212f1e061e71ca95cea8c486

- **OpcDiffer.scala**
  `6a70fdd1b1537e596838aa198239fc10e5ffaa75af417b281d92af48df30d12c`
- **OpcFuncDiffer.scala**
  `c2adeaa6cc2cfb52562bd4f989197e7043fe5488fd79728b950cb4ad948308ea`
- **ReturnOpcDiff.scala**
  `3a10244ead7797ee4922e1f965f233aed8fae5459a5a3fa394e5e51c485326d6`
- **TDBAOpcDiff.scala**
  `f4b6b58e30d1aaebb8b356a5a719ddaa71e09ab356b8703ec8c6f5eade8e387a`
- **TDBAROpcDiff.scala**
  `bff9cdd08b59adbaebdf422416a825821a09282373ee55f412090a940743537b`
- **TDBOpcDiff.scala**
  `45d312184aee4a4835321445093aa5488a5359e1af9ba3189ef1eaed48e6f171`
- **TDBROpcDiff.scala**
  `d40dfbb90ea9e6224e7e63537a05cfb0b24b225ea4530853a1198ad0109f45ba`

account/

- **ContractAccount.scala**
  `fd35cd7ce31e9c8d6e88199eb7d1d513192a36392b4162555d76d6311e0fc01e`

blockchain/state/diffs/

- **RegisterContractTransactionDiff.scala**
  `2b5fc9e4bcdeeb898c0260411547210112f75dd434527133ec155bde2112f842`
- **ExecuteContractFunctionTransactionDiff.scala**
  `1937afbdc50cba9ca16df0f0dfeb4a6e1d8ed3b344b743afaa28962393704d64`

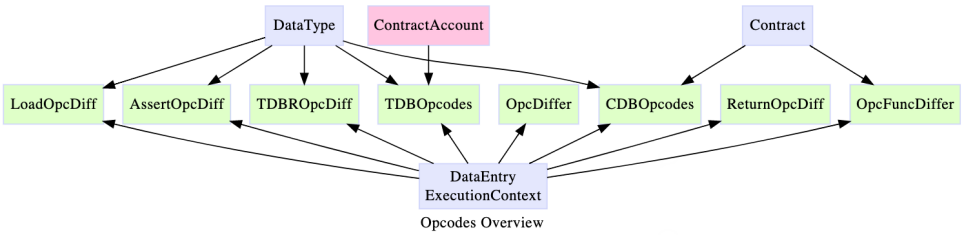
Architect & Workflow Overview

Structure:

```
contract:
    language code
    language version
    initializer: //executed when register contract
        init()
    descriptor: //executed by ExecuteContractFunctionTransaction()
        supersede()
        issue()
        destroy()
        split()
        send()
        transfer()
        deposit()
        withdraw()
        totalSupply()
        maxSupply()
        balanceOf()
        getIssuer()
    stateVar: //2 bytes array: 1st byte: idx, 2nd byte: type
        issuer
        maker
        max
        total
        unity
        shortText
    texture: //function name \& parameters
```

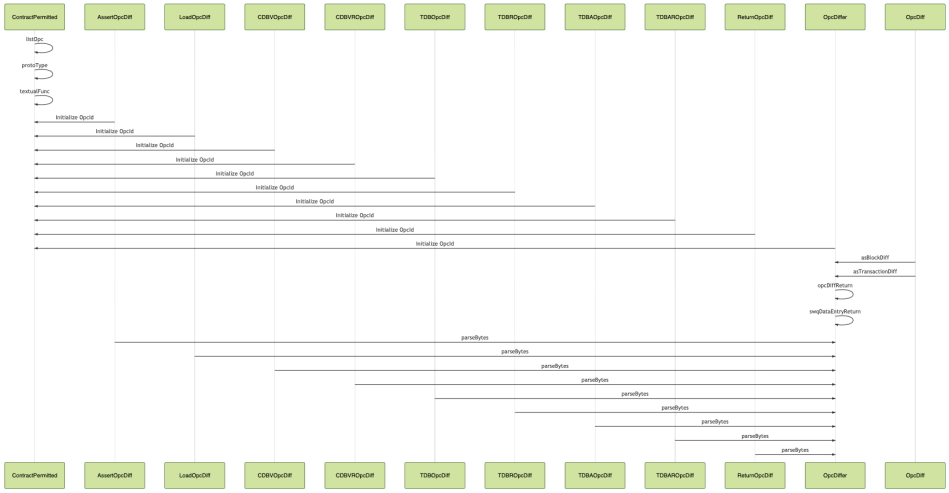
Opcode Workflow

Opcodes would take `DataEntry` and `ExecutionContext` as input parameter. `DataType` would be checked in `LoadOpcDiff`, `AssertOpcDiff`, `TDBOpcodes` and `CDBOpcodes` to ensure the operations are performed by authorized and valid parameters.



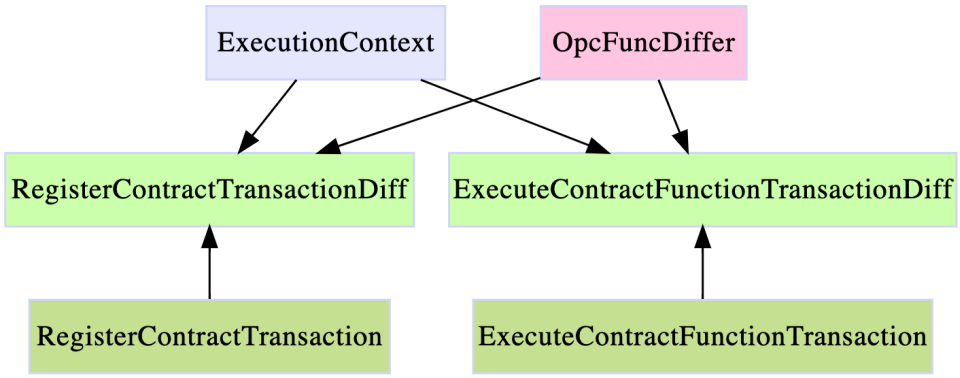
Opcodes Overview

Opcodes would have their `opcId` initialized in `Contracted`. `OpcDiffer` would extract the opcodes by function `parseBytes()`.



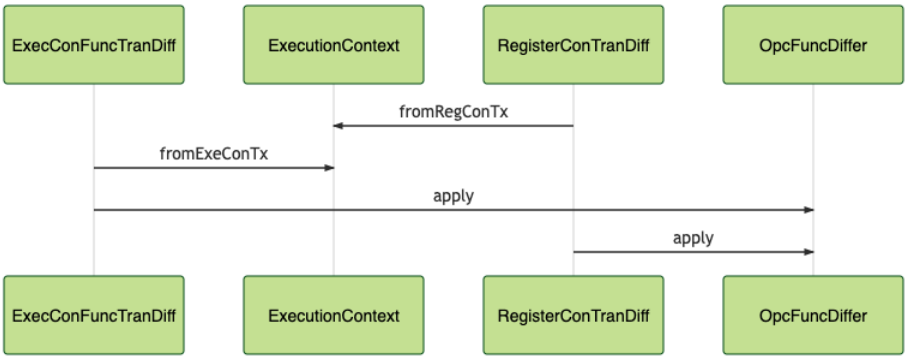
Transaction Workflow

`RegisterContractTransaction` and `ExecuteContractFunctionTransaction` would be used as input parameter, as transactions, for `RegisterContractTransactionDiff` and `ExecuteContractFunctionTransactionDiff` respectively. These two classes would read states from contract transactions and calculate the difference to proceed the states updates.



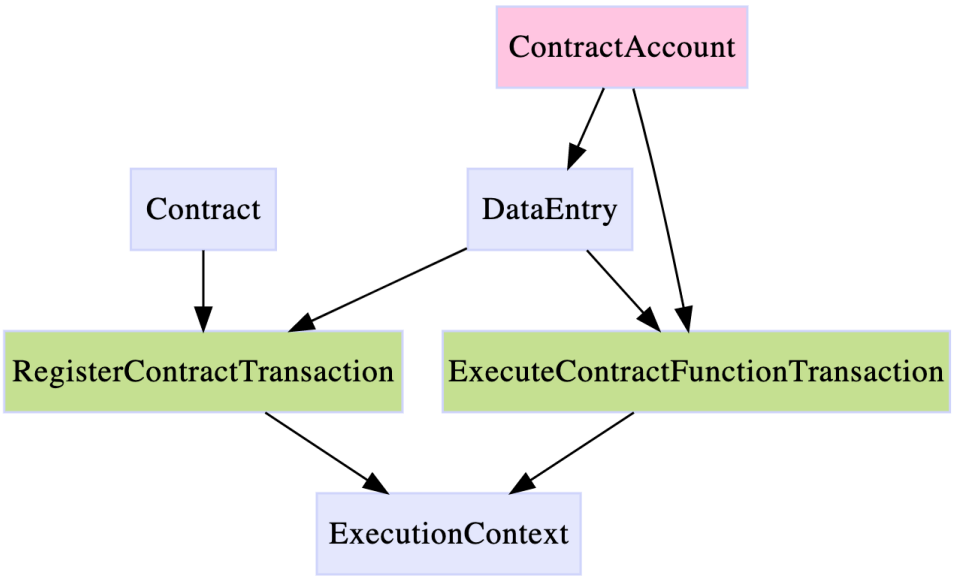
Transaction Overview

Function `apply)()` in the two transacton diff objects call `fromRegConTx()` and `from ExeConTx()` to convert `RegisterContractTransactions` and `ExecuteContractFunctionTransactions` to `ExecutionContext`. Then by calling `OpcFuncDiffer.apply()`, the two transaction diff objects get the opcodes to use the contract and token data.



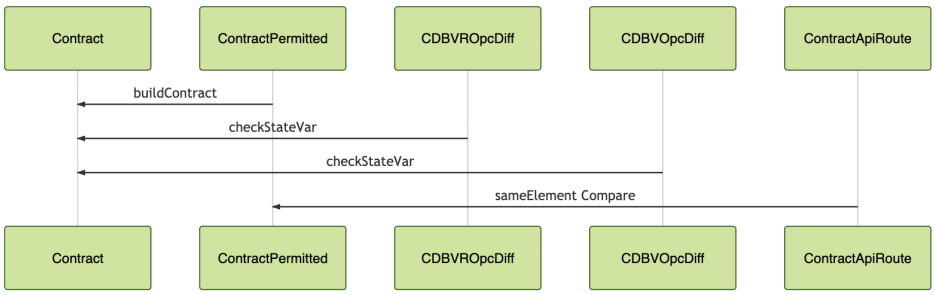
Contract Workflow

Class `RegisterContractTransaction` takes `Contract` and `DataEntry` as input parameters for its functions. Class `ExecuteContractFucntionTransaction` takes `DataEntry` and `ContractAccount` as input parameters for its functions. `ExecutionContext` would then be used to transfer data from transactions to opcodes and thus have the states update.



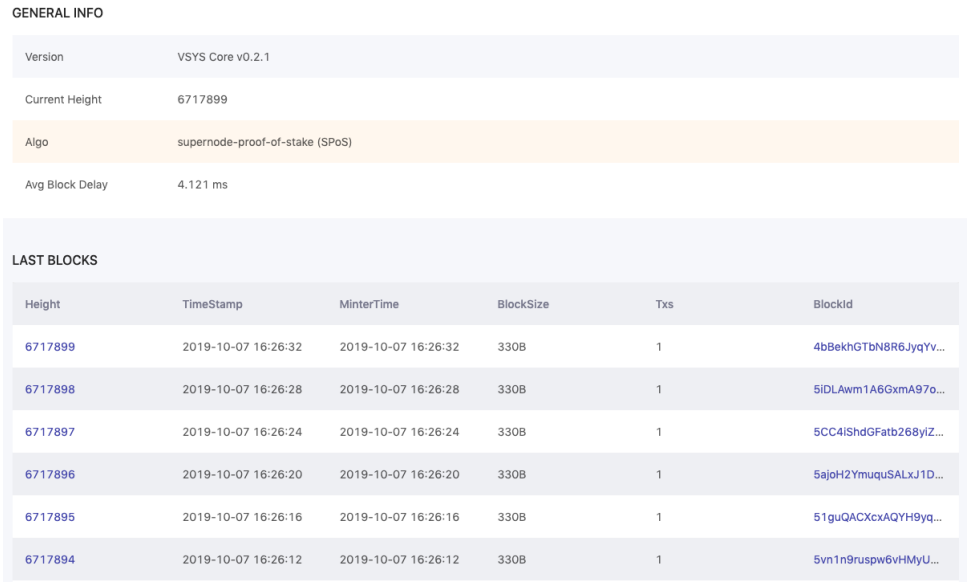
Contract-Transaction Overview

Function `buildContract` would take parameters of `languageCode`, `languageVersion`, `trigger`, etc. to build a `ContractImpl`. In `ContractPermitted`, more constants and state varibales would be defined as valid values for contracts.



Business Logic Workflow

CertiK engineers experience with VSYS chain testnet for diving deep about the VSYS chain core business logic workflow. We created few vsys accounts for interacting on vsys chain token contract creation, below is some of the highlighted findings.
The block generate rate of VSYS chain is relatively fast. The rate of new block would be 4 seconds per block. The average block delay would be 4 to 4.5 ms. The transfer completion time would be 0.8 to 1.2 seconds. Notice that the speed would differ for different generators.



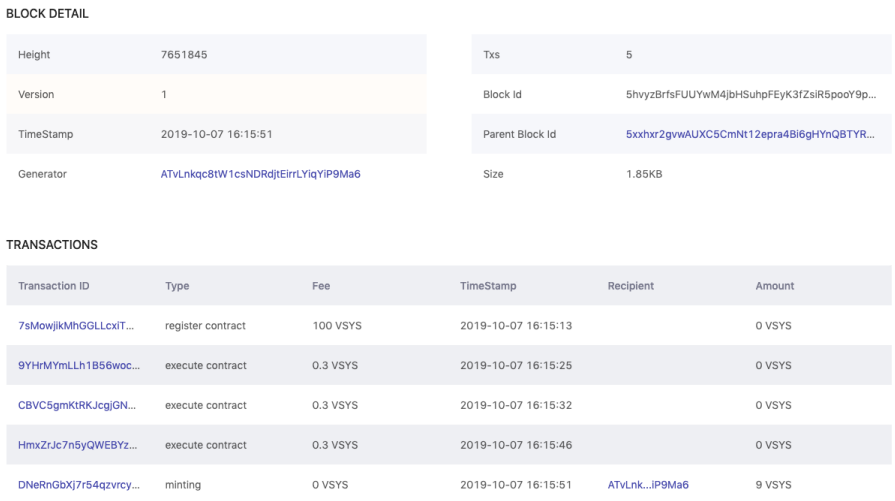
GENERAL INFO

Version	VSYS Core v0.2.1
Current Height	6717899
Algo	supernode-proof-of-stake (SPoS)
Avg Block Delay	4.121 ms

LAST BLOCKS

Height	TimeStamp	MinterTime	BlockSize	Txs	BlockId
6717899	2019-10-07 16:26:32	2019-10-07 16:26:32	330B	1	4bBekhGTbN8R6JyqYv...
6717898	2019-10-07 16:26:28	2019-10-07 16:26:28	330B	1	5iDLAwm1A6GxmA97o...
6717897	2019-10-07 16:26:24	2019-10-07 16:26:24	330B	1	5CC4iShdGFatb268yiZ...
6717896	2019-10-07 16:26:20	2019-10-07 16:26:20	330B	1	5ajoH2YmuquSALxJ1D...
6717895	2019-10-07 16:26:16	2019-10-07 16:26:16	330B	1	51guQACXcxAQYH9yq...
6717894	2019-10-07 16:26:12	2019-10-07 16:26:12	330B	1	5vn1n9ruspw6vHMyU...

For testnet block which height is 7651845, except minting, there are four transactions.



BLOCK DETAIL

Height	7651845	Txs	5
Version	1	Block Id	5hvyzBrfsFUUYwM4jbHSuhpFEyK3fZsiR5pooY9p...
TimeStamp	2019-10-07 16:15:51	Parent Block Id	5xxhxr2gvwAUXC5CmNt12epra4Bi6gHYnQBTYR...
Generator	ATvLnkqc8tW1csNDRdjtEirrLYiqYiP9Ma6	Size	1.85KB

TRANSACTIONS

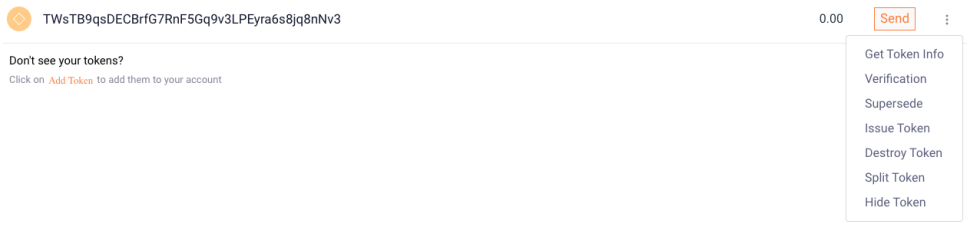
Transaction ID	Type	Fee	TimeStamp	Recipient	Amount
7sMowjikMhGGLLcxiT...	register contract	100 VSYS	2019-10-07 16:15:13		0 VSYS
9YHrMYmLLh1B56woc...	execute contract	0.3 VSYS	2019-10-07 16:15:25		0 VSYS
CBVC5gmKtRKJcgjGN...	execute contract	0.3 VSYS	2019-10-07 16:15:32		0 VSYS
HmxZrJc7n5yQWEBYz...	execute contract	0.3 VSYS	2019-10-07 16:15:46		0 VSYS
DNeRnGbXj7r54qzvrcy...	minting	0 VSYS	2019-10-07 16:15:51	ATvLnk...iP9Ma6	9 VSYS

Take the transaction of type register contract as an example, which transaction address is `7sMowjikMhGGLLcxiTDxgDWdPEdF4P96Xjgk1GWJP6KH`.

TRANSACTION DETAIL -- 7sMowjikMhGGLLcxiTDxgDWdPEdF4P96Xjgk1GWJP6KH

Type	register contract
Id	7sMowjikMhGGLLcxiTDxgDWdPEdF4P96Xjgk1GWJP6KH
TimeStamp	2019-10-07 16:15:13
BlockHeight	7651845
Explain	Create token(TWsTB9qsDECBrfG7RnF5Gq9v3LPEyra6s8jq8nNv3) with Max Supply 500, Unity 1
Fee	100 VSYS
Status	Success
Function	supersede(newIssuer,maker) issue(amount,issuer) destroy(amount,issuer) send(recipient,amount,caller) transfer(sender,recipient,amount) deposit(sender,smart,amount) withdraw(smart,recipient,amount) totalSupply(total) maxSupply(max) balanceOf(address,balance) getIssuer(issuer)

This token does not support split function, there are 11 functions issued. Users can then operate their newly created token with given functions.



Finding Summary

blockchain/contract/

- Contract.scala
  Contract is an object extends `ScorexLogging`. `Contract` defines fields of `trigger`, `descriptor`, `stateVar`, `textual`, `languageCode` and `languageVersion`, and defines functions of `buildContract()`, `fromBytes()`, `fromBase58String()`, `checkStateVar()`, `textualFromBytes()`, `funcFromBytes()`, `paraFromBytes()`, `checkTextual()` and `identifierCheck` to check if the values used for transactions are valid or not.
  - MINOR `val identifierCheck()` hardcoded the `illegalIdf` as string list.
  - ✓ V Systems Would use system contstant list instead of hardcoded string list to check textual string.

- ContractPermitted.scala
  Define constants and state variables that are permitted to contracts, such as `FunId` is the first two bytes of serialization of transaction.
  - INFO We observed that the variables in the object declarations are mostly `public`. If there are no external calls or value checks requirements for these variables. Consider that the visiblity of these variables can be changed to `private`.
  - INFO Too much redundancy as a result of writing a whole new version of everything for `WithoutSplit`. Would be cleaner if converted to a flag during init, which can improve the readability and reduce the cost for future maintenance.

- DataEntry.scala
  Raw data interpretations from bytes with type/validity checking and parsing.
  - INFO Unhandled case in `DataEntry.toJson()`, might be safer to have a default case in the end.
  - INFO Function `parseArraySize()` could be simplified to be cleaner.

- DataType.scala
  Datatypes interpretation from bytes of the seven contract variable types. Functions of `fromByte()`, `check()` and `checkTypes()` to ensure external usage of `DataType` are safe.

