Modeling tokens on blockchains
Summary booklet

Tutors: Dany Moshkovich and Artem Barger

Students: David Valensi and Naomie Abecassis

1) Introduction
2) Part 1: Modeling simple Ownership asset and Rental asset (stand-alone implementation)
3) Part 2: Modeling static implementation of inter-token dependencies
    a) Rules of inter-dependencies
    b) Sequence diagram
    c) Class diagram
4) Part 3: Modeling dynamic implementation of inter-token dependencies using extensions
5) Installing the environment
6) Steps to run each part of the project
7) Explanations for developing new smart contract

1) Introduction

The goal of this project is to implement ownership and renter assets using blockchain ERC20 tokens.

ERC20 tokens are tokens following a list of standards developed by the Ethereum community. In these standards we can denote mandatories functions and optional functions. Each ERC20 token must implement the 6 following functions:

- totalSupply: returns the amount of token in existence
- balanceOf: returns the amount of tokens owned by a certain account
- transfer: moves a certain amount of tokens from the message sender account to a recipient account
- transferFrom: moves a certain amount of tokens from a sender account to a recipient account
- approve: Sets a certain amount as the allowance of a spender over the caller's tokens
- allowance: Returns the remaining number of tokens that a spender will be allowed to spend on behalf of an owner. This is zero by default.

So in each step of the project we are implementing all these function in the tokens.

An ownership asset token represents the owning of an object that we can give to someone else (by transfer) and each person that has the token must be able to show a proof of ownership.

A rental asset token represent the rent of an object. The rent is set for a certain amount of time, and during the validity of the rent each renter can give the token to someone else that has permission to rent the object. When starting a rent, the owner of the object will define a list of persons that are allowed to keep the rent and according to this list the token can be transferred.

As an example to illustrate this we can think of a car. The owner of the car can sell the car and thus give it to someone else, and he can also decide to rent the car to a group of person that can share the car: during the rent validity time, each person can give the car to someone else in the group.

The project is divided into 3 steps of implementation that are explained below. In each step of the project the contract that are implemented follow the standards of ERC20 token (meaning they are defined as being ERC20 tokens and they implement all the mandatory functions of the token).

2) Part 1: Modeling simple Ownership asset and Rental asset (stand-alone implementation)

First, we implement the tokens of ownership asset and rental asset without linking them. Each owner can create an ownership token and a rental token independently. In this

step the fact that someone owns an object and someone else rent it are completely independent and have no influence one on the other.

The implementation of this step can be found under the folder Step1, where the ownership asset is implemented by the contract called OwnershipToken and the rent asset is implemented by the contract called RentalToken.

To this implementation we join a script running a test on the tokens in the file TestTokens.js under the folder Test.

We can illustrate the implementation of this part with a sequence diagram explaining the different operations on the tokens:

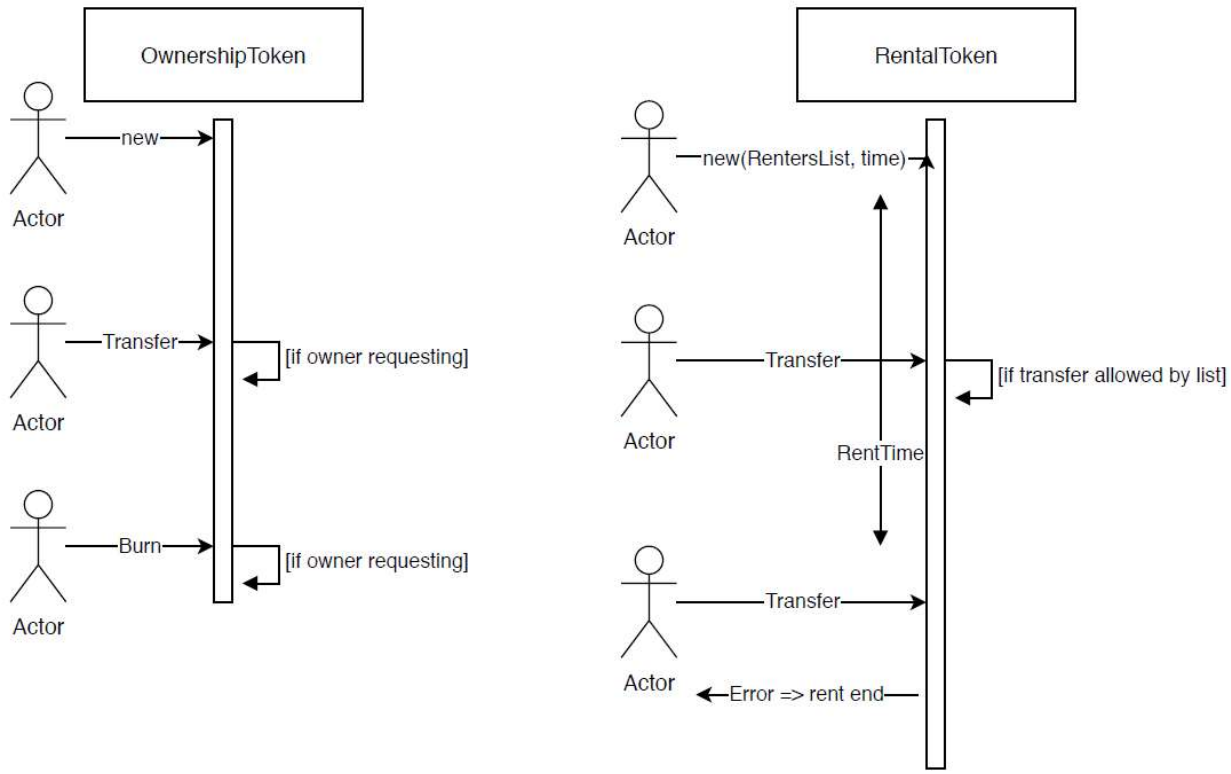

Figure 1. Sequence diagram illustrating part1

3) Part 2: Modeling static implementation of inter-token dependencies

The second part of the project consists on implementing inter-token dependencies. Unlike part 1, now the tokens are 'aware' one of the other and they are linked together. An owner of an object will create a SmartOwnershipToken to represent this ownership asset and then if he wish to start a rent he will use the function startRent of this token.

This function will create a SmartRentalToken which will be linked to the SmartOwnershipToken.

a) Rules of inter-dependencies

The fact that now the tokens are dependent one of the other allows us to impose some rules that a contract has to respect:

- A rent can be allowed only if it is created from an owner (meaning someone already has owning on the object) **//LIMITATION-MISSING**
- Only the owner of an object can decide to start a rent
- At starting of the rent, the owner must provide a list of addresses that are included in the rent (then each address will be able to keep the rent)
- A rent is defined within a period of time: this time is calculated in minutes and starts when the owner uses the function startRent. After this time is finished the rent cannot be transferred anymore and it will go back to the owner automatically. During the time the rent is valid, a renter can give the token to someone allowed by the list of renters.
- If an object is rented (the rent is still valid according to the time) it is forbidden to give the object (the ownership of the object)
- Even an owner is not allowed to take back the rent before the time of the rent ends (the only way for the owner to get the rent back during this time is if the actual renter transfers him the rent)
- An owner can decide to burn a rental token after the time allowed to the rent is over.

This part of the project is more realistic than the first part since it reflects the reality in a better way.

b) Sequence diagram

In this part, someone wishing to create a rent for an object will first create a SmartOwnershipToken for this object, then assign to him a rent using the function startRent which will create a new SmartRentalToken. This function get as parameters a list of addresses that are allowed in the rent and a time defining the rent. Once the rent is set different accounts can freely interact with the SmartRentalToken following the rules defined above. We can illustrate the activity flow of this part with a sequence diagram explaining the different operations on the tokens:

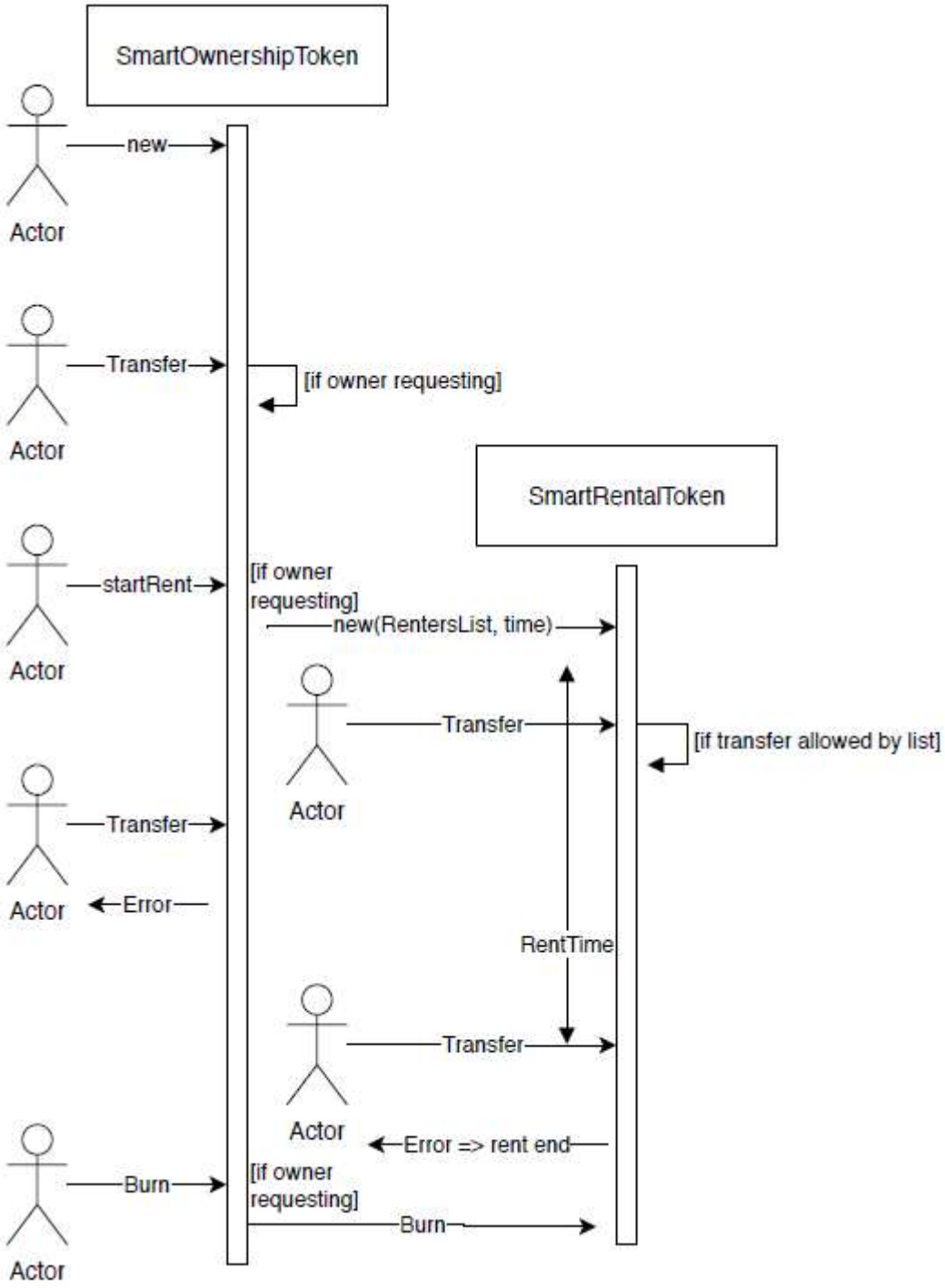

Figure 2. Sequence diagram illustrating part2

c) Class diagram

In this part the dependencies between the two tokens are implemented by using association: the SmartOwnerToken stores in his parameters the address of the SmartRentalToken and in the same way the SmartRentalToken also stores the address of the SmartOwnershipToken. We can illustrate the mechanism by a class diagram:

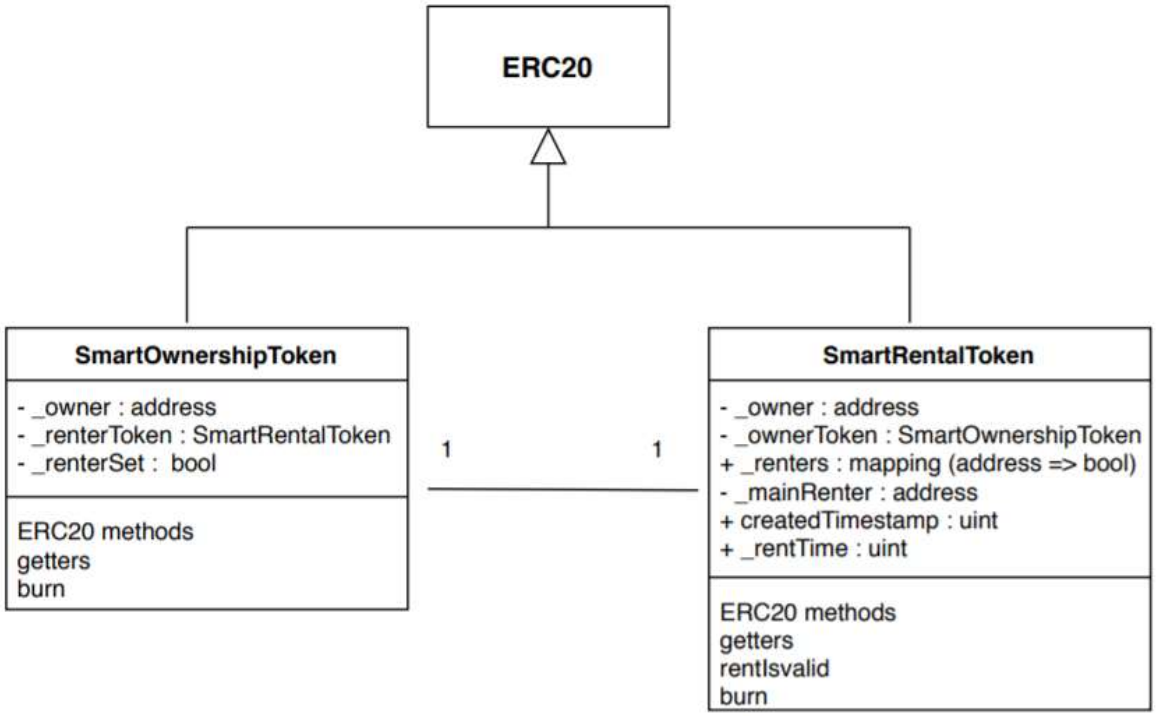

Figure 3. Class diagram illustrating part2

However, this implementation has limitations since in the reality we don't obviously want to allow the rent of an object. This explains the need of part 3, in which we want to change the inter-token dependency to be dynamic and not static such as this part.

4) Part 3: Modeling dynamic implementation of inter-token dependencies using extensions

Finally, in this part we wish to implement a dynamic inter-token dependency.

a) What is dynamic inter-token dependency

We want to implement a mechanism that will allow us to add functionalities to an OwnershipToken. This mechanism includes two main behaviors:

- In the reality, we don't always want to have rent allowed on an object, we can have an object that is not rentable (but is only sellable). In order to solve this problem, in this part we change the rent mechanism to be an extension of the ownership. At the beginning an ownership token is only sellable and there is no

option of rent on him, and if we want to add rent to it we add an extension to it. An extension is a contract of type Extension that implements functions that are in charge of creating a RentalToken and handling the rent. In the same way we can define more options on an OwnershipToken.

- As we explained above, the ownership asset is implemented by an ERC20 token which follows some rules, including functions as transfer that are mandatory. What we wish to do in this part is to be able to add preconditions and postconditions to each action that is done on the token. These pre/post condition will be defined in the extension file (as function) and then when we'll add the extension to the OwnershipToken it will automatically add the running of these functions before/after the relevant action.

Each extension in this part is created at the running time, so we can have a situation where a OwnershipToken exists and then we write the code of an extension compile it and then we add this extension to the OwnershipToken, without needing a rebuild or a recompilation of the OwnershipToken.

b) Developer manual for this mechanism

Let's explain how works this mechanism. First, we define a contract named ExtensionInfo that contains the same parameters as DynamicOwnership and parameters that store informations about an extension. These information are the number of functions contained in the extension and for each function a struct Extension is defined (containing the name of the function that is being extended, the signature of the function extending and the type of extension). There are 3 types of extensions:

- Precondition: the function is linked with a certain ERC20 method and will be called at the beginning of the method. The requirement is that this pre-condition passes in order to be able to compute the ERC20 method.
- Postcondition: the function is linked with a certain ERC20 method and will be called at the end of the method.
- Invocation: the function can be invoked by a DynamicOwnership if the corresponding extension has been added to the token.

To create a new extension you must create a new smart contract that inherits from ExtensionInfo. In this contract you will add the code of the functions you wish to add dynamically. Then for each function like that, you must add it in the constructor as an Extension struct with his parameters (defining the function extended, the signature and the type of extension) and you must increase the number of extensions by 1. When creating a new extension it is important that you don't add any parameters since this will change the context and the mechanism of invoking function will not work fine.

Let's illustrate this with a scheme:

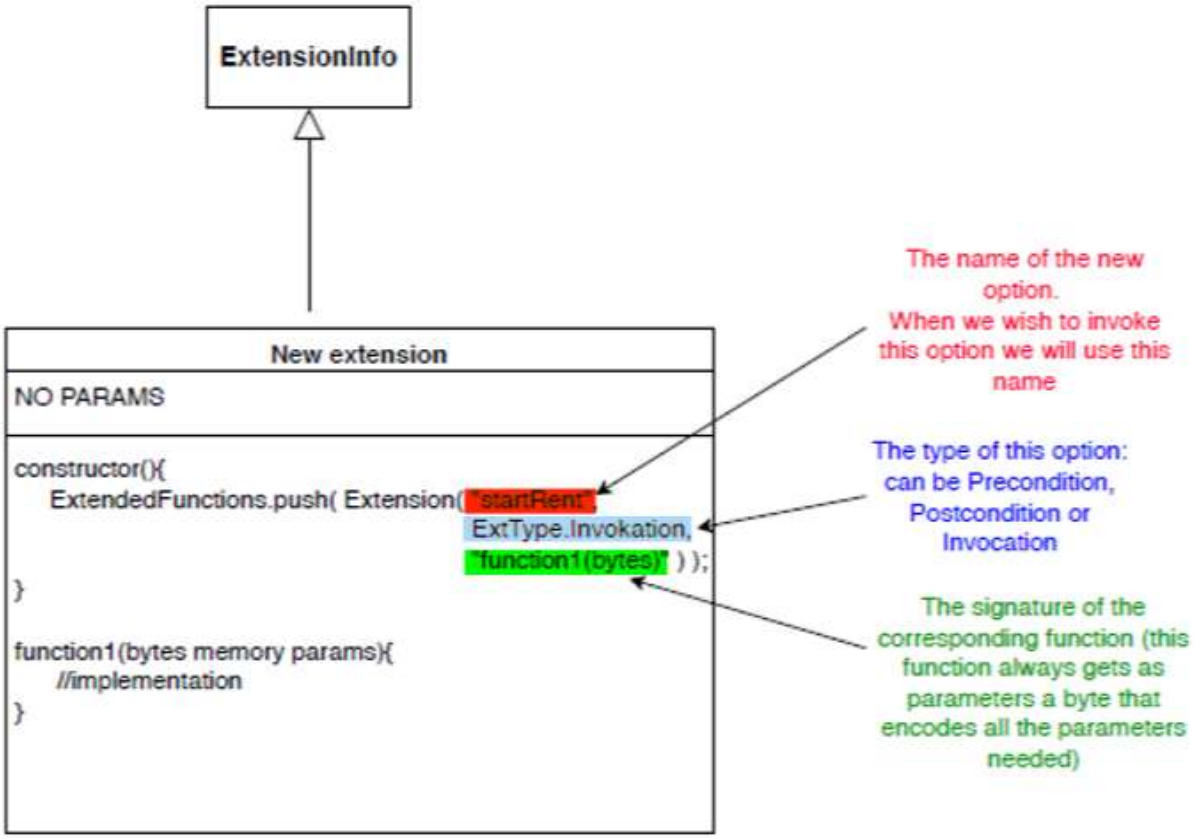

Figure 4. Illustration for adding a new option to an extension

For each pre-condition or post-condition we wish to run before/after an ERC20 method we define a function in the extension and we add it in the constructor in the same way as invocation. The following scheme illustrates a pre-condition to the function transfer:

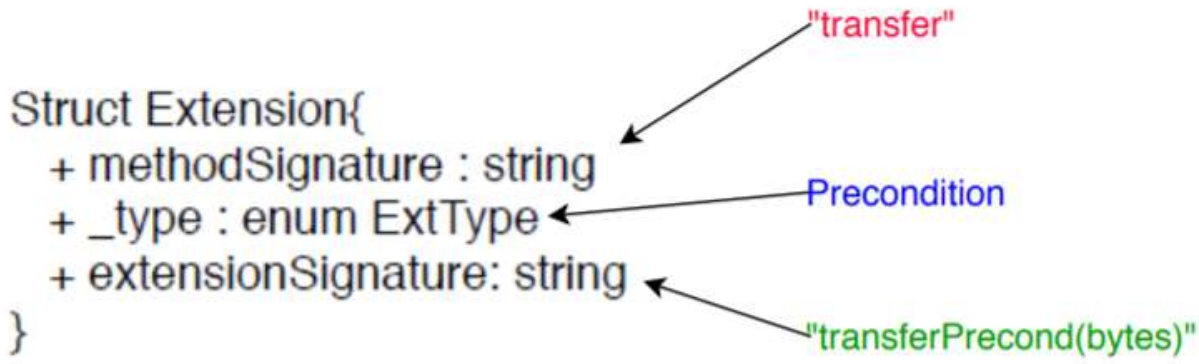

Figure 5. Illustration for adding a new precondition to transfer function

The implementation of the dynamic inter-token dependencies allows an extension to store data in the DynamicOwnership memory: the data is stored in the mapping extensionsData. In order to keep data coherency we define the following convention: data named b that is used by the extension A will be stored in extensionsData[“A_b”].