- ExecutionContext.scala
  ExecutionContext is a core class with massive data interactions between transactions and contracts, setting the states, during registering contract or executing contract function. This is the enumeration datatype representations for fields such as `PublicKey`, `Address`, `Amount`, `Int32`, `ShortText`, `ContractAccount` and `Account`.

api/http/contract/

- ContractApiRoute.scala
  - INFO Hardcoded default value for `ApiImplicitParam` instantiation

- ContractBroadcastApiRoute.scala
  - INFO Hardcoded default value for `ApiImplicitParam` instantiation
- ExecuteContractFunctionRequest.scala
  - MINOR Hardcoded fee values (as an example), which may fluctuate in the future.
    - ✓ V Systems `fee` would be updated in later version to represent max fee that user are willing to pay.
- RegisterContractRequest.scala
  - MINOR Hardcoded fee values (as an example), which may fluctuate in the future.
    - ✓ V Systems `fee` would be updated in later version to represent max fee that user are

blockchain/transaction/contract/

- ExecuteContractFunctionTransaction.scala
  Create a contract function call after signing it
  - MINOR `feeScale` is a constant, stored in transaction to describe or to manipulate transaction fees. In this case, transactions would make no differences for representing fee that users are willing to pay.
    - ✓ V Systems `fee` would be updated in later version to represent max fee that user are willing to pay.
- RegisterContractTransaction.scala
  Function `create()` is called from the `transactionFactory` to register contract. Then `create()` function call `createWithProof` without sign and with sign to create the transaction.

blockchain/state/opcdiffs/

- AssertOpcDiff.scala
  Handling data validation check.
  - `parseBytes(context: ExecutionContext)(bytes: Array[Byte], data: Seq[DataEntry]): Either[ValidationError, OpcDiff]:`
    - Parse `bytes` and `data` to return `OpcDiff`.
    - INFO Recommend to replace magic numbers in `parseBytes()` with symbolic constants.
- CDBVOpcDiff.scala
  Handling setting of contract database values.
  - `set(context: ExecutionContext)(stateVar: Array[Byte],value: DataEntry): Either[ValidationError, OpcDiff]:`

    * Set `contractDB` (and `relatedAddress` if `value` is type of `DataType`)
  - INFO Recommend to replace magic numbers in `parseBytes()` with symbolic constants.

- CDBVROpcDiff.scala
  Getter functions of contract database values.
  - INFO Recommend to replace magic numbers in `parseBytes()` with symbolic constants.

- LoadOpcDiff.scala
  Define object `LoadType` and functions `signer()`, `caller()` and `parseBytes()`.
  - INFO Recommend to replace magic numbers in `parseBytes()` with symbolic constants.

- OpcDiff.scala
  Object to handle states changes caused by opcodes.
  - MINOR Since `combine()` function helps to add the latest OpcDiff with previous ones. It would be safe since V Systems team check the OpcDiff should be valid. It would be a better practice if there could be two function. The first one only handles the OpcDiff combination, and the second one only handles the states update.
    * ✓ V Systems `combine()` function is a transition between two given opcDiff objects. The function would take two given opcDiff objects `older` and `newer` and return a newly combined opcDiff object as a whole.

- OpcDiffer.scala
  Opcode difference (change caused by opcode) calculator.

- OpcFuncDiffer.scala
  Use in `RegisterContractTransactionDiff` and `ExecuteContractFunctionTransactionDiff` to get the opcDiff and related contract/token Map.

- ReturnOpcDiff.scala
  Define object `ReturnType`, function `value()` and `parseBytes()`.
  - INFO Recommend to replace magic numbers in `parseBytes()` with symbolic constants.

- TDBAOpcDiff.scala
  Token database rollback opcodes including `deposit()`, `withdraw()` and `transfer()`, where `deposit()` and `withdraw()` operation interacts with contract address and the behavior of `transfer()` is like `transferFrom()` in Solidity.
  - INFO Use `addExact()` to handle overflow and underflow, which is safe. However, from the design level, since`val recipientCurrentBalance` is derived from `tokenAccountBalance`, which is of `Long` type, recommend to use unsigned number libraries to ensure logically balance cannot be negative.

  - INFO Recommend to check `issuer` in `deposit()` is not zero address.
  - INFO Recommend to check `issuer` in `withdraw()` is not zero address.
  - INFO Recommend to check `recipient` in `transfer()` is not zero address.
  - INFO Recommend to replace magic numbers in `parseBytes()` with symbolic constants.

- TDBAROpcDiff.scala
  Get `balance` of given address.
  - INFO Recommend to replace magic numbers in `parseBytes()` with symbolic constants.
  - INFO Recommend to check `address` in `balance()` is not zero address.

- TDBOpcDiff.scala
  Token database operations that can create newToken and split by setting new unity.
  - INFO Recommend to replace magic numbers in `parseBytes()` with symbolic constants.

- TDBROpcDiff.scala
  Set `max`, `unity`, `total` and `desc` states.
  - INFO Recommend to replace magic numbers in `parseBytes()` with symbolic constants.

account/

- ContractAccount.scala
  Contract Address functionality with encode/decode from bytes and string to `ContractAccount` , as well as valid address bytes check.
  - INFO Recommend to add validity check on `tokenIdBytes` in `contractIdFromBytes` function
  - INFO hardcoded `AddressVersion` which is set to 6. Used as a comparison to pass/fail version check

blockchain/state/diffs/

- RegisterContractTransactionDiff.scala
  Calculate the difference (delta) of the transaction after contract register.
  - `apply()`
    * INFO Duplicate functionality with `ExecuteContractFunctionTransaction.apply`. Could be better to merge them.
    * INFO Sender address not explicitly converted to `Address` type.

- ExecuteContractFunctionTransactionDiff.scala
  Calculate the difference (delta) of the transaction after contract function execution.
  - `apply()` Get transaction difference info from OpcFuncDiff and puts them together into a Diff object

Best Practice

Design of smart contract development requires a particular engineering mindset. A failure in the initial construction can be catastrophic, and changing the project after the fact can be exceedingly difficult.
To ensure success and to avoid the challenges above design of smart contracts should here to best practices at their conception. Below, we summarized a checklist of key points & vulnerability vectors that help to indicate a high overall quality of the current V Systems project. (✓ indicates satisfaction; × indicates unsatisfaction; – indicates inapplicability)

General

Compiling

- ✓ Correct environment settings, e.g. compiler version, test framework
- × External libraries are up to date.

Logging

- ✓ Specify error cases by defining various classes and objects extends `ValidationError`
- ✓ Use status code to monitor transaction status

Arithmetic Vulnerability

Two's Complement / Integer underflow / overflow

- ✓ Use Math library with addExact() before all arithmetic operations to catch integer overflow and underflow errors

Floating Points and Precision

- ✓ Correct handling the right precision when dealing ratios and rates

Access & Privilege Control Vulnerability

Circuit Breaker

- – Provide pause functionality for control and emergency handling

Restriction

- ✓ Provide proper access control for functions
- ✓ Establish rate limiter for certain operations
- ✓ Restrict access to sensitive functions
- – Restrict permission to contract destruction
- – Establish speed bumps slow down some sensitive actions, any malicious actions occur, there is time to recover.

DoS Vulnerability

A type of attacks that make the contract inoperable with certain period of time or permanently.

Unexpected Revert

- ✓ States would be changed if and only if the diffcodes passed all of the validation checks, so that functions would not be reverted in unexpected situations.

Human Factor Manipulation Vulnerability

Transaction Ordering Or Front-Running

- ✓ Setting a constant value for `fee` to avoid this vulnerability
- ✓ The high rate of block generation can help preventing the users taking the benefit of transaction ordering

External Referencing Vulnerability

External calls may execute malicious code in that contract or any other contract that it depends upon. As such, every external call should be treated as a potential security risk.

Avoid state changes before validatoin checks

- ✓ States would be changed if and only if the diffcodes passed all of the validation checks.

Avoid state changes after external calls

- ✓ Using a checks-effects-interactions pattern to minimize the state changes after external contract or call referencing.

Handle errors in external calls

- ✓ Correct handling errors in any external contract or call referencing by checking its return value

Race Conditions Vulnerability

A type of vulnerability caused by calling external contracts that attacker can take over the control flow, and make changes to the data that the calling function wasn't expecting.

Visibility Vulnerability

The visibility determines whether a function can be called externally by users, by other derived contracts, only internally or only externally.

- ✓ Specify the visibility of all functions in a contract, even if they are intentionally public

Incorrect Interface Vulnerability

A contract interface defines functions with a different type signature than the implementation, causing two different method id's to be created. As a result, when the interface is called, the fallback method will be executed.

- ✓ Ensure the defined function signatures are match with the contract interface and implementation

Bad Randomness Vulnerability

Pseudo random number generation is not supported by Solidity as default, which it is an unsafe operation.

- ✓ Avoid using randomness for block variables, there may be a chance manipulated by the miners

Documentation

The presence of documentation helps keep track of all aspects of an application and it improves on the quality of a software product. Its main focuses are development, maintenance and knowledge transfer to other developers.

- × Provide project README and execution guidance
- × Provide inline comment for complex functions intention
- × Provide instruction to initialize and execute the test files

Testing

Rigorous testing of components and systems, and their associated documentation, can help reduce the risk of failures occurring during operation. When defects are detected, and subsequently fixed, this contributes to the quality of the components or systems.

- ✓ Provide test scripts and coverage for potential scenarios

Overall we found the design of smart contracts based on opcodes to follow good practices. With the final update of source code and delivery of the audit report, we conclude that the design of smart contracts is structurally sound and not vulnerable to any classically known anti-patterns or security issues. The audit report itself is not necessarily a guarantee of correctness or trustworthiness, and we always recommend to seek multiple opinions, keep improving the codebase, and more test coverage and sandbox deployments.

Source Code

ContractAccount.scala

```scala
package vsys.account

import scorex.crypto.encode.Base58
import vsys.blockchain.state.ByteStr
import vsys.blockchain.transaction.ValidationError
import vsys.blockchain.transaction.ValidationError.{InvalidContractAddress,
    InvalidAddress}
import vsys.utils.crypto.hash.SecureCryptographicHash._
import vsys.utils.{base58Length, ScorexLogging}

import scala.util.Success

sealed trait ContractAccount extends Serializable {

  val bytes: ByteStr
  lazy val address: String = bytes.base58
  lazy val stringRepr: String = address

  override def toString: String = stringRepr

  override def equals(obj: Any): Boolean = obj match {
    case conAcc: ContractAccount => bytes == conAcc.bytes
    case _ => false
  }

  override def hashCode(): Int = java.util.Arrays.hashCode(bytes.arr)

}

object ContractAccount extends ScorexLogging {

  val Prefix: String = "contractAccount:"

  val AddressVersion: Byte = 6
  val TokenAddressVersion: Byte = -124
  val TokenIndexLength = 4
  val ChecksumLength = 4
  val HashLength = 20
  val AddressLength = 1 + 1 + ChecksumLength + HashLength
  val AddressStringLength = base58Length(AddressLength)

  private def scheme = AddressScheme.current

  private class ContractAddressImpl(val bytes: ByteStr) extends ContractAccount

  def fromId(id: ByteStr): ContractAccount = {
    val contractAccountHash = hash(id.arr).take(HashLength)
    val withoutChecksum = AddressVersion +: scheme.chainId +: contractAccountHash
    val bytes = withoutChecksum ++ calcCheckSum(withoutChecksum)
    new ContractAddressImpl(ByteStr(bytes))
  }

  def fromBytes(addressBytes: Array[Byte]): Either[ValidationError, ContractAccount] =
      {
```

```
53    if (isByteArrayValid(addressBytes)) Right(new ContractAddressImpl(ByteStr(
          addressBytes)))
54    else Left(InvalidContractAddress)
55  }
56
57  private def fromBase58String(address: String): Either[ValidationError,
        ContractAccount] = {
58    if (address.length > AddressStringLength) Left(InvalidContractAddress)
59    else {
60      Base58.decode(address) match {
61        case Success(byteArray) if isByteArrayValid(byteArray) => Right(new
              ContractAddressImpl(ByteStr(byteArray)))
62        case _ => Left(InvalidContractAddress)
63      }
64    }
65  }
66
67  def fromString(address: String): Either[ValidationError, ContractAccount] = {
68    val base58String = if (address.startsWith(Prefix))
69      address.drop(Prefix.length)
70    else address
71    fromBase58String(base58String)
72  }
73
74  private def isByteArrayValid(addressBytes: Array[Byte]): Boolean = {
75    val version = addressBytes.head
76    val network = addressBytes.tail.head
77    if (version != AddressVersion) {
78      log.warn(s"Unknown contract address version: $version")
79      false
80    } else if (network != scheme.chainId) {
81      log.warn(s"~ Expected network: ${scheme.chainId}(${scheme.chainId.toChar}")
82      log.warn(s"~ Actual network: $network(${network.toChar}")
83      false
84    } else {
85      if (addressBytes.length != ContractAccount.AddressLength)
86        false
87      else {
88        val checkSum = addressBytes.takeRight(ChecksumLength)
89        val checkSumGenerated = calcCheckSum(addressBytes.dropRight(ChecksumLength))
90        checkSum.sameElements(checkSumGenerated)
91      }
92    }
93  }
94
95  private def calcCheckSum(withoutChecksum: Array[Byte]): Array[Byte] = hash(
        withoutChecksum).take(ChecksumLength)
96
97  def tokenIdFromBytes(addressBytes: Array[Byte], idxBytes: Array[Byte]): Either[
        ValidationError, ByteStr] = {
98    if (isByteArrayValid(addressBytes)) {
99      val contractIdNoCheckSum = addressBytes.tail.dropRight(ChecksumLength)
100     val withoutChecksum = Array(TokenAddressVersion) ++ contractIdNoCheckSum ++
            idxBytes
101     val bytes = withoutChecksum ++ calcCheckSum(withoutChecksum)
102     Right(ByteStr(bytes))
103   } else Left(InvalidAddress)
104 }
```

```
105
106   def contractIdFromBytes(tokenIdBytes: Array[Byte]): ByteStr = {
107     val contractIdNoCheckSum = tokenIdBytes.tail.dropRight(ChecksumLength +
          TokenIndexLength)
108     val withoutChecksum = Array(AddressVersion) ++ contractIdNoCheckSum
109     val bytes = withoutChecksum ++ calcCheckSum(withoutChecksum)
110     ByteStr(bytes)
111   }
112
113 }
```

ContractApiRoute.scala

```
1  package vsys.api.http.contract
2
3  import javax.ws.rs.Path
4
5  import akka.http.scaladsl.model.StatusCodes
6  import akka.http.scaladsl.server.Route
7  import com.google.common.primitives.Ints
8  import io.netty.channel.group.ChannelGroup
9  import io.swagger.annotations._
10 import play.api.libs.json.{JsArray, JsNumber, JsObject, Json}
11 import vsys.account.Address
12 import vsys.account.ContractAccount.{contractIdFromBytes, tokenIdFromBytes}
13 import vsys.api.http._
14 import vsys.blockchain.state.ByteStr
15 import vsys.blockchain.state.reader.StateReader
16 import vsys.blockchain.transaction._
17 import vsys.blockchain.UtxPool
18 import vsys.blockchain.contract.ContractPermitted
19 import vsys.settings.RestAPISettings
20 import vsys.utils.serialization.Deser
21 import vsys.utils.Time
22 import vsys.wallet.Wallet
23
24 import scala.util.Success
25 import scala.util.control.Exception
26
27 @Path("/contract")
28 @Api(value = "/contract")
29 case class ContractApiRoute (settings: RestAPISettings, wallet: Wallet, utx: UtxPool,
      allChannels: ChannelGroup, time: Time, state: StateReader)
30   extends ApiRoute with BroadcastRoute {
31
32   override val route = pathPrefix("contract") {
33     register ~ content ~ info ~ tokenInfo ~ balance ~ execute ~ tokenId
34   }
35
36   @Path("/register")
37   @ApiOperation(value = "Register a contract",
38     httpMethod = "POST",
39     produces = "application/json",
40     consumes = "application/json")
41   @ApiImplicitParams(Array(
42     new ApiImplicitParam(
43       name = "body",
44       value = "Json with data",
45       required = true,
```

```
46      paramType = "body",
47      dataType = "vsys.api.http.contract.RegisterContractRequest",
48      defaultValue = "{\n\t\"sender\": \"3Mx2afTZ2KbRrLNbytyzTtXukZvqEB8SkW7\",\n\t\"
            contract\": \"contract\",\n\t\"data\":\"data\",\n\t\"description\":\"5
            VECG3ZHwy\",\n\t\"fee\": 100000,\n\t\"feeScale\": 100\n}"
49    )
50  ))
51  @ApiResponses(Array(new ApiResponse(code = 200, message = "Json with response or
        error")))
52  def register: Route = processRequest("register", (t: RegisterContractRequest) =>
        doBroadcast(TransactionFactory.registerContract(t, wallet, time)))
53
54
55  @Path("/content/{contractId}")
56  @ApiOperation(value = "Contract content", notes = "Get contract content associated
        with a contract id.", httpMethod = "GET")
57  @ApiImplicitParams(Array(
58    new ApiImplicitParam(name = "contractId", value = "Contract ID", required = true,
          dataType = "string", paramType = "path")
59  ))
60  def content: Route = (get & path("content" / Segment)) { encoded =>
61    ByteStr.decodeBase58(encoded) match {
62      case Success(id) => state.contractContent(id) match {
63        case Some((h, txId, ct)) => complete(Json.obj("transactionId" -> txId.base58)
              ++ ct.json ++ Json.obj("height" -> JsNumber(h)))
64        case None => complete(ContractNotExists)
65      }
66      case _ => complete(InvalidAddress)
67    }
68  }
69
70  @Path("/info/{contractId}")
71  @ApiOperation(value = "Info", notes = "Get contract info associated with a contract
        id.", httpMethod = "GET")
72  @ApiImplicitParams(Array(
73    new ApiImplicitParam(name = "contractId", value = "Contract ID", required = true,
          dataType = "string", paramType = "path")
74  ))
75  def info: Route = (get & path("info" / Segment)) { contractId =>
76    complete(infoJson(contractId))
77  }
78
79  private def infoJson(contractIdStr: String): Either[ApiError, JsObject] = {
80    ByteStr.decodeBase58(contractIdStr) match {
81      case Success(id) => state.contractContent(id) match {
82        case Some((h, txId, ct)) => Right(Json.obj(
83          "contractId" -> contractIdStr,
84          "transactionId" -> txId.base58,
85          "type" -> typeFromBytes(ct.bytes.arr),
86          "info" -> JsArray((ct.stateVar, paraFromBytes(ct.textual.last)).zipped.map {
                (a, b) =>
87            (state.contractInfo(ByteStr(id.arr ++ Array(a(0)))), b) }.filter(_._1.
                isDefined).map { a => a._1.get.json ++ Json.obj("name" -> a._2) }),
88          "height" -> JsNumber(h))
89        )
90        case None => Left(ContractNotExists)
91      }
92      case _ => Left(InvalidAddress)
```

```
93      }
94    }
95
96    private def typeFromBytes(bytes: Array[Byte]): String = {
97      if (bytes sameElements ContractPermitted.contract.bytes.arr) {
98        "TokenContractWithSplit"
99      } else if (bytes sameElements ContractPermitted.contractWithoutSplit.bytes.arr) {
100       "TokenContract"
101     } else {
102       "GeneralContract"
103     }
104   }
105
106   @Path("/tokenInfo/{tokenId}")
107   @ApiOperation(value = "Token's Info", notes = "Token's info by given token",
          httpMethod = "Get")
108   @ApiImplicitParams(Array(
109     new ApiImplicitParam(name = "tokenId", value = "Token ID", required = true,
            dataType = "string", paramType = "path")
110   ))
111   def tokenInfo: Route = (get & path("tokenInfo" / Segment)) { tokenId =>
112     ByteStr.decodeBase58(tokenId) match {
113       case Success(id) => {
114         val maxKey = ByteStr(id.arr ++ Array(0.toByte))
115         val totalKey = ByteStr(id.arr ++ Array(1.toByte))
116         val unityKey = ByteStr(id.arr ++ Array(2.toByte))
117         val descKey = ByteStr(id.arr ++ Array(3.toByte))
118         state.tokenInfo(maxKey) match {
119           case Some(x) => complete(Json.obj("tokenId" -> tokenId,
120             "contractId" -> contractIdFromBytes(id.arr),
121             "max" -> x.json.value("data"),
122             "total" -> state.tokenAccountBalance(totalKey),
123             "unity" -> state.tokenInfo(unityKey).get.json.value("data"),
124             "description" -> state.tokenInfo(descKey).get.json.value("data")
125           ))
126           case _ => complete(TokenNotExists)
127         }
128       }
129       case _ => complete(InvalidAddress)
130     }
131   }
132
133   private def paraFromBytes(bytes: Array[Byte]): Seq[String] = {
134     val listParaNameBytes = Deser.parseArrays(bytes)
135     listParaNameBytes.foldLeft(Seq.empty[String]) { case (e, b) => {
136       val paraName = Deser.deserilizeString(b)
137       e :+ paraName
138     }
139     }
140   }
141
142   @Path("balance/{address}/{tokenId}")
143   @ApiOperation(value = "Token's balance", notes = "Account's balance by given token",
          httpMethod = "Get")
144   @ApiImplicitParams(Array(
145     new ApiImplicitParam(name = "address", value = "Address", required = true,
            dataType = "string", paramType = "path"),
146     new ApiImplicitParam(name = "tokenId", value = "Token ID", required = true,
```

```
      dataType = "string", paramType = "path")
147 ))
148 def balance: Route = (get & path("balance" / Segment / Segment)) { (address, tokenId
    ) =>
149   complete(balanceJson(address, tokenId))
150 }
151
152 private def balanceJson(address: String, tokenIdStr: String): Either[ApiError,
    JsObject] = {
153   ByteStr.decodeBase58(tokenIdStr) match {
154     case Success(tokenId) =>
155       val unityKey = ByteStr(tokenId.arr ++ Array(2.toByte))
156       state.tokenInfo(unityKey) match {
157         case Some(x) => (for {
158           acc <- Address.fromString(address)
159         } yield Json.obj(
160           "address" -> acc.address,
161           "tokenId" -> tokenIdStr,
162           "balance" -> state.tokenAccountBalance(ByteStr(tokenId.arr ++ acc.bytes.arr
                )),
163           "unity" -> x.json.value("data"))
164           ).left.map(ApiError.fromValidationError)
165         case _ => Left(TokenNotExists)
166       }
167     case _ => Left(InvalidAddress)
168   }
169 }
170
171 @Path("/execute")
172 @ApiOperation(value = "Execute a contract function",
173   httpMethod = "POST",
174   produces = "application/json",
175   consumes = "application/json")
176 @ApiImplicitParams(Array(
177   new ApiImplicitParam(
178     name = "body",
179     value = "Json with data",
180     required = true,
181     paramType = "body",
182     dataType = "vsys.api.http.contract.ExecuteContractFunctionRequest",
183     defaultValue = "{\n\t\"sender\": \"3Mx2afTZ2KbRrLNbytyzTtXukZvqEB8SkW7\",\n\t\"
        contractId\": \"contractId\",\n\t\"funcIdx\": \"0\",\n\t\"data\":\"data\",\n\
        t\"description\":\"5VECG3ZHwy\",\n\t\"fee\": 100000,\n\t\"feeScale\": 100\n}"
184   )
185 ))
186 @ApiResponses(Array(new ApiResponse(code = 200, message = "Json with response or
    error")))
187 def execute: Route = processRequest("execute", (t: ExecuteContractFunctionRequest)
    => doBroadcast(TransactionFactory.executeContractFunction(t, wallet, time)))
188
189 @Path("contractId/{contractId}/tokenIndex/{tokenIndex}")
190 @ApiOperation(value = "Token's Id", notes = "Token Id from contract Id and token
    index", httpMethod = "Get")
191 @ApiImplicitParams(Array(
192   new ApiImplicitParam(name = "contractId", value = "Contract ID", required = true,
      dataType = "string", paramType = "path"),
193   new ApiImplicitParam(name = "tokenIndex", value = "Token Index", required = true,
      dataType = "integer", paramType = "path")
```

```
  ))
  def tokenId: Route = (pathPrefix("contractId") & get) {
    pathPrefix(Segment) { contractIdStr =>
      ByteStr.decodeBase58(contractIdStr) match {
        case Success(c) =>
          pathPrefix("tokenIndex") {
            pathEndOrSingleSlash {
              complete(InvalidTokenIndex)
            } ~
              path(Segment) { tokenIndexStr =>
                Exception.allCatch.opt(tokenIndexStr.toInt) match {
                  case Some(tokenIndex) if tokenIndex >= 0 =>
                    tokenIdFromBytes(c.arr, Ints.toByteArray(tokenIndex)) match {
                      case Right(b) => complete(Json.obj("tokenId" -> b))
                      case Left(e) => complete(e)
                    }
                  case _ =>
                    complete(InvalidTokenIndex)
                }
              }
          } ~ complete(StatusCodes.NotFound)
        case _ => complete(InvalidAddress)
      }
    }
  }
}
```

ContractBroadcastApiRoute.scala

```
package vsys.api.http.contract

import javax.ws.rs.Path

import akka.http.scaladsl.server.Route
import io.netty.channel.group.ChannelGroup
import io.swagger.annotations._
import vsys.api.http._
import vsys.blockchain.UtxPool
import vsys.settings.RestAPISettings


@Path("/contract/broadcast")
@Api(value = "/contract")
case class ContractBroadcastApiRoute(settings: RestAPISettings,
                                     utx: UtxPool,
                                     allChannels: ChannelGroup) extends ApiRoute with
                                         BroadcastRoute {
  override val route = pathPrefix("contract" / "broadcast") {
    signedRegister ~ signedExecute
  }

  @Path("/register")
  @ApiOperation(value = "Broadcasts a signed register contract transaction",
    httpMethod = "POST",
    produces = "application/json",
    consumes = "application/json")
  @ApiImplicitParams(Array(
    new ApiImplicitParam(
      name = "body",
```

```
      value = "Json with data",
      required = true,
      paramType = "body",
      dataType = "vsys.api.http.contract.SignedRegisterContractRequest",
      defaultValue = "{\n\t\"contract\": \"contract\",\n\t\"data\":\"data\",\n\t\"description\":\"5VECG3ZHwy\",\n\t\"senderPublicKey\": \"11111\",\n\t\"fee\": 100000,\n\t\"feeScale\": 100,\"timestamp\": 12345678,\n\t\"signature\": \"asdasdasd\"\n}"
    )
  ))
  @ApiResponses(Array(new ApiResponse(code = 200, message = "Json with response or error")))
  def signedRegister: Route = (path("register") & post) {
    json[SignedRegisterContractRequest] { contractReq =>
      doBroadcast(contractReq.toTx)
    }
  }

  @Path("/execute")
  @ApiOperation(value = "Broadcasts a signed execute contract function transaction",
    httpMethod = "POST",
    produces = "application/json",
    consumes = "application/json")
  @ApiImplicitParams(Array(
    new ApiImplicitParam(
      name = "body",
      value = "Json with data",
      required = true,
      paramType = "body",
      dataType = "vsys.api.http.contract.SignedExecuteContractFunctionRequest",
      defaultValue = "{\n\t\"contractId\": \"contractId\",\n\t\"funcIdx\": \"0\",\n\t\"data\":\"data\",\n\t\"description\":\"5VECG3ZHwy\",\n\t\"senderPublicKey\": \"11111\",\n\t\"fee\": 100000,\n\t\"feeScale\": 100,\"timestamp\": 12345678,\n\t\"signature\": \"asdasdasd\"\n}"
    )
  ))
  @ApiResponses(Array(new ApiResponse(code = 200, message = "Json with response or error")))
  def signedExecute: Route = (path("execute") & post) {
    json[SignedExecuteContractFunctionRequest] { contractReq =>
      doBroadcast(contractReq.toTx)
    }
  }

}
```

ExecuteContractFunctionRequest.scala

```
package vsys.api.http.contract

import io.swagger.annotations.ApiModelProperty
import play.api.libs.json.{Format, Json}


case class ExecuteContractFunctionRequest(@ApiModelProperty(value = "Base58 encoded sender address", required = true)
                                          sender: String,
                                          @ApiModelProperty(value = "Base58 encoded contract id", required = true)
```

```
                                    contractId: String,
                                    @ApiModelProperty(required = true)
                                    functionIndex: Short,
                                    @ApiModelProperty(value = "Base58 encoded
                                        function data", required = true)
                                    functionData: String,
                                    @ApiModelProperty(value = "Base58 encoded
                                        attachment of contract")
                                    attachment: Option[String],
                                    @ApiModelProperty(required = true, example = "
                                        30000000")
                                    fee: Long,
                                    @ApiModelProperty(required = true, example = "
                                        100")
                                    feeScale: Short)

object ExecuteContractFunctionRequest {
  implicit val executeContractFunctionRequestFormat: Format[
      ExecuteContractFunctionRequest] = Json.format
}
```

RegisterContractRequest.scala

```
package vsys.api.http.contract

import io.swagger.annotations.ApiModelProperty
import play.api.libs.json.{Format, Json}


case class RegisterContractRequest(@ApiModelProperty(value = "Base58 encoded sender
    address", required = true)
                                   sender: String,
                                   @ApiModelProperty(value = "Base58 encoded contract",
                                       required = true)
                                   contract: String,
                                   @ApiModelProperty(value = "Base58 encoded init data",
                                       required = true)
                                   initData: String,
                                   @ApiModelProperty(value = "Description of contract")
                                   description: Option[String],
                                   @ApiModelProperty(required = true, example = "
                                       10000000000")
                                   fee: Long,
                                   @ApiModelProperty(required = true, example = "100")
                                   feeScale: Short)

object RegisterContractRequest {
  implicit val registerContractRequestFormat: Format[RegisterContractRequest] = Json.
      format
}
```

SignedExecuteContractFunctionRequest.scala

```
package vsys.api.http.contract

import io.swagger.annotations.ApiModelProperty
import play.api.libs.json.{Format, Json}
import vsys.account.{ContractAccount, PublicKeyAccount}
import vsys.api.http.BroadcastRequest
import vsys.blockchain.transaction.ValidationError
```

```
8  import vsys.blockchain.transaction.TransactionParser.SignatureStringLength
9  import vsys.blockchain.contract.DataEntry
10 import vsys.blockchain.transaction.contract.ExecuteContractFunctionTransaction
11 import scorex.crypto.encode.Base58
12
13
14 case class SignedExecuteContractFunctionRequest(@ApiModelProperty(value = "Base58
       encoded sender public key", required = true)
15                                                  senderPublicKey: String,
16                                                  @ApiModelProperty(value = "Base58 encoded
                                                        contract id", required = true)
17                                                  contractId: String,
18                                                  @ApiModelProperty(required = true)
19                                                  functionIndex: Short,
20                                                  @ApiModelProperty(value = "Base58 encoded
                                                        function data", required = true)
21                                                  functionData: String,
22                                                  @ApiModelProperty(value = "Base58 encoded
                                                        attachment")
23                                                  attachment: Option[String],
24                                                  @ApiModelProperty(required = true)
25                                                  fee: Long,
26                                                  @ApiModelProperty(required = true)
27                                                  feeScale: Short,
28                                                  @ApiModelProperty(required = true)
29                                                  timestamp: Long,
30                                                  @ApiModelProperty(required = true)
31                                                  signature: String) extends BroadcastRequest
                                                        {
32   def toTx: Either[ValidationError, ExecuteContractFunctionTransaction] = for {
33     _sender <- PublicKeyAccount.fromBase58String(senderPublicKey)
34     _signature <- parseBase58(signature, "invalid.signature", SignatureStringLength)
35     _contractId <- ContractAccount.fromString(contractId)
36     _functionData <- DataEntry.fromBase58String(functionData)
37     _attachment = attachment.filter(_.nonEmpty).map(Base58.decode(_).get).getOrElse(
           Array.emptyByteArray)
38     _t <- ExecuteContractFunctionTransaction.create(_sender, _contractId,
           functionIndex, _functionData, _attachment, fee, feeScale, timestamp,
           _signature)
39   } yield _t
40 }
41
42 object SignedExecuteContractFunctionRequest {
43   implicit val broadcastExecuteContractFunctionRequestReadsFormat: Format[
       SignedExecuteContractFunctionRequest] = Json.format
44 }
```

SignedRegisterContractRequest.scala

```
1  package vsys.api.http.contract
2
3  import io.swagger.annotations.ApiModelProperty
4  import play.api.libs.json.{Format, Json}
5  import vsys.account.PublicKeyAccount
6  import vsys.api.http.BroadcastRequest
7  import vsys.utils.serialization.Deser
8  import vsys.blockchain.contract.{Contract, DataEntry}
9  import vsys.blockchain.transaction.TransactionParser.SignatureStringLength
10 import vsys.blockchain.transaction.ValidationError
```

```
11 import vsys.blockchain.transaction.contract.RegisterContractTransaction
12
13
14 case class SignedRegisterContractRequest(@ApiModelProperty(value = "Base58 encoded
       sender public key", required = true)
15                                         senderPublicKey: String,
16                                         @ApiModelProperty(value = "Base58 encoded
                                               contract", required = true)
17                                         contract: String,
18                                         @ApiModelProperty(value = "Base58 encoded init
                                               data", required = true)
19                                         initData: String,
20                                         @ApiModelProperty(value = "Description of
                                               contract")
21                                         description: Option[String],
22                                         @ApiModelProperty(required = true)
23                                         fee: Long,
24                                         @ApiModelProperty(required = true)
25                                         feeScale: Short,
26                                         @ApiModelProperty(required = true)
27                                         timestamp: Long,
28                                         @ApiModelProperty(required = true)
29                                         signature: String) extends BroadcastRequest {
30   def toTx: Either[ValidationError, RegisterContractTransaction] = for {
31     _sender <- PublicKeyAccount.fromBase58String(senderPublicKey)
32     _signature <- parseBase58(signature, "invalid.signature", SignatureStringLength)
33     _contract <- Contract.fromBase58String(contract)
34     _initData <- DataEntry.fromBase58String(initData)
35     _description = description.filter(_.nonEmpty).getOrElse(Deser.deserilizeString(
           Array.emptyByteArray))
36     _t <- RegisterContractTransaction.create(_sender, _contract, _initData,
           _description, fee, feeScale, timestamp, _signature)
37   } yield _t
38 }
39
40 object SignedRegisterContractRequest {
41   implicit val broadcastRegisterContractRequestReadsFormat: Format[
         SignedRegisterContractRequest] = Json.format
42 }
```

Contract.scala

```
1  package vsys.blockchain.contract
2
3  import com.google.common.primitives.Ints
4  import play.api.libs.json.{JsObject, Json}
5  import scorex.crypto.encode.Base58
6  import vsys.blockchain.state.ByteStr
7  import vsys.blockchain.transaction.ValidationError
8  import vsys.blockchain.transaction.ValidationError.InvalidContract
9  import vsys.utils.ScorexLogging
10 import vsys.utils.base58Length
11 import vsys.utils.serialization.Deser
12
13 import scala.util.{Success, Try}
14
15 sealed trait Contract {
16
17   lazy val stringRepr: String = Contract.Prefix + Base58.encode(languageCode) + ":" +
```

```
      Base58.encode(languageVersion)
  lazy val bytes: ByteStr = ByteStr(languageCode ++ languageVersion
    ++ Deser.serializeArray(Deser.serializeArrays(trigger))
    ++ Deser.serializeArray(Deser.serializeArrays(descriptor))
    ++ Deser.serializeArray(Deser.serializeArrays(stateVar))
    ++ Deser.serializeArrays(textual))

  val trigger: Seq[Array[Byte]]
  val descriptor: Seq[Array[Byte]]
  val stateVar: Seq[Array[Byte]]
  val textual: Seq[Array[Byte]]
  val languageCode: Array[Byte]
  val languageVersion: Array[Byte]

  lazy val json: JsObject = Json.obj(
    "languageCode" -> Deser.deserilizeString(languageCode),
    "languageVersion" -> Ints.fromByteArray(languageVersion),
    "triggers" -> trigger.map(p => Base58.encode(p)),
    "descriptors" -> descriptor.map(p => Base58.encode(p)),
    "stateVariables" -> stateVar.map(p => Base58.encode(p)),
    "textual" -> Json.obj("triggers" -> Base58.encode(textual.head),
      "descriptors" -> Base58.encode(textual(1)),
      "stateVariables" -> Base58.encode(textual.last))
  )
}

object Contract extends ScorexLogging {

  val Prefix: String = "contract:"

  val MinContractByteSize = 8
  val MinContractStringSize: Int = base58Length(MinContractByteSize)
  val LanguageCodeByteLength = 4
  val LanguageVersionByteLength = 4
  val LanguageCodeByte: Array[Byte] = Deser.serilizeString("vdds")
  val LanguageVersionByte: Array[Byte] = Ints.toByteArray(1)

  def buildContract(languageCode: Array[Byte], languageVersion: Array[Byte],
                    trigger: Seq[Array[Byte]], descriptor: Seq[Array[Byte]],
                    stateVar: Seq[Array[Byte]], textual: Seq[Array[Byte]]): Either[
                        ValidationError, Contract] = {
    case class ContractImpl(languageCode: Array[Byte], languageVersion: Array[Byte],
                            trigger: Seq[Array[Byte]], descriptor: Seq[Array[Byte]],
                            stateVar: Seq[Array[Byte]], textual: Seq[Array[Byte]])
                                extends Contract
    Right(ContractImpl(languageCode, languageVersion, trigger, descriptor, stateVar,
        textual))
  }

  def fromBytes(bytes: Array[Byte]): Either[ValidationError, Contract] = {
    val contract = Try {
      val languageCode = bytes.slice(0, LanguageCodeByteLength)
      val languageVersion = bytes.slice(LanguageCodeByteLength, LanguageCodeByteLength
          + LanguageVersionByteLength)
      val (triggerBytes, triggerEnd) = Deser.parseArraySize(bytes,
          LanguageCodeByteLength + LanguageVersionByteLength)
      val trigger = Deser.parseArrays(triggerBytes)
      val (descriptorBytes, descriptorEnd) = Deser.parseArraySize(bytes, triggerEnd)
```

```
70      val descriptor = Deser.parseArrays(descriptorBytes)
71      val (stateVarBytes, stateVarEnd) = Deser.parseArraySize(bytes, descriptorEnd)
72      val stateVar = Deser.parseArrays(stateVarBytes)
73      val textual = Deser.parseArrays(bytes.slice(stateVarEnd, bytes.length))
74      if (isByteArrayValid(bytes, textual)){
75        buildContract(languageCode, languageVersion, trigger, descriptor, stateVar,
              textual)
76      } else {
77        Left(InvalidContract)
78      }
79    }
80    contract.getOrElse(Left(InvalidContract))
81  }
82
83  def fromBase58String(base58String: String): Either[ValidationError, Contract] = {
84    if (base58String.length < MinContractStringSize) Left(InvalidContract)
85    else {
86      Base58.decode(base58String) match {
87        case Success(byteArray) => fromBytes(byteArray)
88        case _ => Left(InvalidContract)
89      }
90    }
91  }
92
93  def checkStateVar(stateVar: Array[Byte], dataType: DataType.Value): Boolean =
94    stateVar.length == 2 && dataType == DataType(stateVar(1))
95
96  private def isByteArrayValid(bytes: Array[Byte], textual: Seq[Array[Byte]]): Boolean
       = {
97    val textualStr = textualFromBytes(textual)
98    if (!(bytes sameElements ContractPermitted.contract.bytes.arr) &&
99      !(bytes sameElements ContractPermitted.contractWithoutSplit.bytes.arr)) {
100     log.warn(s"Illegal contract ${bytes.mkString(" ")}")
101     false
102   } else if (textualStr.isFailure ||
103     !checkTextual(textualStr.getOrElse((Seq.empty[Seq[String]], Seq.empty[Seq[String
          ]], Seq.empty[String])))) {
104     log.warn(s"Illegal textual ${textual.mkString(" ")}")
105     false
106   } else true
107 }
108
109 private def textualFromBytes(bs: Seq[Array[Byte]]): Try[(Seq[Seq[String]], Seq[Seq[
      String]], Seq[String])] = Try {
110   val triggerFuncBytes = Deser.parseArrays(bs.head)
111   val triggerFunc = funcFromBytes(triggerFuncBytes)
112   val descriptorFuncBytes = Deser.parseArrays(bs(1))
113   val descriptorFunc = funcFromBytes(descriptorFuncBytes)
114   val stateVar = paraFromBytes(bs.last)
115   (triggerFunc, descriptorFunc, stateVar)
116 }
117
118 private def funcFromBytes(bs: Seq[Array[Byte]]): Seq[Seq[String]] = {
119   bs.foldLeft(Seq.empty[Seq[String]]) { case (e, b) => {
120     val (funcNameBytes, funcNameEnd) = Deser.parseArraySize(b, 0)
121     val funcName = Deser.deserilizeString(funcNameBytes)
122     val (listReturnNameBytes, listReturnNameEnd) = Deser.parseArraySize(b,
          funcNameEnd)
```

```
123       val listReturnNames = paraFromBytes(listReturnNameBytes)
124       val listParaNameBytes = b.slice(listReturnNameEnd, b.length)
125       val listParaNames = paraFromBytes(listParaNameBytes)
126       e :+ (listReturnNames ++ Seq(funcName) ++ listParaNames)
127     }
128     }
129   }
130
131   private def paraFromBytes(bytes: Array[Byte]): Seq[String] = {
132     val listParaNameBytes = Deser.parseArrays(bytes)
133     listParaNameBytes.foldLeft(Seq.empty[String]) { case (e, b) => {
134       val paraName = Deser.deserilizeString(b)
135       e :+ paraName
136     }
137     }
138   }
139
140   private def checkTextual(textual: (Seq[Seq[String]], Seq[Seq[String]], Seq[String]))
          : Boolean = {
141     textual._1.flatten.forall(x => identifierCheck(x)) && textual._2.flatten.forall(x
          => identifierCheck(x)) &&
142       textual._3.forall(x => identifierCheck(x))
143   }
144
145   private def identifierCheck(str: String): Boolean = {
146     def checkChar(c: Char): Boolean = {
147       (c == '_') || (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c
            <= '9')
148     }
149
150     val illegalIdf: List[String] = List("register", "unit", "while", "if", "for", "
          return", "match", "context",
151       "contract", "int", "long", "short", "boolean", "trait", "lazy", "class", "else",
            "true", "false", "private",
152       "val", "var", "try", "catch", "throw", "define", "transaction", "else", "public"
          , "jump", "trigger", "then")
153     if ((str.head == '_') || (str.head >= 'a' && str.head <= 'z') || (str.head >= 'A'
          && str.head <= 'Z')) {
154       str.tail.forall(x => checkChar(x)) && !illegalIdf.contains(str.toLowerCase)
155     } else {
156       false
157     }
158   }
159 }
```