This allows any extension to use the data of an DynamicOwnership token when being invoked. For example the data renterSet used by the extension Rent will be stored in extensionsData[“Rent_renterSet”]. The data is stored in an encoded form of bytes.

Moreover each function thas is of type Invocation can receive parameters that are encoded as bytes. When adding a function to an extension we define as a convention that the function gets only one parameter that is of type bytes. Thus any function in an extension must have a signature as: *function_name*(bytes memory params). We make a difference between:

- Precondition/PostCondition functions: the byte received as a parameters will always contains the parameters passed to the extended method. For example if the function transferPreCond(bytes memory params) is extending the transfer ERC20 method, params will contain the encoded version of the parameters received by the transfer ERC20 method (which are (address recipient, uint amount)).
- Invocation functions: each developer defines which parameters an invocation function should get. These parameters are received in an encoded form (in the parameter params of type bytes), and the developer must decode params in order to get the actual values of the parameters. For example if you wish to write a function in the extensions that will receive as first parameter an address and as second parameter a uint, you must define:
  (address param1, uint param2) = abi.decode(params, (address, uint));
  And then you can use param1 and param2.
  To follow this implementation, when invoking an extension you must pass this bytes parameter as you defined the parameters of the function. If the parameters encoded are not corresponding to the parameters defined by the function the behavior of the function is not defined.

We can illustrate the implementation of the mechanism by the following class diagram:

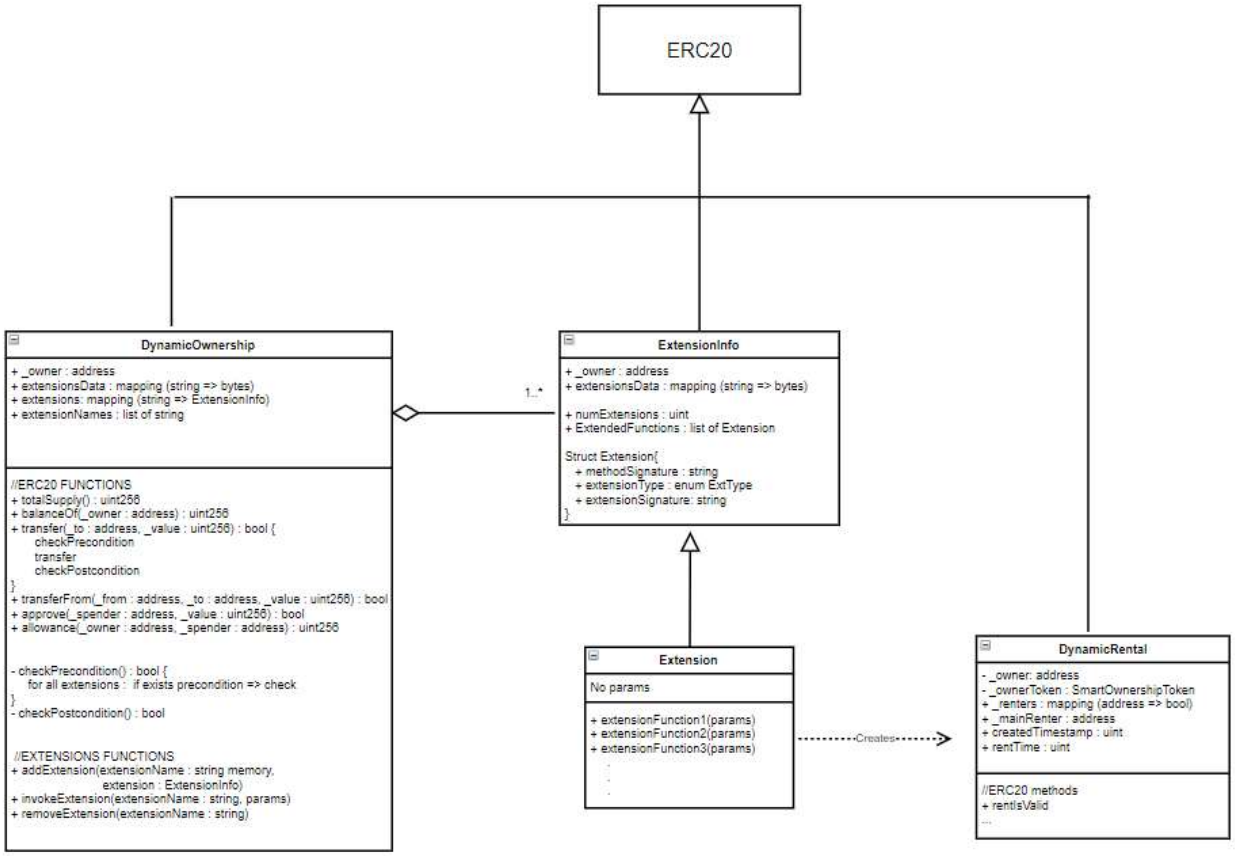

Figure 6. Class diagram illustrating part3

c) User manual for this mechanism

Once all the code is written in a good way we can describe the general flow of the activities in order to add an extension. First you need to create an instance of a DynamicOwnership token, and an instance of the extension you prepared. Then, use the function addExtension of the DynamicOwnership token to add the current extension to the token. This function gets as parameters the name of the extension (this name will be used later to invoke options) and the address of the extension that you wish to add. Once the extension is added you can invoke functions of the extension using the function invokeExtension. This function gets as parameters the name of the extension where the function can be find, the signature of the function you wish to invoke and the bytecode of the params you wish to pass to the function. We define that if the function doesn't exist in any extension added to the token, the invokeExtension function will return false.

In the same way, once the extension has been added to the token, when the token will be requested to run an ERC20 method, it will first check all pre-condition functions of the extension (corresponding to this ERC20 method), then compute the core of the

method, and then run the post-condition functions of the extension (corresponding to this ERC20 method).

We can describe this flow of activity with a sequence diagram:

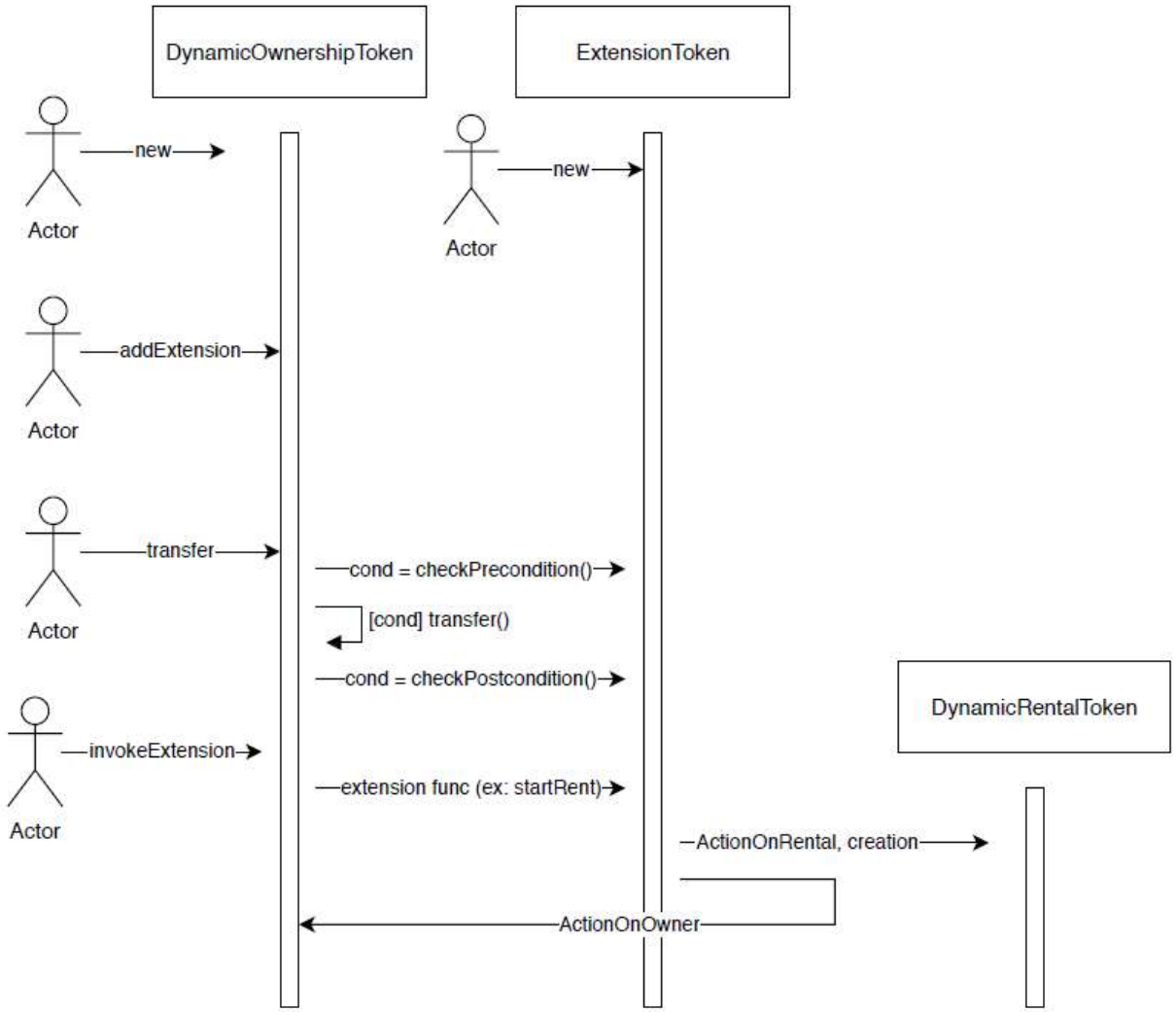

Figure 7. Sequence diagram illustrating part3

As explained above each function of an extension gets a parameter of type bytes that contains an encoding of the parameters of the function. For example, if the function is defined to receive as parameters an address (denoted parameter1) and a uint (denoted parameter2) in order to invoke the function we need to pass the argument:

abi.encode(parameter1, parameter2)

Then in the function a mechanism is implemented in order to unpack these arguments.

Moreover as defined above each extension can add data to the DynamicOwnership and this is done by storing this data in the mapping extensionsData. As a convention we

define that the data named b added by an extension named A is stored in extensionsData[“A_b”]. For example the variable named renterSet defined by Extension will be stored in extensionsData[“Extension_renterSet”]. In order to get access to this data you can use the DynamicOwnership method getMapElement which get as a parameter the name of the data as defined by the convention (as a string) and returns the encoded data stored. In order to decode the data you must use the function abi.decodeParameters.

5) Installing the environment

1) Download Nodejs (this will install npm too) with the following link https://nodejs.org/en/ and install it.

(After installing it you can check the installation version by writing the command "node -v" in the command prompt and "nmp -v")

2) Install the IDE: we choose to work with Visual Studio since it is convenient to developp smart contracts.

- You can download and install it from https://code.visualstudio.com/

- Install Visual Studio extension: go to the extensions section and install these plugins

  - Solidity
  - Material Icon Theme

- Enable icon theme: select File -> Preferences -> Fill icon theme

3) Install truffle : npm install -g truffle (in the regular command prompt)

4) Install ganache, an application that is used in order to test the different contracts implemented along the project. Ganache gives us the ability to fire up a personal Ethereum blockchain that is used to run tests. You can download it with the following link: https://www.trufflesuite.com/ganache

6) Steps to run each part of the project

- Create a new directory => mkdir step1
- In this directory run the following commands (in the command prompt):
  - truffle init
  - npm init -y
  - npm install -E openzeppelin-solidity
  - npm install truffle-assertions
  - npm install -g ganache-cli
- Now the current directory has several folders: contracts, migrations, test and node_modules:
  - contracts: directory to store the smart contracts that are created
  - migrations: directory for deploying the smart contracts into the blockchain

- tests: directory for testing the smart contracts
- node_modules: directory that contains the library that we need (erc20…)

- Add in each directory the files that you can find under the same directory in the Step1 folder
- Change the content of the file truffle-config.js with the one on the git repository (this file must be located in the directory step1)
- In order to open visual studio, in the directory step1, run the following command: code .
- In order to run the test of the contracts run in the terminal the following command: truffle test --network ganache
  (important: we run the test using the ganache network since we are using the accounts provided by ganache in the test. So it is important before running the test that you open ganache application an start a new Ethereum workspace)
- If you wish to test the contracts on you own you can:
  - First migrate the contracts with the command: truffle migrate --compile-all –reset
  - Then open truffle console with the command: truffle console --network ganache

7) Explanations for developing new smart contract

If you wish to extend the implementations that was done in this project and create new contracts (for example new extensions for the third part), here are the steps to follow in order to add a new contract to the blockchain:

1) Add the code of your smart contract under the directory contracts/ let's say for the simple example NewExtension.sol

2) Add the test for NewExtension under the test/ directory.

3) Add a migration file that will deploy the NewExtension, named 2_deploy_contracts.js and define there the deployment of the NewExtension (the index is important because it defines in which order the compiler should look at the files)

4) Run the test with the command "truffle test" (in the command prompt). This command will compile the files and run the tests that are written for the NewExtension

5) Now you can deploy the smart contracts using ganache. First install ganache with the following command:

npm install -g ganache-cli

6) Open a truffle console with the command

truffle develop

7) For convenience you can open a second tab where you can see the log of the transactions. Run the command

```
truffle develop --log
```

8) In the first tab use the command "migrate" to deploy the contracts

9) You can see the transactions cache, the contract address, the account that it came from, the balance of that account...