ContractPermitted.scala

```
1  package vsys.blockchain.contract
2
3  import com.google.common.primitives.{Bytes, Ints, Shorts}
4  import vsys.blockchain.state.opcdiffs._
5  import vsys.utils.serialization.Deser
6
7  object ContractPermitted {
8    lazy val contract: Contract = Contract.buildContract(Deser.serilizeString("vdds"),
         Ints.toByteArray(1), Seq(initFunc),
9      Seq(supersedeFunc, issueFunc, destroyFunc, splitFunc, sendFunc, transferFunc,
         depositFunc, withdrawFunc,
10       totalSupplyFunc, maxSupplyFunc, balanceOfFunc, getIssuerFunc),
```

```
11    Seq(Array(StateVar.issuer, DataType.Address.id.toByte), Array(StateVar.maker,
          DataType.Address.id.toByte)),
12    Seq(triggerTextual, descriptorTextual, stateVarTextual)
13  ).right.get
14
15  lazy val contractWithoutSplit: Contract = Contract.buildContract(Deser.
        serilizeString("vdds"), Ints.toByteArray(1), Seq(initFunc),
16    Seq(supersedeFuncWithoutSplit, issueFuncWithoutSplit, destroyFuncWithoutSplit,
          sendFuncWithoutSplit,
17      transferFuncWithoutSplit, depositFuncWithoutSplit, withdrawFuncWithoutSplit,
            totalSupplyFuncWithoutSplit,
18      maxSupplyFuncWithoutSplit, balanceOfFuncWithoutSplit, getIssuerFuncWithoutSplit)
            ,
19    Seq(Array(StateVar.issuer, DataType.Address.id.toByte), Array(StateVar.maker,
          DataType.Address.id.toByte)),
20    Seq(triggerTextual, descriptorTextualWithoutSplit, stateVarTextual)
21  ).right.get
22
23  object FunId {
24    val init: Short = 0
25    val supersede: Short = 0
26    val issue: Short = 1
27    val destroy: Short = 2
28    val split: Short = 3
29    val send: Short = 4
30    val transfer: Short = 5
31    val deposit: Short = 6
32    val withdraw: Short = 7
33    val totalSupply: Short = 8
34    val maxSupply: Short = 9
35    val balanceOf: Short = 10
36    val getIssuer: Short = 11
37  }
38
39  object FunIdWithoutSplit {
40    val init: Short = 0
41    val supersede: Short = 0
42    val issue: Short = 1
43    val destroy: Short = 2
44    val send: Short = 3
45    val transfer: Short = 4
46    val deposit: Short = 5
47    val withdraw: Short = 6
48    val totalSupply: Short = 7
49    val maxSupply: Short = 8
50    val balanceOf: Short = 9
51    val getIssuer: Short = 10
52  }
53
54  object ProtoType {
55    val initParaType: Array[Byte] = Array(DataType.Amount.id.toByte, DataType.Amount.
          id.toByte, DataType.ShortText.id.toByte)
56    val supersedeParaType: Array[Byte] = Array(DataType.Account.id.toByte)
57    val issueParaType: Array[Byte] = Array(DataType.Amount.id.toByte)
58    val destroyParaType: Array[Byte] = Array(DataType.Amount.id.toByte)
59    val splitParaType: Array[Byte] = Array(DataType.Amount.id.toByte)
60    val sendParaType: Array[Byte] = Array(DataType.Account.id.toByte, DataType.Amount.
          id.toByte)
```

```
61      val transferParaType: Array[Byte] = Array(DataType.Account.id.toByte, DataType.
            Account.id.toByte, DataType.Amount.id.toByte)
62      val depositParaType: Array[Byte] = Array(DataType.Account.id.toByte, DataType.
            ContractAccount.id.toByte, DataType.Amount.id.toByte)
63      val withdrawParaType: Array[Byte] = Array(DataType.ContractAccount.id.toByte,
            DataType.Account.id.toByte, DataType.Amount.id.toByte)
64      val totalSupplyParaType: Array[Byte] = Array()
65      val maxSupplyParaType: Array[Byte] = Array()
66      val balanceOfParaType: Array[Byte] = Array(DataType.Account.id.toByte)
67      val getIssuerParaType: Array[Byte] = Array()
68    }
69
70    def listOpc(ids: List[Array[Byte]], indexInput: List[Array[Byte]]): Array[Byte] = {
71      val length = Shorts.toByteArray((ids.zip(indexInput).map(x => ((x._1 ++ x._2).
            length + 2).toShort).sum + 2).toShort)
72      val numOpc = Shorts.toByteArray(ids.length.toShort)
73      val listOpc = ids.zip(indexInput).map(x => Shorts.toByteArray((x._1 ++ x._2).
            length.toShort) ++ x._1 ++ x._2).toArray.flatten
74      Bytes.concat(length, numOpc, listOpc)
75    }
76
77    object OpcId {
78      val opcAssertGteqZero: Array[Byte] = Array(OpcDiffer.OpcType.AssertOpc.id.toByte,
            AssertOpcDiff.AssertType.GteqZeroAssert.id.toByte)
79      val opcAssertLteq: Array[Byte] = Array(OpcDiffer.OpcType.AssertOpc.id.toByte,
            AssertOpcDiff.AssertType.LteqAssert.id.toByte)
80      val opcAssertLtInt64: Array[Byte] = Array(OpcDiffer.OpcType.AssertOpc.id.toByte,
            AssertOpcDiff.AssertType.LtInt64Assert.id.toByte)
81      val opcAssertGtZero: Array[Byte] = Array(OpcDiffer.OpcType.AssertOpc.id.toByte,
            AssertOpcDiff.AssertType.GtZeroAssert.id.toByte)
82      val opcAssertEq: Array[Byte] = Array(OpcDiffer.OpcType.AssertOpc.id.toByte,
            AssertOpcDiff.AssertType.EqAssert.id.toByte)
83      val opcAssertIsCallerOrigin: Array[Byte] = Array(OpcDiffer.OpcType.AssertOpc.id.
            toByte, AssertOpcDiff.AssertType.IsCallerOriginAssert.id.toByte)
84      val opcAssertIsSignerOrigin: Array[Byte] = Array(OpcDiffer.OpcType.AssertOpc.id.
            toByte, AssertOpcDiff.AssertType.IsSignerOriginAssert.id.toByte)
85
86      val opcLoadSigner: Array[Byte] = Array(OpcDiffer.OpcType.LoadOpc.id.toByte,
            LoadOpcDiff.LoadType.SignerLoad.id.toByte)
87      val opcLoadCaller: Array[Byte] = Array(OpcDiffer.OpcType.LoadOpc.id.toByte,
            LoadOpcDiff.LoadType.CallerLoad.id.toByte)
88
89      val opcCDBVSet: Array[Byte] = Array(OpcDiffer.OpcType.CDBVOpc.id.toByte,
            CDBVOpcDiff.CDBVType.SetCDBV.id.toByte)
90
91      val opcCDBVRGet: Array[Byte] = Array(OpcDiffer.OpcType.CDBVROpc.id.toByte,
            CDBVROpcDiff.CDBVRType.GetCDBVR.id.toByte)
92
93      val opcTDBNewToken: Array[Byte] = Array(OpcDiffer.OpcType.TDBOpc.id.toByte,
            TDBOpcDiff.TDBType.NewTokenTDB.id.toByte)
94      val opcTDBSplit: Array[Byte] = Array(OpcDiffer.OpcType.TDBOpc.id.toByte,
            TDBOpcDiff.TDBType.SplitTDB.id.toByte)
95
96      val opcTDBROpcMax: Array[Byte] = Array(OpcDiffer.OpcType.TDBROpc.id.toByte,
            TDBROpcDiff.TDBRType.MaxTDBR.id.toByte)
97      val opcTDBROpcTotal: Array[Byte] = Array(OpcDiffer.OpcType.TDBROpc.id.toByte,
            TDBROpcDiff.TDBRType.TotalTDBR.id.toByte)
98
```

```
99      val opcTDBADeposit: Array[Byte] = Array(OpcDiffer.OpcType.TDBAOpc.id.toByte,
            TDBAOpcDiff.TDBAType.DepositTDBA.id.toByte)
100     val opcTDBAWithdraw: Array[Byte] = Array(OpcDiffer.OpcType.TDBAOpc.id.toByte,
            TDBAOpcDiff.TDBAType.WithdrawTDBA.id.toByte)
101     val opcTDBATransfer: Array[Byte] = Array(OpcDiffer.OpcType.TDBAOpc.id.toByte,
            TDBAOpcDiff.TDBAType.TransferTDBA.id.toByte)
102
103     val opcTDBARBalance: Array[Byte] = Array(OpcDiffer.OpcType.TDBAROpc.id.toByte,
            TDBAROpcDiff.TDBARType.BalanceTBDAR.id.toByte)
104
105     val opcReturnValue: Array[Byte] = Array(OpcDiffer.OpcType.ReturnOpc.id.toByte,
            ReturnOpcDiff.ReturnType.ValueReturn.id.toByte)
106   }
107
108   object StateVar {
109     val issuer: Byte = 0
110     val maker: Byte = 1
111   }
112
113   object DataStack {
114     object initInput {
115       val maxIndex: Byte = 0
116       val unityIndex: Byte = 1
117       val shortTextIndex: Byte = 2
118       val issuerLoadIndex: Byte = 3
119     }
120
121     object supersedeIndex {
122       val newIssuerIndex: Byte = 0
123       val maker: Byte = 1
124     }
125
126     object issueInput {
127       val amountIndex: Byte = 0
128       val issuerGetIndex: Byte = 1
129     }
130
131     object destroyInput {
132       val amountIndex: Byte = 0
133       val issuerGetIndex: Byte = 1
134     }
135
136     object splitInput {
137       val amountIndex: Byte = 0
138       val issuerGetIndex: Byte = 1
139     }
140
141     object sendInput {
142       val receiptIndex: Byte = 0
143       val amountIndex: Byte = 1
144       val callerLoadIndex: Byte = 2
145     }
146
147     object transferInput {
148       val senderIndex: Byte = 0
149       val receiptIndex: Byte = 1
150       val amountIndex: Byte = 2
151     }
```

```
152
153     object depositInput {
154       val senderIndex: Byte = 0
155       val smartIndex: Byte = 1
156       val amountIndex: Byte = 2
157     }
158
159     object withdrawInput {
160       val smartIndex: Byte = 0
161       val receiptIndex: Byte = 1
162       val amountIndex: Byte = 2
163     }
164
165     object balanceOfInput {
166       val addressIndex: Byte = 0
167     }
168
169   }
170
171   object ListOpc {
172     val opcLoadSignerIndex: Array[Byte] = Array(3.toByte)
173     val opcLoadCallerIndex: Array[Byte] = Array(2.toByte)
174
175     val opcCDBVSetIssuerInitIndex: Array[Byte] = Array(StateVar.issuer, DataStack.
          initInput.issuerLoadIndex)
176     val opcCDBVSetIssuerSupersedeIndex: Array[Byte] = Array(StateVar.issuer, DataStack
          .supersedeIndex.newIssuerIndex)
177     val opcCDBVSetMakerIndex: Array[Byte] = Array(StateVar.maker, DataStack.initInput.
          issuerLoadIndex)
178
179     val opcCDBVRGetIssuerIndex: Array[Byte] = Array(StateVar.issuer, 1.toByte)
180     val opcCDBVRGetMakerIndex: Array[Byte] = Array(StateVar.maker, 1.toByte)
181
182     val opcAssertIsCallerOriginIssueIndex: Array[Byte] = Array(DataStack.issueInput.
          issuerGetIndex)
183     val opcAssertIsCallerOriginDestroyIndex: Array[Byte] = Array(DataStack.
          destroyInput.issuerGetIndex)
184     val opcAssertIsCallerOriginSplitIndex: Array[Byte] = Array(DataStack.splitInput.
          issuerGetIndex)
185     val opcAssertIsCallerOriginTransferIndex: Array[Byte] = Array(DataStack.
          transferInput.senderIndex)
186     val opcAssertIsCallerOriginDepositIndex: Array[Byte] = Array(DataStack.
          depositInput.senderIndex)
187     val opcAssertIsCallerOriginWithdrawIndex: Array[Byte] = Array(DataStack.
          withdrawInput.receiptIndex)
188     val opcAssertIsMakerOriginSupersedeIndex: Array[Byte] = Array(DataStack.
          supersedeIndex.maker)
189
190     val opcTDBNewTokenIndex: Array[Byte] = Array(DataStack.initInput.maxIndex,
          DataStack.initInput.unityIndex, DataStack.initInput.shortTextIndex)
191     val opcTDBSplitIndex: Array[Byte] = Array(DataStack.splitInput.amountIndex)
192
193     val opcTDBRTotalIndex: Array[Byte] = Array(0.toByte)
194     val opcTDBRMaxIndex: Array[Byte] = Array(0.toByte)
195
196     val opcTDBADepositIssueIndex: Array[Byte] = Array(DataStack.issueInput.
          issuerGetIndex, DataStack.issueInput.amountIndex)
197     val opcTDBAWithdrawDestroyIndex: Array[Byte] = Array(DataStack.destroyInput.
```

```
        issuerGetIndex, DataStack.destroyInput.amountIndex)
198     val opcTDBATransferSendIndex: Array[Byte] = Array(DataStack.sendInput.
            callerLoadIndex, DataStack.sendInput.receiptIndex, DataStack.sendInput.
            amountIndex)
199     val opcTDBATransferTransferIndex: Array[Byte] = Array(DataStack.transferInput.
            senderIndex, DataStack.transferInput.receiptIndex, DataStack.transferInput.
            amountIndex)
200     val opcTDBATransferDepositIndex: Array[Byte] = Array(DataStack.depositInput.
            senderIndex, DataStack.depositInput.smartIndex, DataStack.depositInput.
            amountIndex)
201     val opcTDBATransferWithdrawIndex: Array[Byte] = Array(DataStack.withdrawInput.
            smartIndex, DataStack.withdrawInput.receiptIndex, DataStack.withdrawInput.
            amountIndex)
202
203     val opcTDBARBalanceOfIndex: Array[Byte] = Array(DataStack.balanceOfInput.
            addressIndex, 1.toByte)
204
205     // init
206     val initOpc: List[Array[Byte]] = List(OpcId.opcLoadSigner, OpcId.opcCDBVSet, OpcId
            .opcCDBVSet, OpcId.opcTDBNewToken)
207     val initOpcIndex: List[Array[Byte]] = List(opcLoadSignerIndex,
            opcCDBVSetIssuerInitIndex, opcCDBVSetMakerIndex, opcTDBNewTokenIndex)
208     // supersede index and opc
209     val supersedeOpc: List[Array[Byte]] = List(OpcId.opcCDBVRGet, OpcId.
            opcAssertIsSignerOrigin, OpcId.opcCDBVSet)
210     val supersedeOpcIndex: List[Array[Byte]] = List(opcCDBVRGetMakerIndex,
            opcAssertIsMakerOriginSupersedeIndex, opcCDBVSetIssuerSupersedeIndex)
211
212     // issue
213     val issueOpc: List[Array[Byte]] = List(OpcId.opcCDBVRGet, OpcId.
            opcAssertIsCallerOrigin, OpcId.opcTDBADeposit)
214     val issueOpcIndex: List[Array[Byte]] = List(opcCDBVRGetIssuerIndex,
            opcAssertIsCallerOriginIssueIndex, opcTDBADepositIssueIndex)
215     // destroy
216     val destroyOpc: List[Array[Byte]] = List(OpcId.opcCDBVRGet, OpcId.
            opcAssertIsCallerOrigin, OpcId.opcTDBAWithdraw)
217     val destroyOpcIndex: List[Array[Byte]] = List(opcCDBVRGetIssuerIndex,
            opcAssertIsCallerOriginDestroyIndex, opcTDBAWithdrawDestroyIndex)
218
219     // split
220     val splitOpc: List[Array[Byte]] = List(OpcId.opcCDBVRGet, OpcId.
            opcAssertIsCallerOrigin, OpcId.opcTDBSplit)
221     val splitOpcIndex: List[Array[Byte]] = List(opcCDBVRGetIssuerIndex,
            opcAssertIsCallerOriginSplitIndex, opcTDBSplitIndex)
222
223     // send
224     val sendOpc: List[Array[Byte]] = List(OpcId.opcLoadCaller, OpcId.opcTDBATransfer)
225     val sendOpcIndex: List[Array[Byte]] = List(opcLoadCallerIndex,
            opcTDBATransferSendIndex)
226     // transfer
227     val transferOpc: List[Array[Byte]] = List(OpcId.opcAssertIsCallerOrigin, OpcId.
            opcTDBATransfer)
228     val transferOpcIndex: List[Array[Byte]] = List(
            opcAssertIsCallerOriginTransferIndex, opcTDBATransferTransferIndex)
229     // deposit
230     val depositOpc: List[Array[Byte]] = List(OpcId.opcAssertIsCallerOrigin, OpcId.
            opcTDBATransfer)
231     val depositOpcIndex: List[Array[Byte]] = List(opcAssertIsCallerOriginDepositIndex,
```

```
        opcTDBATransferDepositIndex)
232   // withdraw
233   val withdrawOpc: List[Array[Byte]] = List(OpcId.opcAssertIsCallerOrigin, OpcId.
          opcTDBATransfer)
234   val withdrawOpcIndex: List[Array[Byte]] = List(
          opcAssertIsCallerOriginWithdrawIndex, opcTDBATransferWithdrawIndex)
235   // totalSupply
236   val totalSupplyOpc: List[Array[Byte]] = List(OpcId.opcTDBROpcTotal, OpcId.
          opcReturnValue)
237   val totalSupplyOpcIndex: List[Array[Byte]] = List(opcTDBRTotalIndex, Array(0.
          toByte))
238   // maxSupplyOpc
239   val maxSupplyOpc: List[Array[Byte]] = List(OpcId.opcTDBROpcMax, OpcId.
          opcReturnValue)
240   val maxSupplyOpcIndex: List[Array[Byte]] = List(opcTDBRMaxIndex, Array(0.toByte))
241   // balanceOfOpc
242   val balanceOfOpc: List[Array[Byte]] = List(OpcId.opcTDBARBalance, OpcId.
          opcReturnValue)
243   val balanceOfOpcIndex: List[Array[Byte]] = List(opcTDBARBalanceOfIndex, Array(1.
          toByte))
244   // getIssuerOpc
245   val getIssuerOpc: List[Array[Byte]] = List(OpcId.opcCDBVRGet, OpcId.opcReturnValue
          )
246   val getIssuerOpcIndex: List[Array[Byte]] = List(Array(StateVar.issuer, 0.toByte),
          Array(0.toByte))
247 }
248
249 object OpcLine {
250   val initOpcLine: Array[Byte] = listOpc(ListOpc.initOpc, ListOpc.initOpcIndex)
251   val supersedeOpcLine: Array[Byte] = listOpc(ListOpc.supersedeOpc, ListOpc.
          supersedeOpcIndex)
252   val issueOpcLine: Array[Byte] = listOpc(ListOpc.issueOpc, ListOpc.issueOpcIndex)
253   val destroyOpcLine: Array[Byte] = listOpc(ListOpc.destroyOpc, ListOpc.
          destroyOpcIndex)
254   val splitOpcLine: Array[Byte] = listOpc(ListOpc.splitOpc, ListOpc.splitOpcIndex)
255   val sendOpcLine: Array[Byte] = listOpc(ListOpc.sendOpc, ListOpc.sendOpcIndex)
256   val transferOpcLine: Array[Byte] = listOpc(ListOpc.transferOpc, ListOpc.
          transferOpcIndex)
257   val depositOpcLine: Array[Byte] = listOpc(ListOpc.depositOpc, ListOpc.
          depositOpcIndex)
258   val withdrawOpcLine: Array[Byte] = listOpc(ListOpc.withdrawOpc, ListOpc.
          withdrawOpcIndex)
259   val totalSupplyOpcLine: Array[Byte] = listOpc(ListOpc.totalSupplyOpc, ListOpc.
          totalSupplyOpcIndex)
260   val maxSupplyOpcLine: Array[Byte] = listOpc(ListOpc.maxSupplyOpc, ListOpc.
          maxSupplyOpcIndex)
261   val balanceOfOpcLine: Array[Byte] = listOpc(ListOpc.balanceOfOpc, ListOpc.
          balanceOfOpcIndex)
262   val getIssuerOpcLine: Array[Byte] = listOpc(ListOpc.getIssuerOpc, ListOpc.
          getIssuerOpcIndex)
263 }
264
265 def protoType(listReturnType: Array[Byte], listParaTypes: Array[Byte]): Array[Byte]
      = {
266   val retType = Deser.serializeArray(listReturnType)
267   val paraType = Deser.serializeArray(listParaTypes)
268   Bytes.concat(retType, paraType)
269 }
```

```
270
271   lazy val nonReturnType: Array[Byte] = Array[Byte]()
272   lazy val onInitTriggerType: Byte = 0
273   lazy val publicFuncType: Byte = 0
274   lazy val initFunc: Array[Byte] = Shorts.toByteArray(FunId.init) ++ Array(
          onInitTriggerType) ++ protoType(nonReturnType, ProtoType.initParaType) ++
          OpcLine.initOpcLine
275   lazy val supersedeFunc: Array[Byte] = Shorts.toByteArray(FunId.supersede) ++ Array(
          publicFuncType) ++ protoType(nonReturnType, ProtoType.supersedeParaType) ++
          OpcLine.supersedeOpcLine
276   lazy val issueFunc: Array[Byte] = Shorts.toByteArray(FunId.issue) ++ Array(
          publicFuncType) ++ protoType(nonReturnType, ProtoType.issueParaType) ++ OpcLine.
          issueOpcLine
277   lazy val destroyFunc: Array[Byte] = Shorts.toByteArray(FunId.destroy) ++ Array(
          publicFuncType) ++ protoType(nonReturnType, ProtoType.destroyParaType) ++
          OpcLine.destroyOpcLine
278   lazy val splitFunc: Array[Byte] = Shorts.toByteArray(FunId.split) ++ Array(
          publicFuncType) ++ protoType(nonReturnType, ProtoType.splitParaType) ++ OpcLine.
          splitOpcLine
279   lazy val sendFunc: Array[Byte] = Shorts.toByteArray(FunId.send) ++ Array(
          publicFuncType) ++ protoType(nonReturnType, ProtoType.sendParaType) ++ OpcLine.
          sendOpcLine
280   lazy val transferFunc: Array[Byte] = Shorts.toByteArray(FunId.transfer) ++ Array(
          publicFuncType) ++ protoType(nonReturnType, ProtoType.transferParaType) ++
          OpcLine.transferOpcLine
281   lazy val depositFunc: Array[Byte] = Shorts.toByteArray(FunId.deposit) ++ Array(
          publicFuncType) ++ protoType(nonReturnType, ProtoType.depositParaType) ++
          OpcLine.depositOpcLine
282   lazy val withdrawFunc: Array[Byte] = Shorts.toByteArray(FunId.withdraw) ++ Array(
          publicFuncType) ++ protoType(nonReturnType, ProtoType.withdrawParaType) ++
          OpcLine.withdrawOpcLine
283   lazy val totalSupplyFunc: Array[Byte] = Shorts.toByteArray(FunId.totalSupply) ++
          Array(publicFuncType) ++ protoType(Array(DataType.Amount.id.toByte), ProtoType.
          totalSupplyParaType) ++ OpcLine.totalSupplyOpcLine
284   lazy val maxSupplyFunc: Array[Byte] = Shorts.toByteArray(FunId.maxSupply) ++ Array(
          publicFuncType) ++ protoType(Array(DataType.Amount.id.toByte), ProtoType.
          maxSupplyParaType) ++ OpcLine.maxSupplyOpcLine
285   lazy val balanceOfFunc: Array[Byte] = Shorts.toByteArray(FunId.balanceOf) ++ Array(
          publicFuncType) ++ protoType(Array(DataType.Amount.id.toByte), ProtoType.
          balanceOfParaType) ++ OpcLine.balanceOfOpcLine
286   lazy val getIssuerFunc: Array[Byte] = Shorts.toByteArray(FunId.getIssuer) ++ Array(
          publicFuncType) ++ protoType(Array(DataType.Account.id.toByte), ProtoType.
          getIssuerParaType) ++ OpcLine.getIssuerOpcLine
287
288   lazy val initFuncWithoutSplit: Array[Byte] = Shorts.toByteArray(FunIdWithoutSplit.
          init) ++ Array(onInitTriggerType) ++ protoType(nonReturnType, ProtoType.
          initParaType) ++ OpcLine.initOpcLine
289   lazy val supersedeFuncWithoutSplit: Array[Byte] = Shorts.toByteArray(
          FunIdWithoutSplit.supersede) ++ Array(publicFuncType) ++ protoType(nonReturnType
          , ProtoType.supersedeParaType) ++ OpcLine.supersedeOpcLine
290   lazy val issueFuncWithoutSplit: Array[Byte] = Shorts.toByteArray(FunIdWithoutSplit.
          issue) ++ Array(publicFuncType) ++ protoType(nonReturnType, ProtoType.
          issueParaType) ++ OpcLine.issueOpcLine
291   lazy val destroyFuncWithoutSplit: Array[Byte] = Shorts.toByteArray(FunIdWithoutSplit
          .destroy) ++ Array(publicFuncType) ++ protoType(nonReturnType, ProtoType.
          destroyParaType) ++ OpcLine.destroyOpcLine
292   lazy val sendFuncWithoutSplit: Array[Byte] = Shorts.toByteArray(FunIdWithoutSplit.
          send) ++ Array(publicFuncType) ++ protoType(nonReturnType, ProtoType.
```

```
    sendParaType) ++ OpcLine.sendOpcLine
  lazy val transferFuncWithoutSplit: Array[Byte] = Shorts.toByteArray(
      FunIdWithoutSplit.transfer) ++ Array(publicFuncType) ++ protoType(nonReturnType,
       ProtoType.transferParaType) ++ OpcLine.transferOpcLine
  lazy val depositFuncWithoutSplit: Array[Byte] = Shorts.toByteArray(FunIdWithoutSplit
      .deposit) ++ Array(publicFuncType) ++ protoType(nonReturnType, ProtoType.
      depositParaType) ++ OpcLine.depositOpcLine
  lazy val withdrawFuncWithoutSplit: Array[Byte] = Shorts.toByteArray(
      FunIdWithoutSplit.withdraw) ++ Array(publicFuncType) ++ protoType(nonReturnType,
       ProtoType.withdrawParaType) ++ OpcLine.withdrawOpcLine
  lazy val totalSupplyFuncWithoutSplit: Array[Byte] = Shorts.toByteArray(
      FunIdWithoutSplit.totalSupply) ++ Array(publicFuncType) ++ protoType(Array(
      DataType.Amount.id.toByte), ProtoType.totalSupplyParaType) ++ OpcLine.
      totalSupplyOpcLine
  lazy val maxSupplyFuncWithoutSplit: Array[Byte] = Shorts.toByteArray(
      FunIdWithoutSplit.maxSupply) ++ Array(publicFuncType) ++ protoType(Array(
      DataType.Amount.id.toByte), ProtoType.maxSupplyParaType) ++ OpcLine.
      maxSupplyOpcLine
  lazy val balanceOfFuncWithoutSplit: Array[Byte] = Shorts.toByteArray(
      FunIdWithoutSplit.balanceOf) ++ Array(publicFuncType) ++ protoType(Array(
      DataType.Amount.id.toByte), ProtoType.balanceOfParaType) ++ OpcLine.
      balanceOfOpcLine
  lazy val getIssuerFuncWithoutSplit: Array[Byte] = Shorts.toByteArray(
      FunIdWithoutSplit.getIssuer) ++ Array(publicFuncType) ++ protoType(Array(
      DataType.Account.id.toByte), ProtoType.getIssuerParaType) ++ OpcLine.
      getIssuerOpcLine

  def textualFunc(name: String, ret: Seq[String], para: Seq[String]): Array[Byte] = {
    val funcByte = Deser.serializeArray(Deser.serilizeString(name))
    val retByte = Deser.serializeArray(Deser.serializeArrays(ret.map(x => Deser.
        serilizeString(x))))
    val paraByte = Deser.serializeArrays(para.map(x => Deser.serilizeString(x)))
    Bytes.concat(funcByte, retByte, paraByte)
  }

  object ParaName {
    val initPara: Seq[String] = Seq("max", "unity", "tokenDescription", "signer")
    val supersedePara: Seq[String] = Seq("newIssuer", "maker")
    val issuePara: Seq[String] = Seq("amount", "issuer")
    val destroyPara: Seq[String] = Seq("amount", "issuer")
    val splitPara: Seq[String] = Seq("newUnity", "issuer")
    val sendPara: Seq[String] = Seq("recipient", "amount", "caller")
    val transferPara: Seq[String]= Seq("sender", "recipient", "amount")
    val depositPara: Seq[String] = Seq("sender", "smart", "amount")
    val withdrawPara: Seq[String]= Seq("smart", "recipient", "amount")
    val totalSupplyPara: Seq[String] = Seq("total")
    val maxSupplyPara: Seq[String] = Seq("max")
    val balanceOfPara: Seq[String] = Seq("address", "balance")
    val getIssuerPara: Seq[String] = Seq("issuer")
  }

  val stateVarName = List("issuer", "maker")
  lazy val stateVarTextual: Array[Byte] = Deser.serializeArrays(stateVarName.map(x =>
      Deser.serilizeString(x)))

  val initFuncBytes: Array[Byte] = textualFunc("init", Seq(), ParaName.initPara)
  val supersedeFuncBytes: Array[Byte] = textualFunc("supersede", Seq(), ParaName.
      supersedePara)
```

```
  val issueFuncBytes: Array[Byte] = textualFunc("issue", Seq(), ParaName.issuePara)
  val destroyFuncBytes: Array[Byte] = textualFunc("destroy", Seq(), ParaName.
      destroyPara)
  val splitFuncBytes: Array[Byte] = textualFunc("split", Seq(), ParaName.splitPara)
  val sendFuncBytes: Array[Byte] = textualFunc("send", Seq(), ParaName.sendPara)
  val transferFuncBytes: Array[Byte] = textualFunc("transfer", Seq(), ParaName.
      transferPara)
  val depositFuncBytes: Array[Byte] = textualFunc("deposit", Seq(), ParaName.
      depositPara)
  val withdrawFuncBytes: Array[Byte] = textualFunc("withdraw", Seq(), ParaName.
      withdrawPara)
  val totalSupplyFuncBytes: Array[Byte] = textualFunc("totalSupply", Seq("total"),
      ParaName.totalSupplyPara)
  val maxSupplyFuncBytes: Array[Byte] = textualFunc("maxSupply", Seq("max"), ParaName.
      maxSupplyPara)
  val balanceOfFuncBytes: Array[Byte] = textualFunc("balanceOf", Seq("balance"),
      ParaName.balanceOfPara)
  val getIssuerFuncBytes: Array[Byte] = textualFunc("getIssuer", Seq("issuer"),
      ParaName.getIssuerPara)

  lazy val triggerTextual: Array[Byte] = Deser.serializeArrays(Seq(initFuncBytes))
  lazy val descriptorTextual: Array[Byte] = Deser.serializeArrays(Seq(
      supersedeFuncBytes, issueFuncBytes,
    destroyFuncBytes, splitFuncBytes, sendFuncBytes, transferFuncBytes,
        depositFuncBytes, withdrawFuncBytes,
    totalSupplyFuncBytes, maxSupplyFuncBytes, balanceOfFuncBytes, getIssuerFuncBytes))
  lazy val descriptorTextualWithoutSplit: Array[Byte] = Deser.serializeArrays(Seq(
      supersedeFuncBytes, issueFuncBytes,
    destroyFuncBytes, sendFuncBytes, transferFuncBytes, depositFuncBytes,
        withdrawFuncBytes,
    totalSupplyFuncBytes, maxSupplyFuncBytes, balanceOfFuncBytes, getIssuerFuncBytes))

}
```

DataEntry.scala

```
package vsys.blockchain.contract

import com.google.common.primitives.{Bytes, Ints, Longs, Shorts}
import play.api.libs.json.{JsObject, JsValue, Json}
import scorex.crypto.encode.Base58
import vsys.account.{Address, PublicKeyAccount, ContractAccount}
import vsys.blockchain.transaction.contract.RegisterContractTransaction.
    MaxDescriptionSize
import vsys.blockchain.transaction.TransactionParser.{AmountLength, KeyLength}
import vsys.blockchain.transaction.ValidationError
import vsys.blockchain.transaction.ValidationError.InvalidDataEntry

import scala.util.Success

case class DataEntry(data: Array[Byte],
                     dataType: DataType.Value) {

  lazy val bytes: Array[Byte] = Array(dataType.id.asInstanceOf[Byte]) ++ data

  lazy val json: JsObject = Json.obj(
    "data" -> toJson(data, dataType),
    "type" -> dataType
  )
```

```
23
24    private def toJson(d: Array[Byte], t: DataType.Value): JsValue = {
25      t match {
26        case DataType.PublicKey => Json.toJson(PublicKeyAccount(d).address)
27        case DataType.Address => Json.toJson(Address.fromBytes(d).right.get.address)
28        case DataType.Amount => Json.toJson(Longs.fromByteArray(d))
29        case DataType.Int32 => Json.toJson(Ints.fromByteArray(d))
30        case DataType.ShortText => Json.toJson(Base58.encode(d))
31        case DataType.ContractAccount => Json.toJson(ContractAccount.fromBytes(d).right.
              get.address)
32      }
33    }
34  }
35
36  object DataEntry {
37
38    def create(data: Array[Byte], dataType: DataType.Value): Either[ValidationError,
          DataEntry] = {
39      dataType match {
40        case DataType.ShortText if checkDataType(Shorts.toByteArray(data.length.toShort)
              ++ data, dataType) => Right(DataEntry(Shorts.toByteArray(data.length.toShort
              ) ++ data, dataType))
41        case _ if checkDataType(data, dataType) => Right(DataEntry(data, dataType))
42        case _ => Left(InvalidDataEntry)
43      }
44    }
45
46    def fromBytes(bytes: Array[Byte]): Either[ValidationError, DataEntry] = {
47      if (bytes.length == 0 || DataType.fromByte(bytes(0)).isEmpty)
48        Left(InvalidDataEntry)
49      else
50        DataType.fromByte(bytes(0)) match {
51          case Some(DataType.ShortText) => create(bytes.slice(3, bytes.length), DataType(
                bytes(0)))
52          case _ => create(bytes.tail, DataType(bytes(0)))
53        }
54    }
55
56    def fromBase58String(base58String: String): Either[ValidationError, Seq[DataEntry]]
          = {
57      Base58.decode(base58String) match {
58        case Success(byteArray) => parseArrays(byteArray)
59        case _ => Left(InvalidDataEntry)
60      }
61    }
62
63    def parseArraySize(bytes: Array[Byte], position: Int): Either[ValidationError, (
          DataEntry, Int)] = {
64      DataType.fromByte(bytes(position)) match {
65        case Some(DataType.PublicKey) if checkDataType(bytes.slice(position + 1,
              position + 1 + KeyLength), DataType.PublicKey) =>
66          Right((DataEntry(bytes.slice(position + 1, position + 1 + KeyLength), DataType.
                PublicKey), position + 1 + KeyLength))
67        case Some(DataType.Address) if checkDataType(bytes.slice(position + 1, position
              + 1 + Address.AddressLength), DataType.Address) =>
68          Right((DataEntry(bytes.slice(position + 1, position + 1 + Address.AddressLength
                ), DataType.Address), position + 1 + Address.AddressLength))
69        case Some(DataType.Amount) if checkDataType(bytes.slice(position + 1, position +
```

```
            1 + AmountLength), DataType.Amount) =>
70        Right((DataEntry(bytes.slice(position + 1, position + 1 + AmountLength),
              DataType.Amount), position + 1 + AmountLength))
71      case Some(DataType.Int32) if checkDataType(bytes.slice(position + 1, position +
            1 + 4), DataType.Int32) =>
72        Right((DataEntry(bytes.slice(position + 1, position + 1 + 4), DataType.Int32),
              position + 1 + 4))
73      case Some(DataType.ShortText) if checkDataType(bytes.slice(position + 1,
            position + 3 + Shorts.fromByteArray(bytes.slice(position + 1, position + 3)))
            , DataType.ShortText) =>
74        Right((DataEntry(bytes.slice(position + 1, position + 3 + Shorts.fromByteArray(
              bytes.slice(position + 1, position + 3))), DataType.ShortText), position +
              3 + Shorts.fromByteArray(bytes.slice(position + 1, position + 3))))
75      case Some(DataType.ContractAccount) if checkDataType(bytes.slice(position + 1,
            position + 1 + ContractAccount.AddressLength), DataType.ContractAccount) =>
76        Right((DataEntry(bytes.slice(position + 1, position + 1 + ContractAccount.
              AddressLength), DataType.ContractAccount), position + 1 + ContractAccount.
              AddressLength))
77      case _ => Left(InvalidDataEntry)
78    }
79  }
80
81  def serializeArrays(ds: Seq[DataEntry]): Array[Byte] = {
82    Shorts.toByteArray(ds.length.toShort) ++ Bytes.concat(ds.map(_.bytes): _*)
83  }
84
85  def right(structure: (Seq[DataEntry], Int)): Either[ValidationError, (Seq[DataEntry
        ], Int)] = Right(structure)
86
87  def parseArrays(bytes: Array[Byte]): Either[ValidationError, Seq[DataEntry]] = {
88    val length = Shorts.fromByteArray(bytes.slice(0, 2))
89    (0 until length).foldLeft(right((Seq.empty[DataEntry], 2))) {
90      case (accPos, _) => accPos.flatMap(ap => parseArraySize(bytes, ap._2) match {
91        case Right((arr, nextPos)) => Right((ap._1 :+ arr, nextPos))
92        case Left(l) => Left(l)
93      })
94
95    } match {
96      case Right((acc, _)) => Right(acc)
97      case Left(l) => Left(l)
98    }
99  }
100
101 private def checkDataType(data: Array[Byte], dataType: DataType.Value): Boolean =
        dataType match {
102     case DataType.PublicKey => data.length == KeyLength
103     case DataType.Address => Address.fromBytes(data).isRight
104     case DataType.Amount => data.length == AmountLength && Longs.fromByteArray(data)
            >= 0
105     case DataType.Int32 => data.length == 4 && Ints.fromByteArray(data) >= 0
106     case DataType.ShortText => Shorts.fromByteArray(data.slice(0, 2)) + 2 == data.
            length && data.length <= 2 + MaxDescriptionSize
107     case DataType.ContractAccount => ContractAccount.fromBytes(data).isRight
108     case _ => false
109   }
110
111 }
```

DataType.scala

```
1  package vsys.blockchain.contract
2
3  object DataType extends Enumeration {
4    val PublicKey = Value(1)
5    val Address = Value(2)
6    val Amount = Value(3)
7    val Int32 = Value(4)
8    val ShortText = Value(5)
9    val ContractAccount = Value(6)
10   val Account = Value(7)
11
12   def fromByte(b: Byte): Option[DataType.Value] = {
13     if (b < DataType.PublicKey.id || b > DataType.Account.id)
14       None
15     else
16       Some(DataType(b))
17   }
18
19   private def check(a: Byte, b: Byte): Boolean = {
20     if (a == b) true
21     else if (a == DataType.Account.id) b == DataType.Address.id || b == DataType.
           ContractAccount.id
22     else if (b == DataType.Account.id) check(b, a)
23     else false
24   }
25
26   def checkTypes(paraTypes: Array[Byte], dataTypes: Array[Byte]): Boolean = {
27     paraTypes.length == dataTypes.length && (paraTypes, dataTypes).zipped.forall {
           case (a, b) => check(a, b) }
28   }
29
30 }
```

ExecutionContext.scala

```
1  package vsys.blockchain.contract
2
3  import vsys.account.{ContractAccount, PublicKeyAccount}
4  import vsys.blockchain.state.reader.StateReader
5  import vsys.blockchain.transaction.{ProvenTransaction, ValidationError}
6  import vsys.blockchain.transaction.ValidationError.{InvalidContractAddress,
       InvalidFunctionIndex}
7  import vsys.blockchain.transaction.contract.{ExecuteContractFunctionTransaction,
       RegisterContractTransaction}
8  import vsys.blockchain.transaction.proof.EllipticCurve25519Proof
9  import vsys.utils.serialization.Deser
10
11 case class ExecutionContext(signers: Seq[PublicKeyAccount],
12                              state: StateReader,
13                              height: Int,
14                              transaction: ProvenTransaction,
15                              contractId: ContractAccount,
16                              opcFunc: Array[Byte],
17                              stateVar: Seq[Array[Byte]],
18                              description: Array[Byte]) {
19
20 }
21
```

```
object ExecutionContext {

  def fromRegConTx(s: StateReader,
                   height: Int,
                   tx: RegisterContractTransaction): Either[ValidationError,
                       ExecutionContext] = {
    val signers = tx.proofs.proofs.map(x => EllipticCurve25519Proof.fromBytes(x.bytes.
        arr).toOption.get.publicKey)
    val contractId = tx.contractId
    val opcFunc = tx.contract.trigger.find(a => (a.length > 2) && (a(2) == 0.toByte)).
        getOrElse(Array[Byte]())
    val stateVar = tx.contract.stateVar
    val description = Deser.serilizeString(tx.description)
    Right(ExecutionContext(signers, s, height, tx, contractId, opcFunc, stateVar,
        description))
  }

  def fromExeConTx(s: StateReader,
                   height: Int,
                   tx: ExecuteContractFunctionTransaction): Either[ValidationError,
                       ExecutionContext] = {
    val signers = tx.proofs.proofs.map(x => EllipticCurve25519Proof.fromBytes(x.bytes.
        arr).toOption.get.publicKey)
    val contractId = tx.contractId
    val description = tx.attachment
    s.contractContent(tx.contractId.bytes) match {
      case Some(c) if tx.funcIdx >=0 && tx.funcIdx < c._3.descriptor.length => Right(
          ExecutionContext(signers, s, height, tx, contractId, c._3.descriptor(tx.
          funcIdx), c._3.stateVar, description))
      case Some(_) => Left(InvalidFunctionIndex)
      case _ => Left(InvalidContractAddress)
    }
  }

}
```

ExecuteContractFunctionTransactionDiff.scala

```
package vsys.blockchain.state.diffs

import vsys.blockchain.contract.ExecutionContext
import vsys.blockchain.state.opcdiffs.OpcFuncDiffer
import vsys.blockchain.state.reader.StateReader
import vsys.blockchain.state.{Diff, LeaseInfo, Portfolio}
import vsys.blockchain.transaction.contract.ExecuteContractFunctionTransaction
import vsys.blockchain.transaction.{TransactionStatus, ValidationError}
import vsys.blockchain.transaction.ValidationError._

object ExecuteContractFunctionTransactionDiff {
  def apply(s: StateReader, height: Int)(tx: ExecuteContractFunctionTransaction):
      Either[ValidationError, Diff] = {
    tx.proofs.firstCurveProof.flatMap( proof => {
      val senderAddress = proof.publicKey.toAddress
      ( for {
        exContext <- ExecutionContext.fromExeConTx(s, height, tx)
        diff <- OpcFuncDiffer(exContext)(tx.data)
      } yield Diff(
        height = height,
        tx = tx,
```

```
      portfolios = Map(senderAddress -> Portfolio(-tx.transactionFee, LeaseInfo.empty
          , Map.empty)),
      tokenDB = diff.tokenDB,
      tokenAccountBalance = diff.tokenAccountBalance,
      contractDB = diff.contractDB,
      contractTokens = diff.contractTokens,
      relatedAddress = diff.relatedAddress,
      chargedFee = tx.transactionFee
    ))
    .left.flatMap( e =>
      Right(Diff(
        height = height,
        tx = tx,
        portfolios = Map(senderAddress -> Portfolio(-tx.transactionFee, LeaseInfo.
            empty, Map.empty)),
        chargedFee = tx.transactionFee,
        txStatus = e match {
          case ce: ContractValidationError => ce.transactionStatus
          case _ => TransactionStatus.Failed
        }
      ))
    )
  })
 }
}
```

RegisterContractTransactionDiff.scala

```
package vsys.blockchain.state.diffs

import vsys.account.Address
import vsys.blockchain.contract.ExecutionContext
import vsys.blockchain.state.reader.StateReader
import vsys.blockchain.state.{Diff, LeaseInfo, Portfolio}
import vsys.blockchain.state.opcdiffs.OpcFuncDiffer
import vsys.blockchain.transaction.contract.RegisterContractTransaction
import vsys.blockchain.transaction.{TransactionStatus, ValidationError}
import vsys.blockchain.transaction.ValidationError._

object RegisterContractTransactionDiff {
  def apply(s: StateReader, height: Int)(tx: RegisterContractTransaction): Either[
      ValidationError, Diff] = {
    /**
      no need to validate the name duplication coz that will create a duplicate
          transaction and
      will fail with duplicated transaction id
    */
    tx.proofs.firstCurveProof.flatMap( proof => {
      val senderAddr: Address = proof.publicKey
      val contractInfo = (height, tx.id, tx.contract, Set(senderAddr))
      ( for {
        exContext <- ExecutionContext.fromRegConTx(s, height, tx)
        diff <- OpcFuncDiffer(exContext)(tx.data)
      } yield Diff(
        height = height,
        tx = tx,
        portfolios = Map(senderAddr -> Portfolio(-tx.transactionFee, LeaseInfo.empty,
            Map.empty)),
        contracts = Map(tx.contractId.bytes -> contractInfo),
```

```
        contractDB = diff.contractDB,
        contractTokens = diff.contractTokens,
        tokenDB = diff.tokenDB,
        tokenAccountBalance = diff.tokenAccountBalance,
        relatedAddress = diff.relatedAddress,
        chargedFee = tx.transactionFee
      ))
      .left.flatMap( e =>
        Right(Diff(
          height = height,
          tx = tx,
          portfolios = Map(senderAddr -> Portfolio(-tx.transactionFee, LeaseInfo.empty,
               Map.empty)),
          chargedFee = tx.transactionFee,
          txStatus = e match {
            case ce: ContractValidationError => ce.transactionStatus
            case _ => TransactionStatus.Failed
          }
        ))
      )
    })
  }
}
```

AssertOpcDiff.scala

```
package vsys.blockchain.state.opcdiffs

import vsys.account.Address
import vsys.blockchain.contract.{DataEntry, DataType, ExecutionContext}
import vsys.blockchain.state.ByteStr
import vsys.blockchain.transaction.ValidationError
import vsys.blockchain.transaction.ValidationError._
import vsys.utils.crypto.hash.FastCryptographicHash

import scala.util.{Left, Right, Try}

object AssertOpcDiff {

  def gtEq0(v: DataEntry): Either[ValidationError, OpcDiff] = {
    if (v.dataType == DataType.Amount && Longs.fromByteArray(v.data) >= 0)
      Right(OpcDiff.empty)
    else
      Left(GenericError(s"Invalid Assert (gteq0): Value ${Longs.fromByteArray(v.data)}
           is negative"))
  }

  def ltEq(v1: DataEntry, v2: DataEntry): Either[ValidationError, OpcDiff] = {
    if (v1.dataType == DataType.Amount && v2.dataType == DataType.Amount
      && Longs.fromByteArray(v1.data) <= Longs.fromByteArray(v2.data))
      Right(OpcDiff.empty)
    else
      Left(GenericError(s"Invalid Assert (lteq0): Value ${Longs.fromByteArray(v2.data)
          } is larger than $v1"))
  }

  def ltInt64(m: DataEntry): Either[ValidationError, OpcDiff] = {
    if (m.dataType == DataType.Amount && Longs.fromByteArray(m.data) <= Long.MaxValue)
      Right(OpcDiff.empty)
```

```
32      else
33        Left(GenericError(s"Invalid Assert (ltint64): Value ${Longs.fromByteArray(m.data
              )} is invalid"))
34    }
35
36    def gt0(v: DataEntry): Either[ValidationError, OpcDiff] = {
37      if (v.dataType == DataType.Amount && Longs.fromByteArray(v.data) > 0)
38        Right(OpcDiff.empty)
39      else
40        Left(GenericError(s"Invalid Assert (gt0): Value $v is non-positive"))
41    }
42
43    def eq(add1: DataEntry, add2: DataEntry): Either[ValidationError, OpcDiff] = {
44      if (add1.dataType == DataType.Address && add2.dataType == DataType.Address
45        && Address.fromBytes(add1.data) == Address.fromBytes(add2.data))
46        Right(OpcDiff.empty)
47      else if (add1.dataType == DataType.Amount && add2.dataType == DataType.Amount
48        && Longs.fromByteArray(add1.data) == Longs.fromByteArray(add2.data))
49        Right(OpcDiff.empty)
50      else
51        Left(GenericError(s"Invalid Assert (eq): DataEntry ${add1.data} is not equal to
              ${add2.data}"))
52    }
53
54    def isCallerOrigin(context: ExecutionContext)(address: DataEntry): Either[
          ValidationError, OpcDiff] = {
55      val signer = context.signers.head
56      if (address.dataType != DataType.Address)
57        Left(ContractDataTypeMismatch)
58      else if (!(address.data sameElements signer.bytes.arr))
59        Left(ContractInvalidCaller)
60      else
61        Right(OpcDiff.empty)
62    }
63
64    def isSignerOrigin(context: ExecutionContext)(address: DataEntry): Either[
          ValidationError, OpcDiff] = {
65      val signer = context.signers.head
66      if (address.dataType != DataType.Address)
67        Left(ContractDataTypeMismatch)
68      else if (!(address.data sameElements signer.bytes.arr))
69        Left(ContractInvalidSigner)
70      else
71        Right(OpcDiff.empty)
72    }
73
74    def checkHash(hashValue: DataEntry, hashKey: DataEntry): Either[ValidationError,
          OpcDiff] = {
75      if (hashValue.dataType != DataType.ShortText || hashKey.dataType != DataType.
            ShortText)
76        Left(ContractDataTypeMismatch)
77      else {
78        val hashResult = ByteStr(FastCryptographicHash(hashKey.data))
79        Either.cond(hashResult.equals(ByteStr(hashValue.data)), OpcDiff.empty,
              ContractInvalidHash)
80      }
81    }
82
```

```
  object AssertType extends Enumeration(1) {
    val GteqZeroAssert, LteqAssert, LtInt64Assert, GtZeroAssert, EqAssert,
        IsCallerOriginAssert, IsSignerOriginAssert = Value
  }

  def parseBytes(context: ExecutionContext)
              (bytes: Array[Byte], data: Seq[DataEntry]): Either[ValidationError,
                  OpcDiff] = {
    if (checkAssertDataIndex(bytes, data.length)) {
      (bytes.headOption.flatMap(f => Try(AssertType(f)).toOption), bytes.length) match
          {
        case (Some(AssertType.GteqZeroAssert), 2) => gtEq0(data(bytes(1)))
        case (Some(AssertType.LteqAssert), 3) => ltEq(data(bytes(1)), data(bytes(2)))
        case (Some(AssertType.LtInt64Assert), 2) => ltInt64(data(bytes(1)))
        case (Some(AssertType.GtZeroAssert), 2) => gt0(data(bytes(1)))
        case (Some(AssertType.EqAssert), 3) => eq(data(bytes(1)), data(bytes(2)))
        case (Some(AssertType.IsCallerOriginAssert), 2) => isCallerOrigin(context)(data
            (bytes(1)))
        case (Some(AssertType.IsSignerOriginAssert), 2) => isSignerOrigin(context)(data
            (bytes(1)))
        case _ => Left(ContractInvalidOPCData)
      }
    }
    else
      Left(ContractInvalidOPCData)
  }

  private def checkAssertDataIndex(bytes: Array[Byte], dataLength: Int): Boolean =
    bytes.tail.max < dataLength && bytes.tail.min >= 0

}
```

CDBVOpcDiff.scala

```
package vsys.blockchain.state.opcdiffs

import vsys.blockchain.state._
import vsys.blockchain.transaction.ValidationError
import vsys.blockchain.transaction.ValidationError.{ContractInvalidOPCData,
    ContractInvalidStateVariable}
import vsys.account.Address
import vsys.blockchain.contract.{DataEntry, DataType, ExecutionContext}
import vsys.blockchain.contract.Contract.checkStateVar

import scala.util.{Left, Right, Try}

object CDBVOpcDiff {

  def set(context: ExecutionContext)(stateVar: Array[Byte],
                                     value: DataEntry): Either[ValidationError, OpcDiff]
                                         = {
    if (!checkStateVar(stateVar, value.dataType)) {
      Left(ContractInvalidStateVariable)
    } else {
      if (value.dataType == DataType.Address) {
        val a = Address.fromBytes(value.data).toOption.get
        Right(OpcDiff(relatedAddress = Map(a -> true),
          contractDB = Map(ByteStr(context.contractId.bytes.arr ++ Array(stateVar(0)))
              -> value.bytes)))
```

```
23        } else {
24          Right(OpcDiff(contractDB = Map(ByteStr(context.contractId.bytes.arr
25            ++ Array(stateVar(0))) -> value.bytes)))
26        }
27      }
28    }
29
30    object CDBVType extends Enumeration {
31      val SetCDBV = Value(1)
32    }
33
34    def parseBytes(context: ExecutionContext)
35                  (bytes: Array[Byte], data: Seq[DataEntry]): Either[ValidationError,
                        OpcDiff] = bytes.headOption.flatMap(f => Try(CDBVType(f)).toOption)
                        match {
36      case Some(CDBVType.SetCDBV) if bytes.length == 3 && bytes(1) < context.stateVar.
            length
37        && bytes.last < data.length && bytes.tail.min >= 0 => set(context)(context.
              stateVar(bytes(1)), data(bytes(2)))
38      case _ => Left(ContractInvalidOPCData)
39    }
40
41  }
```

CDBVROpcDiff.scala

```
1   package vsys.blockchain.state.opcdiffs
2
3   import vsys.blockchain.state._
4   import vsys.blockchain.transaction.ValidationError
5   import vsys.blockchain.transaction.ValidationError.{ContractInvalidOPCData,
        ContractInvalidStateVariable, ContractLocalVariableIndexOutOfRange,
        ContractStateVariableNotDefined}
6   import vsys.blockchain.contract.{DataEntry, DataType, ExecutionContext}
7   import vsys.blockchain.contract.Contract.checkStateVar
8
9   import scala.util.{Left, Right, Try}
10
11  object CDBVROpcDiff {
12
13    def get(context: ExecutionContext)(stateVar: Array[Byte], dataStack: Seq[DataEntry],
14                                       pointer: Byte): Either[ValidationError, Seq[
                                             DataEntry]] = {
15      if (!checkStateVar(stateVar, DataType(stateVar(1)))) {
16        Left(ContractInvalidStateVariable)
17      } else if (pointer > dataStack.length || pointer < 0) {
18        Left(ContractLocalVariableIndexOutOfRange)
19      } else {
20        context.state.contractInfo(ByteStr(context.contractId.bytes.arr ++ Array(
              stateVar(0)))) match {
21          case Some(v) => Right(dataStack.patch(pointer, Seq(v), 1))
22          case _ => Left(ContractStateVariableNotDefined)
23        }
24      }
25    }
26
27    object CDBVRType extends Enumeration {
28      val GetCDBVR = Value(1)
29    }
```

```
30
31    def parseBytes(context: ExecutionContext)
32                  (bytes: Array[Byte], data: Seq[DataEntry]): Either[ValidationError, Seq
                        [DataEntry]] = bytes.headOption.flatMap(f => Try(CDBVRType(f)).
                        toOption) match {
33      case Some(CDBVRType.GetCDBVR) if bytes.length == 3 && bytes(1) < context.stateVar.
            length &&
34        bytes(1) >= 0 => get(context)(context.stateVar(bytes(1)), data, bytes(2))
35      case _ => Left(ContractInvalidOPCData)
36    }
37
38  }
```

LoadOpcDiff.scala

```
1   package vsys.blockchain.state.opcdiffs
2
3   import vsys.blockchain.transaction.ValidationError
4   import vsys.blockchain.transaction.ValidationError.{ContractInvalidOPCData,
        ContractLocalVariableIndexOutOfRange}
5   import vsys.blockchain.contract.{DataEntry, DataType, ExecutionContext}
6
7   import scala.util.{Left, Right, Try}
8
9   object LoadOpcDiff {
10
11    def signer(context: ExecutionContext)(dataStack: Seq[DataEntry], pointer: Byte):
          Either[ValidationError, Seq[DataEntry]] = {
12      if (pointer > dataStack.length || pointer < 0) {
13        Left(ContractLocalVariableIndexOutOfRange)
14      } else {
15        Right(dataStack.patch(pointer, Seq(DataEntry(context.signers.head.bytes.arr,
              DataType.Address)), 1))
16      }
17    }
18
19    def caller(context: ExecutionContext)(dataStack: Seq[DataEntry], pointer: Byte):
          Either[ValidationError, Seq[DataEntry]] = {
20      signer(context)(dataStack, pointer)
21    }
22
23    object LoadType extends Enumeration {
24      val SignerLoad = Value(1)
25      val CallerLoad = Value(2)
26    }
27
28    def parseBytes(context: ExecutionContext)
29                  (bytes: Array[Byte], data: Seq[DataEntry]): Either[ValidationError, Seq
                        [DataEntry]] = bytes.headOption.flatMap(f => Try(LoadType(f)).
                        toOption) match {
30      case Some(LoadType.SignerLoad) if bytes.length == 2 => signer(context)(data, bytes
            .last)
31      case Some(LoadType.CallerLoad) if bytes.length == 2 => caller(context)(data, bytes
            .last)
32      case _ => Left(ContractInvalidOPCData)
33    }
34
35  }
```

OpcDiff.scala

```
package vsys.blockchain.state.opcdiffs

import cats.Monoid
import cats.implicits._
import vsys.blockchain.state.{BlockDiff, ByteStr, Diff}
import vsys.blockchain.transaction.Transaction
import vsys.account.Address

case class OpcDiff(contractDB: Map[ByteStr, Array[Byte]] = Map.empty,
                   contractTokens: Map[ByteStr, Int] = Map.empty,
                   tokenDB: Map[ByteStr, Array[Byte]] = Map.empty,
                   tokenAccountBalance: Map[ByteStr, Long] = Map.empty,
                   relatedAddress: Map[Address, Boolean] = Map.empty) {

}

object OpcDiff {

  val empty = new OpcDiff(Map.empty, Map.empty, Map.empty, Map.empty, Map.empty)

  implicit class OpcDiffExt(d: OpcDiff) {
    def asTransactionDiff(height: Int, tx: Transaction): Diff = Diff(height = height, tx = tx,
                                                                     contractDB = d.contractDB,
                                                                     contractTokens = d.contractTokens,
                                                                     tokenDB = d.tokenDB,
                                                                     tokenAccountBalance = d.tokenAccountBalance,
                                                                     relatedAddress = d.relatedAddress
    )
    def asBlockDiff(height: Int, tx: Transaction): BlockDiff = BlockDiff(d.asTransactionDiff(height, tx), 0, Map.empty)

  }

  implicit val opcDiffMonoid = new Monoid[OpcDiff] {
    override def empty: OpcDiff = OpcDiff.empty

    override def combine(older: OpcDiff, newer: OpcDiff): OpcDiff = OpcDiff(
      contractDB = older.contractDB ++ newer.contractDB,
      contractTokens = Monoid.combine(older.contractTokens, newer.contractTokens),
      tokenDB = older.tokenDB ++ newer.tokenDB,
      tokenAccountBalance = Monoid.combine(older.tokenAccountBalance, newer.tokenAccountBalance),
      relatedAddress = older.relatedAddress ++ newer.relatedAddress
    )
  }
}
```

OpcDiffer.scala

```
package vsys.blockchain.state.opcdiffs

import vsys.blockchain.transaction.ValidationError
import vsys.blockchain.transaction.ValidationError.ContractUnsupportedOPC
```

```
5  import vsys.blockchain.contract.{DataEntry, ExecutionContext}
6
7  import scala.util.Try
8
9
10 object OpcDiffer {
11
12   object OpcType extends Enumeration(1) {
13     val AssertOpc, LoadOpc, CDBVOpc, CDBVROpc, TDBOpc, TDBROpc, TDBAOpc, TDBAROpc,
         ReturnOpc = Value
14   }
15
16   def apply(context: ExecutionContext)
17            (opc: Array[Byte],
18             data: Seq[DataEntry]): Either[ValidationError, (OpcDiff, Seq[DataEntry])] =
                 {
19     opc.headOption.flatMap(f => Try(OpcType(f)).toOption) match {
20       case Some(OpcType.AssertOpc) => opcDiffReturn(AssertOpcDiff.parseBytes(context)(
           opc.tail, data), data)
21       case Some(OpcType.LoadOpc) => seqDataEntryReturn(LoadOpcDiff.parseBytes(context)
           (opc.tail, data))
22       case Some(OpcType.CDBVOpc) => opcDiffReturn(CDBVOpcDiff.parseBytes(context)(opc.
           tail, data), data)
23       case Some(OpcType.CDBVROpc) => seqDataEntryReturn(CDBVROpcDiff.parseBytes(
           context)(opc.tail, data))
24       case Some(OpcType.TDBOpc) => opcDiffReturn(TDBOpcDiff.parseBytes(context)(opc.
           tail, data), data)
25       case Some(OpcType.TDBROpc) => seqDataEntryReturn(TDBROpcDiff.parseBytes(context)
           (opc.tail, data))
26       case Some(OpcType.TDBAOpc) => opcDiffReturn(TDBAOpcDiff.parseBytes(context)(opc.
           tail, data), data)
27       case Some(OpcType.TDBAROpc) => seqDataEntryReturn(TDBAROpcDiff.parseBytes(
           context)(opc.tail, data))
28       case Some(OpcType.ReturnOpc) => seqDataEntryReturn(ReturnOpcDiff.parseBytes(
           context)(opc.tail, data))
29       case _ => Left(ContractUnsupportedOPC)
30     }
31   }
32
33   private def seqDataEntryReturn(res: Either[ValidationError, Seq[DataEntry]]): Either
       [ValidationError, (OpcDiff, Seq[DataEntry])] = {
34     res match {
35       case Right(d: Seq[DataEntry]) => Right((OpcDiff.empty, d))
36       case Left(validationError: ValidationError) => Left(validationError)
37     }
38   }
39
40   private def opcDiffReturn(res: Either[ValidationError, OpcDiff], data: Seq[DataEntry
       ]): Either[ValidationError, (OpcDiff, Seq[DataEntry])] = {
41     res match {
42       case Right(opcDiff: OpcDiff) => Right((opcDiff, data))
43       case Left(validationError: ValidationError) => Left(validationError)
44     }
45   }
46
47 }
```

OpcFuncDiffer.scala

```
1  package vsys.blockchain.state.opcdiffs
2
3  import cats.implicits._
4  import com.google.common.primitives.Shorts
5  import vsys.blockchain.state.reader.CompositeStateReader
6  import vsys.utils.serialization.Deser
7  import vsys.blockchain.transaction.ValidationError
8  import vsys.blockchain.transaction.ValidationError.{ContractDataTypeMismatch,
       ContractInvalidFunction, ContractInvalidOPCData}
9  import vsys.utils.ScorexLogging
10 import vsys.blockchain.contract.{DataEntry, ExecutionContext}
11 import vsys.blockchain.contract.DataType.checkTypes
12
13 import scala.util.{Failure, Success, Try}
14
15
16 object OpcFuncDiffer extends ScorexLogging {
17
18   def right(structure: (OpcDiff, Seq[DataEntry])): Either[ValidationError, (OpcDiff,
         Seq[DataEntry])] = Right(structure)
19
20   def apply(executionContext: ExecutionContext)
21            (data: Seq[DataEntry]): Either[ValidationError, OpcDiff] = {
22     val opcFunc = executionContext.opcFunc
23     val height = executionContext.height
24     val tx = executionContext.transaction
25     val s = executionContext.state
26     fromBytes(opcFunc) match {
27       case Success((_, _, _, listParaTypes, listOpcLines)) =>
28         if (!checkTypes(listParaTypes, data.map(_.dataType.id.toByte).toArray)) {
29           Left(ContractDataTypeMismatch)
30         } else if (listOpcLines.forall(_.length < 2)) {
31           Left(ContractInvalidOPCData)
32         } else {
33           listOpcLines.foldLeft(right((OpcDiff.empty, data))) { case (ei, opc) => ei.
                 flatMap(st =>
34             OpcDiffer(executionContext.copy(state = new CompositeStateReader(s,
35               st._1.asBlockDiff(height, tx))))(opc, st._2) match {
36               case Right((opcDiff, d)) => Right((st._1.combine(opcDiff), d))
37               case Left(l) => Left(l)
38             }
39           )} match {
40             case Right((opcDiff, _)) => Right(opcDiff)
41             case Left(l) => Left(l)
42           }
43         }
44       case Failure(_) => Left(ContractInvalidFunction)
45     }
46   }
47
48   private def fromBytes(bytes: Array[Byte]): Try[(Short, Byte, Array[Byte], Array[Byte
         ], Seq[Array[Byte]])] = Try {
49     val funcIdx = Shorts.fromByteArray(bytes.slice(0, 2))
50     val funcType = bytes(2)
51     val (listReturnTypes, listReturnTypeEnd) = Deser.parseArraySize(bytes, 3)
52     val (listParaTypes, listParaTypeEnd) = Deser.parseArraySize(bytes,
         listReturnTypeEnd)
53     val (listOpcLinesBytes, _) = Deser.parseArraySize(bytes, listParaTypeEnd)
```

```
    val listOpcLines = Deser.parseArrays(listOpcLinesBytes)
    (funcIdx, funcType,listReturnTypes, listParaTypes, listOpcLines)
  }

}
```

ReturnOpcDiff.scala

```
package vsys.blockchain.state.opcdiffs

import vsys.blockchain.transaction.ValidationError
import vsys.blockchain.transaction.ValidationError.{ContractInvalidOPCData,
    ContractUnsupportedOPC}
import vsys.blockchain.contract.{DataEntry, ExecutionContext}

import scala.util.{Left, Try}

object ReturnOpcDiff {

  def value(context: ExecutionContext)(dataStack: Seq[DataEntry], pointer: Byte):
      Either[ValidationError, Seq[DataEntry]] = {
    Left(ContractUnsupportedOPC)
  }

  object ReturnType extends Enumeration {
    val ValueReturn = Value(1)
  }

  def parseBytes(context: ExecutionContext)
              (bytes: Array[Byte], data: Seq[DataEntry]): Either[ValidationError, Seq
                  [DataEntry]] = bytes.headOption.flatMap(f => Try(ReturnType(f)).
                  toOption) match {
    case Some(ReturnType.ValueReturn) if bytes.length == 2 => value(context)(data,
        bytes.last)
    case _ => Left(ContractInvalidOPCData)
  }

}
```

TDBAOpcDiff.scala

```
package vsys.blockchain.state.opcdiffs

import com.google.common.primitives.{Bytes, Ints, Longs}
import vsys.blockchain.state._
import vsys.account.Address
import vsys.blockchain.transaction.ValidationError
import vsys.blockchain.transaction.ValidationError._
import vsys.account.ContractAccount.tokenIdFromBytes
import vsys.blockchain.contract.{DataEntry, DataType}
import vsys.blockchain.contract.ExecutionContext

import scala.util.{Left, Right, Try}

object TDBAOpcDiff {

  def deposit(context: ExecutionContext)
            (issuer: DataEntry, amount: DataEntry, tokenIndex: DataEntry): Either[
                ValidationError, OpcDiff] = {

```

```
    if ((issuer.dataType != DataType.Address) || (amount.dataType != DataType.Amount)
      || (tokenIndex.dataType != DataType.Int32)) {
      Left(ContractDataTypeMismatch)
    } else {
      val contractTokens = context.state.contractTokens(context.contractId.bytes)
      val tokenNumber = Ints.fromByteArray(tokenIndex.data)
      val depositAmount = Longs.fromByteArray(amount.data)
      val tokenID: ByteStr = tokenIdFromBytes(context.contractId.bytes.arr, tokenIndex
          .data).right.get
      val tokenTotalKey = ByteStr(Bytes.concat(tokenID.arr, Array(1.toByte)))
      val issuerBalanceKey = ByteStr(Bytes.concat(tokenID.arr, issuer.data))
      val currentTotal = context.state.tokenAccountBalance(tokenTotalKey)
      val tokenMaxKey = ByteStr(Bytes.concat(tokenID.arr, Array(0.toByte)))
      val tokenMax = Longs.fromByteArray(context.state.tokenInfo(tokenMaxKey).
          getOrElse(
        DataEntry(Longs.toByteArray(0), DataType.Amount)).data)
      if (tokenNumber >= contractTokens || tokenNumber < 0) {
        Left(ContractInvalidTokenIndex)
      } else if (Try(Math.addExact(depositAmount, currentTotal)).isFailure) {
        Left(ValidationError.OverflowError)
      } else if (depositAmount < 0) {
        Left(ContractInvalidAmount)
      } else if (depositAmount + currentTotal > tokenMax) {
        Left(ContractTokenMaxExceeded)
      } else {
        val a = Address.fromBytes(issuer.data).toOption.get
        Right(OpcDiff(relatedAddress = Map(a -> true),
          tokenAccountBalance = Map(tokenTotalKey -> depositAmount, issuerBalanceKey ->
              depositAmount)))
      }
    }
  }

  def depositWithoutTokenIndex(context: ExecutionContext)
            (issuer: DataEntry, amount: DataEntry): Either[ValidationError, OpcDiff] =
                {

    val tokenIndex = DataEntry(Ints.toByteArray(0), DataType.Int32)
    deposit(context)(issuer, amount, tokenIndex)
  }

  def withdraw(context: ExecutionContext)
             (issuer: DataEntry, amount: DataEntry, tokenIndex: DataEntry): Either[
                 ValidationError, OpcDiff] = {

    if ((issuer.dataType != DataType.Address) || (amount.dataType != DataType.Amount)
      || (tokenIndex.dataType != DataType.Int32)) {
      Left(ContractDataTypeMismatch)
    } else {
      val contractTokens = context.state.contractTokens(context.contractId.bytes)
      val tokenNumber = Ints.fromByteArray(tokenIndex.data)
      val withdrawAmount = Longs.fromByteArray(amount.data)
      val tokenID: ByteStr = tokenIdFromBytes(context.contractId.bytes.arr, tokenIndex
          .data).right.get
      val tokenTotalKey = ByteStr(Bytes.concat(tokenID.arr, Array(1.toByte)))
      val issuerBalanceKey = ByteStr(Bytes.concat(tokenID.arr, issuer.data))
      val issuerCurrentBalance = context.state.tokenAccountBalance(issuerBalanceKey)
      if (tokenNumber >= contractTokens || tokenNumber < 0) {
```

```
71        Left(ContractInvalidTokenIndex)
72      } else if (withdrawAmount > issuerCurrentBalance) {
73        Left(ContractTokenBalanceInsufficient)
74      } else if (withdrawAmount < 0){
75        Left(ContractInvalidAmount)
76      }
77      else {
78        val a = Address.fromBytes(issuer.data).toOption.get
79        Right(OpcDiff(relatedAddress = Map(a -> true),
80          tokenAccountBalance = Map(tokenTotalKey -> -withdrawAmount, issuerBalanceKey
              -> -withdrawAmount)
81        ))
82      }
83    }
84  }
85
86  def withdrawWithoutTokenIndex(context: ExecutionContext)
87                               (issuer: DataEntry, amount: DataEntry): Either[
                                     ValidationError, OpcDiff] = {
88
89    val tokenIndex = DataEntry(Ints.toByteArray(0), DataType.Int32)
90    withdraw(context)(issuer, amount, tokenIndex)
91  }
92
93  def transfer(context: ExecutionContext)
94              (sender: DataEntry, recipient: DataEntry, amount: DataEntry,
95               tokenIndex: DataEntry): Either[ValidationError, OpcDiff] = {
96
97    if (sender.dataType == DataType.ContractAccount) {
98      Left(ContractUnsupportedWithdraw)
99    } else if (recipient.dataType == DataType.ContractAccount) {
100     Left(ContractUnsupportedDeposit)
101   } else if ((sender.dataType != DataType.Address) || (recipient.dataType !=
          DataType.Address) ||
102     (amount.dataType != DataType.Amount) || (tokenIndex.dataType != DataType.Int32))
            {
103     Left(ContractDataTypeMismatch)
104   } else {
105     val contractTokens = context.state.contractTokens(context.contractId.bytes)
106     val tokenNumber = Ints.fromByteArray(tokenIndex.data)
107     val transferAmount = Longs.fromByteArray(amount.data)
108     val tokenID: ByteStr = tokenIdFromBytes(context.contractId.bytes.arr, tokenIndex
            .data).right.get
109     val senderBalanceKey = ByteStr(Bytes.concat(tokenID.arr, sender.data))
110     val senderCurrentBalance = context.state.tokenAccountBalance(senderBalanceKey)
111     val recipientBalanceKey = ByteStr(Bytes.concat(tokenID.arr, recipient.data))
112     val recipientCurrentBalance = context.state.tokenAccountBalance(
            recipientBalanceKey)
113     if (tokenNumber >= contractTokens || tokenNumber < 0) {
114       Left(ContractInvalidTokenIndex)
115     } else if (transferAmount > senderCurrentBalance) {
116       Left(ContractTokenBalanceInsufficient)
117     } else if (Try(Math.addExact(transferAmount, recipientCurrentBalance)).isFailure
          ) {
118       Left(ValidationError.OverflowError)
119     } else if (transferAmount < 0) {
120       Left(ContractInvalidAmount)
121     } else {
```

```
122         val s = Address.fromBytes(sender.data).toOption.get
123         val r = Address.fromBytes(recipient.data).toOption.get
124         if (sender.bytes sameElements recipient.bytes) {
125           Right(OpcDiff(relatedAddress = Map(s -> true, r -> true)
126           ))
127         } else {
128           Right(OpcDiff(relatedAddress = Map(s -> true, r -> true),
129             tokenAccountBalance = Map(senderBalanceKey -> -transferAmount,
130               recipientBalanceKey -> transferAmount)
131           ))
132         }
133       }
134     }
135   }
136
137   def transferWithoutTokenIndex(context: ExecutionContext)
138                                (sender: DataEntry, recipient: DataEntry, amount: DataEntry
                                       ): Either[ValidationError, OpcDiff] = {
139
140     val tokenIndex = DataEntry(Ints.toByteArray(0), DataType.Int32)
141     transfer(context)(sender, recipient, amount, tokenIndex)
142   }
143
144   object TDBAType extends Enumeration {
145     val DepositTDBA = Value(1)
146     val WithdrawTDBA = Value(2)
147     val TransferTDBA = Value(3)
148   }
149
150   def parseBytes(context: ExecutionContext)
151                 (bytes: Array[Byte], data: Seq[DataEntry]): Either[ValidationError,
                          OpcDiff] = {
152     if (checkTDBADataIndex(bytes, data.length)) {
153       (bytes.headOption.flatMap(f => Try(TDBAType(f)).toOption), bytes.length) match {
154         case (Some(TDBAType.DepositTDBA), 3) => depositWithoutTokenIndex(context)(data(
                  bytes(1)), data(bytes(2)))
155         case (Some(TDBAType.DepositTDBA), 4) => deposit(context)(data(bytes(1)), data(
                  bytes(2)), data(bytes(3)))
156         case (Some(TDBAType.WithdrawTDBA), 3) => withdrawWithoutTokenIndex(context)(
                  data(bytes(1)), data(bytes(2)))
157         case (Some(TDBAType.WithdrawTDBA), 4) => withdraw(context)(data(bytes(1)), data
                  (bytes(2)), data(bytes(3)))
158         case (Some(TDBAType.TransferTDBA), 4) => transferWithoutTokenIndex(context)(
                  data(bytes(1)), data(bytes(2)), data(bytes(3)))
159         case (Some(TDBAType.TransferTDBA), 5) => transfer(context)(data(bytes(1)), data
                  (bytes(2)), data(bytes(3)), data(bytes(4)))
160         case _ => Left(ContractInvalidOPCData)
161       }
162     }
163     else
164       Left(ContractInvalidOPCData)
165   }
166
167   private def checkTDBADataIndex(bytes: Array[Byte], dataLength: Int): Boolean =
168     bytes.tail.max < dataLength && bytes.tail.min >= 0
169
170 }
```

TDBAROpcDiff.scala

```
1  package vsys.blockchain.state.opcdiffs
2
3  import com.google.common.primitives.{Bytes, Ints, Longs}
4  import vsys.blockchain.state._
5  import vsys.blockchain.transaction.ValidationError
6  import vsys.blockchain.transaction.ValidationError.{ContractDataTypeMismatch,
       ContractInvalidOPCData, ContractInvalidTokenIndex,
       ContractLocalVariableIndexOutOfRange}
7  import vsys.account.ContractAccount.tokenIdFromBytes
8  import vsys.blockchain.contract.{DataEntry, DataType}
9  import vsys.blockchain.contract.ExecutionContext
10
11 import scala.util.{Left, Right, Try}
12
13 object TDBAROpcDiff {
14
15   def balance(context: ExecutionContext)(address: DataEntry, tokenIndex: DataEntry,
16                                          dataStack: Seq[DataEntry], pointer: Byte): Either
                                               [ValidationError, Seq[DataEntry]] = {
17
18     if (tokenIndex.dataType != DataType.Int32 || address.dataType != DataType.Address)
           {
19       Left(ContractDataTypeMismatch)
20     } else if (pointer > dataStack.length || pointer < 0) {
21       Left(ContractLocalVariableIndexOutOfRange)
22     } else {
23       val contractTokens = context.state.contractTokens(context.contractId.bytes)
24       val tokenNumber = Ints.fromByteArray(tokenIndex.data)
25       val tokenID: ByteStr = tokenIdFromBytes(context.contractId.bytes.arr, tokenIndex
             .data).right.get
26       val tokenBalanceKey = ByteStr(Bytes.concat(tokenID.arr, address.data))
27       if (tokenNumber >= contractTokens || tokenNumber < 0) {
28         Left(ContractInvalidTokenIndex)
29       } else {
30         val b = context.state.tokenAccountBalance(tokenBalanceKey)
31         Right(dataStack.patch(pointer, Seq(DataEntry(Longs.toByteArray(b), DataType.
               Amount)), 1))
32       }
33     }
34   }
35
36   def balanceWithoutTokenIndex(context: ExecutionContext)(address: DataEntry,
37                                          dataStack: Seq[DataEntry], pointer: Byte): Either
                                               [ValidationError, Seq[DataEntry]] = {
38
39     val tokenIndex = DataEntry(Ints.toByteArray(0), DataType.Int32)
40     balance(context)(address, tokenIndex, dataStack, pointer)
41   }
42
43   object TDBARType extends Enumeration {
44     val BalanceTBDAR= Value(1)
45   }
46
47   def parseBytes(context: ExecutionContext)
48                 (bytes: Array[Byte], data: Seq[DataEntry]): Either[ValidationError, Seq
                       [DataEntry]] = bytes.headOption.flatMap(f => Try(TDBARType(f)).
                       toOption) match {
```

```
49      case Some(TDBARType.BalanceTBDAR) if checkInput(bytes.slice(0, bytes.length - 1),
            2, context.stateVar.length, data.length, 1) =>
50        balanceWithoutTokenIndex(context)(data(bytes(1)), data, bytes(2))
51      case Some(TDBARType.BalanceTBDAR) if checkInput(bytes.slice(0, bytes.length - 1),
            3, context.stateVar.length, data.length, 1) =>
52        balance(context)(data(bytes(1)), data(bytes(2)), data, bytes(3))
53      case _ => Left(ContractInvalidOPCData)
54    }
55
56    private def checkInput(bytes: Array[Byte], bLength: Int, stateVarLength: Int,
          dataLength: Int, sep: Int): Boolean = {
57      bytes.length == bLength && bytes.slice(1, sep).forall(_ < stateVarLength) && bytes
            .slice(sep, bLength).forall(_ < dataLength) && bytes.tail.min >= 0
58    }
59
60  }
```

TDBOpcDiff.scala

```
1   package vsys.blockchain.state.opcdiffs
2
3   import com.google.common.primitives.{Bytes, Ints, Longs}
4   import vsys.blockchain.state._
5   import vsys.blockchain.transaction.ValidationError
6   import vsys.blockchain.transaction.ValidationError.{ContractDataTypeMismatch,
        ContractInvalidOPCData, ContractInvalidTokenIndex, ContractInvalidTokenInfo}
7   import vsys.account.ContractAccount.tokenIdFromBytes
8   import vsys.blockchain.contract.{DataEntry, DataType, ExecutionContext}
9
10  import scala.util.{Left, Right, Try}
11
12  object TDBOpcDiff {
13
14    def newToken(context: ExecutionContext)
15                (max: DataEntry, unity: DataEntry, desc: DataEntry):Either[
                        ValidationError, OpcDiff] = {
16
17      if (max.dataType != DataType.Amount || unity.dataType != DataType.Amount || desc.
            dataType != DataType.ShortText) {
18        Left(ContractDataTypeMismatch)
19      } else if (Longs.fromByteArray(max.data) < 0) {
20        Left(ContractInvalidTokenInfo)
21      } else if (Longs.fromByteArray(unity.data) <= 0) {
22        Left(ContractInvalidTokenInfo)
23      } else {
24        val contractTokens = context.state.contractTokens(context.contractId.bytes)
25        val tokenID: ByteStr = tokenIdFromBytes(context.contractId.bytes.arr, Ints.
              toByteArray(contractTokens)).right.get
26        val tokenMaxKey = Bytes.concat(tokenID.arr, Array(0.toByte))
27        val tokenTotalKey = Bytes.concat(tokenID.arr, Array(1.toByte))
28        val tokenUnityKey = Bytes.concat(tokenID.arr, Array(2.toByte))
29        val tokenDescKey = Bytes.concat(tokenID.arr, Array(3.toByte))
30        Right(OpcDiff(
31          tokenDB = Map(
32            ByteStr(tokenMaxKey) -> max.bytes,
33            ByteStr(tokenUnityKey) -> unity.bytes,
34            ByteStr(tokenDescKey) -> desc.bytes),
35          contractTokens = Map(context.contractId.bytes -> 1),
36          tokenAccountBalance = Map(ByteStr(tokenTotalKey) -> 0L)
```

```
37      ))
38    }
39  }
40
41  def split(context: ExecutionContext)
42          (newUnity: DataEntry, tokenIndex: DataEntry): Either[ValidationError,
                OpcDiff] = {
43
44    if (newUnity.dataType != DataType.Amount || tokenIndex.dataType != DataType.Int32)
          {
45      Left(ContractDataTypeMismatch)
46    } else {
47      val contractTokens = context.state.contractTokens(context.contractId.bytes)
48      val tokenNumber = Ints.fromByteArray(tokenIndex.data)
49      val newUnityValue = Longs.fromByteArray(newUnity.data)
50      val tokenID: ByteStr = tokenIdFromBytes(context.contractId.bytes.arr, tokenIndex
            .data).right.get
51      val tokenUnityKey = ByteStr(Bytes.concat(tokenID.arr, Array(2.toByte)))
52      if (tokenNumber >= contractTokens || tokenNumber < 0) {
53        Left(ContractInvalidTokenIndex)
54      } else if (newUnityValue <= 0) {
55        Left(ContractInvalidTokenInfo)
56      } else {
57        Right(OpcDiff(tokenDB = Map(tokenUnityKey -> newUnity.bytes)))
58      }
59    }
60  }
61
62  def splitWithoutTokenIndex(context: ExecutionContext)
63          (newUnity: DataEntry): Either[ValidationError, OpcDiff] = {
64
65    val tokenIndex = DataEntry(Ints.toByteArray(0), DataType.Int32)
66    split(context)(newUnity, tokenIndex)
67  }
68
69  object TDBType extends Enumeration {
70    val NewTokenTDB = Value(1)
71    val SplitTDB = Value(2)
72  }
73
74  def parseBytes(context: ExecutionContext)
75              (bytes: Array[Byte], data: Seq[DataEntry]): Either[ValidationError,
                  OpcDiff] = {
76    if (checkTDBDataIndex(bytes, data.length)) {
77      (bytes.headOption.flatMap(f => Try(TDBType(f)).toOption), bytes.length) match {
78        case (Some(TDBType.NewTokenTDB), 4) => newToken(context)(data(bytes(1)), data(
              bytes(2)), data(bytes(3)))
79        case (Some(TDBType.SplitTDB), 2) => splitWithoutTokenIndex(context)(data(bytes
              (1)))
80        case (Some(TDBType.SplitTDB), 3) => split(context)(data(bytes(1)), data(bytes
              (2)))
81        case _ => Left(ContractInvalidOPCData)
82      }
83    }
84    else
85      Left(ContractInvalidOPCData)
86  }
87
```

```
  private def checkTDBDataIndex(bytes: Array[Byte], dataLength: Int): Boolean =
    bytes.tail.max < dataLength && bytes.tail.min >= 0

}
```

TDBROpcDiff.scala

```
package vsys.blockchain.state.opcdiffs

import com.google.common.primitives.{Bytes, Ints, Longs}
import vsys.blockchain.state._
import vsys.blockchain.transaction.ValidationError
import vsys.blockchain.transaction.ValidationError.{ContractDataTypeMismatch,
    ContractInvalidOPCData, ContractInvalidTokenIndex, ContractInvalidTokenInfo,
    ContractLocalVariableIndexOutOfRange}
import vsys.account.ContractAccount.tokenIdFromBytes
import vsys.blockchain.contract.{DataEntry, DataType, ExecutionContext}

import scala.util.{Left, Right, Try}

object TDBROpcDiff {

  def max(context: ExecutionContext)(tokenIndex: DataEntry,
                                     dataStack: Seq[DataEntry], pointer: Byte): Either[
                                        ValidationError, Seq[DataEntry]] = {

    if (tokenIndex.dataType != DataType.Int32) {
      Left(ContractDataTypeMismatch)
    } else if (pointer > dataStack.length || pointer < 0) {
      Left(ContractLocalVariableIndexOutOfRange)
    } else {
      val contractTokens = context.state.contractTokens(context.contractId.bytes)
      val tokenNumber = Ints.fromByteArray(tokenIndex.data)
      val tokenID: ByteStr = tokenIdFromBytes(context.contractId.bytes.arr, tokenIndex
          .data).right.get
      val tokenMaxKey = ByteStr(Bytes.concat(tokenID.arr, Array(0.toByte)))
      if (tokenNumber >= contractTokens || tokenNumber < 0) {
        Left(ContractInvalidTokenIndex)
      } else {
        context.state.tokenInfo(tokenMaxKey) match {
          case Some(v) => Right(dataStack.patch(pointer, Seq(v), 1))
          case _ => Left(ContractInvalidTokenInfo)
        }
      }
    }
  }

  def maxWithoutTokenIndex(context: ExecutionContext)(dataStack: Seq[DataEntry],
      pointer: Byte): Either[ValidationError, Seq[DataEntry]] = {

    val tokenIndex = DataEntry(Ints.toByteArray(0), DataType.Int32)
    max(context)(tokenIndex, dataStack, pointer)
  }

  // in current version only total store in tokenAccountBalance DB
  def total(context: ExecutionContext)(tokenIndex: DataEntry,
                                       dataStack: Seq[DataEntry], pointer: Byte): Either[
                                           ValidationError, Seq[DataEntry]] = {

```

```
47      if (tokenIndex.dataType != DataType.Int32) {
48        Left(ContractDataTypeMismatch)
49      } else if (pointer > dataStack.length || pointer < 0) {
50        Left(ContractLocalVariableIndexOutOfRange)
51      } else {
52        val contractTokens = context.state.contractTokens(context.contractId.bytes)
53        val tokenNumber = Ints.fromByteArray(tokenIndex.data)
54        val tokenID: ByteStr = tokenIdFromBytes(context.contractId.bytes.arr, tokenIndex
              .data).right.get
55        val tokenTotalKey = ByteStr(Bytes.concat(tokenID.arr, Array(1.toByte)))
56        if (tokenNumber >= contractTokens || tokenNumber < 0) {
57          Left(ContractInvalidTokenIndex)
58        } else {
59          val t = context.state.tokenAccountBalance(tokenTotalKey)
60          Right(dataStack.patch(pointer, Seq(DataEntry(Longs.toByteArray(t), DataType.
                Amount)), 1))
61        }
62      }
63    }
64
65    def totalWithoutTokenIndex(context: ExecutionContext)(dataStack: Seq[DataEntry],
          pointer: Byte): Either[ValidationError, Seq[DataEntry]] = {
66
67      val tokenIndex = DataEntry(Ints.toByteArray(0), DataType.Int32)
68      total(context)(tokenIndex, dataStack, pointer)
69    }
70
71    def unity(context: ExecutionContext)(tokenIndex: DataEntry,
72                                         dataStack: Seq[DataEntry], pointer: Byte): Either[
                                             ValidationError, Seq[DataEntry]] = {
73
74      if (tokenIndex.dataType != DataType.Int32) {
75        Left(ContractDataTypeMismatch)
76      } else if (pointer > dataStack.length || pointer < 0) {
77        Left(ContractLocalVariableIndexOutOfRange)
78      } else {
79        val contractTokens = context.state.contractTokens(context.contractId.bytes)
80        val tokenNumber = Ints.fromByteArray(tokenIndex.data)
81        val tokenID: ByteStr = tokenIdFromBytes(context.contractId.bytes.arr, tokenIndex
              .data).right.get
82        val tokenUnityKey = ByteStr(Bytes.concat(tokenID.arr, Array(2.toByte)))
83        if (tokenNumber >= contractTokens || tokenNumber < 0) {
84          Left(ContractInvalidTokenIndex)
85        } else {
86          context.state.tokenInfo(tokenUnityKey) match {
87            case Some(v) => Right(dataStack.patch(pointer, Seq(v), 1))
88            case _ => Left(ContractInvalidTokenInfo)
89          }
90        }
91      }
92    }
93
94    def unityWithoutTokenIndex(context: ExecutionContext)(dataStack: Seq[DataEntry],
          pointer: Byte): Either[ValidationError, Seq[DataEntry]] = {
95
96      val tokenIndex = DataEntry(Ints.toByteArray(0), DataType.Int32)
97      unity(context)(tokenIndex, dataStack, pointer)
98    }
```

```
99
100 def desc(context: ExecutionContext)(tokenIndex: DataEntry,
101                                    dataStack: Seq[DataEntry], pointer: Byte): Either[
                                           ValidationError, Seq[DataEntry]] = {
102
103   if (tokenIndex.dataType != DataType.Int32) {
104     Left(ContractDataTypeMismatch)
105   } else if (pointer > dataStack.length || pointer < 0) {
106     Left(ContractLocalVariableIndexOutOfRange)
107   } else {
108     val contractTokens = context.state.contractTokens(context.contractId.bytes)
109     val tokenNumber = Ints.fromByteArray(tokenIndex.data)
110     val tokenID: ByteStr = tokenIdFromBytes(context.contractId.bytes.arr, tokenIndex
          .data).right.get
111     val tokenDescKey = ByteStr(Bytes.concat(tokenID.arr, Array(3.toByte)))
112     if (tokenNumber >= contractTokens || tokenNumber < 0) {
113       Left(ContractInvalidTokenIndex)
114     } else {
115       context.state.tokenInfo(tokenDescKey) match {
116         case Some(v) => Right(dataStack.patch(pointer, Seq(v), 1))
117         case _ => Left(ContractInvalidTokenInfo)
118       }
119     }
120   }
121 }
122
123 def descWithoutTokenIndex(context: ExecutionContext)(dataStack: Seq[DataEntry],
      pointer: Byte): Either[ValidationError, Seq[DataEntry]] = {
124
125   val tokenIndex = DataEntry(Ints.toByteArray(0), DataType.Int32)
126   desc(context)(tokenIndex, dataStack, pointer)
127 }
128
129 object TDBRType extends Enumeration(1) {
130   val MaxTDBR, TotalTDBR, UnityTDBR, DescTDBR = Value
131 }
132
133 def parseBytes(context: ExecutionContext)
134               (bytes: Array[Byte], data: Seq[DataEntry]): Either[ValidationError, Seq
                      [DataEntry]] = {
135   if (checkTDBRODataIndex(bytes.slice(0, bytes.length - 1), data.length)) {
136     getTDBRDiff(context)(bytes, data)
137   }
138   else
139     Left(ContractInvalidOPCData)
140 }
141
142 private def getTDBRDiff(context: ExecutionContext)
143                        (bytes: Array[Byte], data: Seq[DataEntry]): Either[
                               ValidationError, Seq[DataEntry]] = {
144   (bytes.headOption.flatMap(f => Try(TDBRType(f)).toOption), bytes.length) match {
145     case (Some(TDBRType.MaxTDBR), 2) => maxWithoutTokenIndex(context)(data, bytes(1)
          )
146     case (Some(TDBRType.MaxTDBR), 3) => max(context)(data(bytes(1)), data, bytes(2))
147     case (Some(TDBRType.TotalTDBR), 2) => totalWithoutTokenIndex(context)(data,
          bytes(1))
148     case (Some(TDBRType.TotalTDBR), 3) => total(context)(data(bytes(1)), data, bytes
          (2))
```

```
149       case (Some(TDBRType.UnityTDBR), 2) => unityWithoutTokenIndex(context)(data,
              bytes(1))
150       case (Some(TDBRType.UnityTDBR), 3) => unity(context)(data(bytes(1)), data, bytes
              (2))
151       case (Some(TDBRType.DescTDBR), 2) => descWithoutTokenIndex(context)(data, bytes
              (1))
152       case (Some(TDBRType.DescTDBR), 3) => desc(context)(data(bytes(1)), data, bytes
              (2))
153       case _ => Left(ContractInvalidOPCData)
154     }
155   }
156
157   private def checkTDBRODataIndex(bytes: Array[Byte], dataLength: Int): Boolean =
158     bytes.length == 1 || (bytes.tail.max < dataLength && bytes.tail.min >= 0)
159 }
```

ExecuteContractFunctionTransaction.scala

```
1  package vsys.blockchain.transaction.contract
2
3  import com.google.common.primitives.{Bytes, Longs, Shorts}
4  import play.api.libs.json.{JsObject, Json}
5  import scorex.crypto.encode.Base58
6  import vsys.account.{ContractAccount, PrivateKeyAccount, PublicKeyAccount}
7  import vsys.blockchain.contract.DataEntry
8  import vsys.blockchain.state.ByteStr
9  import vsys.blockchain.transaction.TransactionParser._
10 import vsys.blockchain.transaction._
11 import vsys.blockchain.transaction.proof._
12 import vsys.utils.base58Length
13 import vsys.utils.serialization.{BytesSerializable, Deser}
14
15 import scala.util.{Failure, Success, Try}
16
17 case class ExecuteContractFunctionTransaction private(contractId: ContractAccount,
18                                                       funcIdx: Short,
19                                                       data: Seq[DataEntry],
20                                                       attachment: Array[Byte],
21                                                       transactionFee: Long,
22                                                       feeScale: Short,
23                                                       timestamp: Long,
24                                                       proofs: Proofs) extends
                                                           ProvenTransaction {
25
26   val transactionType = TransactionType.ExecuteContractFunctionTransaction
27
28   lazy val toSign: Array[Byte] = Bytes.concat(
29     Array(transactionType.id.toByte),
30     contractId.bytes.arr,
31     Shorts.toByteArray(funcIdx),
32     Deser.serializeArray(DataEntry.serializeArrays(data)),
33     BytesSerializable.arrayWithSize(attachment),
34     Longs.toByteArray(transactionFee),
35     Shorts.toByteArray(feeScale),
36     Longs.toByteArray(timestamp)
37   )
38
39   override lazy val json: JsObject = jsonBase() ++ Json.obj(
40     "contractId" -> contractId.address,
```

```
    "functionIndex" -> funcIdx,
    "functionData" -> Base58.encode(DataEntry.serializeArrays(data)),
    "attachment" -> Base58.encode(attachment),
    "timestamp" -> timestamp
  )

  override lazy val bytes: Array[Byte] = Bytes.concat(toSign, proofs.bytes)

}

object ExecuteContractFunctionTransaction extends TransactionParser {

  val MaxDescriptionSize = 140
  val maxDescriptionStringSize: Int = base58Length(MaxDescriptionSize)

  def parseTail(bytes: Array[Byte]): Try[ExecuteContractFunctionTransaction] = Try {
    (for {
      contractId <- ContractAccount.fromBytes(bytes.slice(0, ContractAccount.
          AddressLength))
      funcIdx = Shorts.fromByteArray(bytes.slice(ContractAccount.AddressLength,
          ContractAccount.AddressLength + 2))
      (dataBytes, dataEnd) = Deser.parseArraySize(bytes, ContractAccount.AddressLength
          + 2)
      data <- DataEntry.parseArrays(dataBytes)
      (description, descriptionEnd) = Deser.parseArraySize(bytes, dataEnd)
      fee = Longs.fromByteArray(bytes.slice(descriptionEnd, descriptionEnd + 8))//CTK:
          slice starting inclusive ending exclusive
      feeScale = Shorts.fromByteArray(bytes.slice(descriptionEnd + 8, descriptionEnd +
          10))
      timestamp = Longs.fromByteArray(bytes.slice(descriptionEnd + 10, descriptionEnd
          + 18))
      proofs <- Proofs.fromBytes(bytes.slice(descriptionEnd + 18, bytes.length))
      tx <- ExecuteContractFunctionTransaction.createWithProof(contractId, funcIdx,
        data, description, fee, feeScale, timestamp, proofs)
    } yield tx).fold(left => Failure(new Exception(left.toString)), right => Success(
        right))
  }.flatten

  def createWithProof(contractId: ContractAccount,
                      funcIdx: Short,
                      data: Seq[DataEntry],
                      attachment: Array[Byte],
                      fee: Long,
                      feeScale: Short,
                      timestamp: Long,
                      proofs: Proofs): Either[ValidationError,
                          ExecuteContractFunctionTransaction] =
    if (attachment.length > MaxDescriptionSize) {
      Left(ValidationError.TooBigArray)
    } else if (fee <= 0) {
      Left(ValidationError.InsufficientFee)
    } else if (feeScale != DefaultFeeScale) {
      Left(ValidationError.WrongFeeScale(feeScale))
    } else {
      Right(ExecuteContractFunctionTransaction(contractId, funcIdx, data, attachment,
          fee, feeScale, timestamp, proofs))
    }

```

```
  def create(sender: PrivateKeyAccount,
             contractId: ContractAccount,
             funcIdx: Short,
             data: Seq[DataEntry],
             attachment: Array[Byte],
             fee: Long,
             feeScale: Short,
             timestamp: Long): Either[ValidationError,
                 ExecuteContractFunctionTransaction] = for {
    unsigned <- createWithProof(contractId, funcIdx, data, attachment, fee, feeScale,
        timestamp, Proofs.empty)
    proofs <- Proofs.create(List(EllipticCurve25519Proof.createProof(unsigned.toSign,
        sender).bytes))
    tx <- createWithProof(contractId, funcIdx, data, attachment, fee, feeScale,
        timestamp, proofs)
  } yield tx

  def create(sender: PublicKeyAccount,
             contractId: ContractAccount,
             funcIdx: Short,
             data: Seq[DataEntry],
             description: Array[Byte],
             fee: Long,
             feeScale: Short,
             timestamp: Long,
             signature: ByteStr): Either[ValidationError,
                 ExecuteContractFunctionTransaction] = for {
    proofs <- Proofs.create(List(EllipticCurve25519Proof.buildProof(sender, signature)
        .bytes))
    tx <- createWithProof(contractId, funcIdx, data, description, fee, feeScale,
        timestamp, proofs)
  } yield tx
}
```

RegisterContractTransaction.scala

```
package vsys.blockchain.transaction.contract

import com.google.common.primitives.{Bytes, Ints, Longs, Shorts}
import play.api.libs.json.{JsObject, Json}
import scorex.crypto.encode.Base58
import vsys.account._
import vsys.blockchain.contract.{Contract, DataEntry}
import vsys.blockchain.state.ByteStr
import vsys.blockchain.transaction.TransactionParser._
import vsys.blockchain.transaction._
import vsys.blockchain.transaction.proof._
import vsys.utils.serialization.{BytesSerializable, Deser}

import scala.util.{Failure, Success, Try}

case class RegisterContractTransaction private(contract: Contract,
                                               data: Seq[DataEntry],
                                               description: String,
                                               transactionFee: Long,
                                               feeScale: Short,
                                               timestamp: Long,
                                               proofs: Proofs) extends ProvenTransaction {

```

```
  val transactionType = TransactionType.RegisterContractTransaction

  lazy val contractId: ContractAccount = ContractAccount.fromId(id)

  lazy val toSign: Array[Byte] = Bytes.concat(
    Array(transactionType.id.toByte),
    BytesSerializable.arrayWithSize(contract.bytes.arr),
    Deser.serializeArray(DataEntry.serializeArrays(data)),
    Deser.serializeArray(Deser.serilizeString(description)),
    Longs.toByteArray(transactionFee),
    Shorts.toByteArray(feeScale),
    Longs.toByteArray(timestamp))

  override lazy val json: JsObject = jsonBase() ++ Json.obj(
    "contractId" -> contractId.address,
    "contract" -> Json.obj("languageCode" ->Deser.deserilizeString(contract.
        languageCode),
      "languageVersion" -> Ints.fromByteArray(contract.languageVersion),
      "triggers" -> contract.trigger.map(p => Base58.encode(p)),
      "descriptors" -> contract.descriptor.map(p => Base58.encode(p)),
      "stateVariables" -> contract.stateVar.map(p => Base58.encode(p)),
      "textual" -> Json.obj("triggers" -> Base58.encode(contract.textual.head),
        "descriptors" -> Base58.encode(contract.textual(1)),
        "stateVariables" -> Base58.encode(contract.textual.last))),
    "initData" -> Base58.encode(DataEntry.serializeArrays(data)),
    "description" -> description,
    "timestamp" -> timestamp
  )

  override lazy val bytes: Array[Byte] = Bytes.concat(toSign, proofs.bytes)

}

object RegisterContractTransaction extends TransactionParser {

  val MaxDescriptionSize = 140
  val MinDescriptionSize = 0

  def parseTail(bytes: Array[Byte]): Try[RegisterContractTransaction] = Try {
    val (contractBytes, contractEnd) = Deser.parseArraySize(bytes, 0)
    (for {
      contract <- Contract.fromBytes(contractBytes)
      (dataBytes, dataEnd) = Deser.parseArraySize(bytes, contractEnd)
      data <- DataEntry.parseArrays(dataBytes)
      (descriptionBytes, descriptionEnd) = Deser.parseArraySize(bytes, dataEnd)
      description = Deser.deserilizeString(descriptionBytes)
      fee = Longs.fromByteArray(bytes.slice(descriptionEnd, descriptionEnd + 8))
      feeScale = Shorts.fromByteArray(bytes.slice(descriptionEnd + 8, descriptionEnd +
          10))
      timestamp = Longs.fromByteArray(bytes.slice(descriptionEnd + 10, descriptionEnd
          + 18))
      proofs <- Proofs.fromBytes(bytes.slice(descriptionEnd + 18, bytes.length))
      tx <- RegisterContractTransaction.createWithProof(contract, data, description,
          fee, feeScale, timestamp, proofs)
    } yield tx).fold(left => Failure(new Exception(left.toString)), right => Success(
        right))
  }.flatten

```

```
77   def createWithProof(contract: Contract,
78                       data: Seq[DataEntry],
79                       description: String,
80                       fee: Long,
81                       feeScale: Short,
82                       timestamp: Long,
83                       proofs: Proofs): Either[ValidationError,
                             RegisterContractTransaction] =
84     if ((Deser.serilizeString(description).length > MaxDescriptionSize) || !Deser.
           validUTF8(description)) {
85       Left(ValidationError.InvalidUTF8String("contractDescription"))
86     } else if(fee <= 0) {
87       Left(ValidationError.InsufficientFee)
88     } else if (feeScale != DefaultFeeScale) {
89       Left(ValidationError.WrongFeeScale(feeScale))
90     } else {
91       Right(RegisterContractTransaction(contract, data, description, fee, feeScale,
             timestamp, proofs))
92     }
93
94   def create(sender: PrivateKeyAccount,
95              contract: Contract,
96              data: Seq[DataEntry],
97              description: String,
98              fee: Long,
99              feeScale: Short,
100             timestamp: Long): Either[ValidationError, RegisterContractTransaction] =
                    for {
101    unsigned <- createWithProof(contract, data, description, fee, feeScale, timestamp,
           Proofs.empty)
102    proofs <- Proofs.create(List(EllipticCurve25519Proof.createProof(unsigned.toSign,
           sender).bytes))
103    tx <- createWithProof(contract, data, description, fee, feeScale, timestamp,
           proofs)
104  } yield tx
105
106
107  def create(sender: PublicKeyAccount,
108             contract: Contract,
109             data: Seq[DataEntry],
110             description: String,
111             fee: Long,
112             feeScale: Short,
113             timestamp: Long,
114             signature: ByteStr): Either[ValidationError, RegisterContractTransaction]
                    = for {
115    proofs <- Proofs.create(List(EllipticCurve25519Proof.buildProof(sender, signature)
           .bytes))
116    tx <- createWithProof(contract, data, description, fee, feeScale, timestamp,
           proofs)
117  } yield tx
118 }
```

CERTIK

Building Fully Trustworthy Smart Contracts and Blockchain Ecosystems