DePool Phase 2 report

1. Common part

a. Overview

The activity described and discussed at the present report is a continuation of initial preparations for DePool provided described in the Phase-1 specification document (see Submission 2). While Phase-1 was concentrated on descriptions in the form of natural language and different kinds of diagrams the most important part of the current Phase-2 is a code written in Coq language while the present report limits its value to supporting documentation.

The main goal of Phase-2 is to prepare the specification in the form of the formal computer language (such as Coq) as well as to provide proofs at functional level leaving all the cross-functional (business) level proofs to Phase-3.

The present report describes and discusses the complete workflow of the Phase-2 providing code examples as illustrations, however it's just an auxiliary part of the Phase-2 keeping the **codebase** provided as the main part of the delivery.

b. Code acquiring and usage

The code developed for Phase-2 may be found and acquired from the following three repositories: https://github.com/Pruvendo/depool_contract.git , https://github.com/Pruvendo/depool_contract_scenarios.git and the binary submodule https://github.com/Pruvendo/coq-finproof-base.git. The code was developed and tested under Ubuntu and MacOS, other operating systems such as Windows were not tested and ability to be compiled under those OS is not guaranteed.

As a prerequisite **Coq 8.12.0 must be installed** and properly configured. To build the project:

- Clone both source repositories
- Enter the modules subdirectory of each repository and clone the binary submodule
- At the root dir of each repository just type *make*. Please allow 15-30 minutes for the scenario repository and a few hours for the main repository to build
- Proofs have its own Makefile and should be build separately or using `make proofs` from the root

c. Underlying technologies

The proposed solution is based on *Coq Proof Assistant*[1]. This tool is primarily designed to make an environment for proving mathematical theorems and for this purpose it provides

[1] https://coq.inria.fr/

OCaml[2]-like language (named *Gallina*) for specifying the entities to be proved, specific language for proves (that is roughly the sequence of so called tactics (that stand for a step in terms of traditional proving[3]) as well as a specific language *Itac* for defining custom tactics.

In addition *Coq* provides its own comprehensive IDE as well as API to be integrated with other development environments such as *Microsoft Visual Studio Code*.

Coq itself is based on a pure mathematical paradigm called *Calculus of constructions*[4] (and its extension called *Calculus of Inductive Constructions*[5]) that allows to use mathematical induction[6] in addition to pure formal logic[7].

Coq was initially introduced in 1989, was dramatically developed since that time, used for many theoretical and practical applications and, as an outcome, the authors of the present document suggest to consider it as a reliable tool that means:

- If *Coq* states that some statement is proved it's considered as proved
- At the same time *Coq* may have any number of bugs not related to the statement written above

According to *Curry–Howard isomorphism*[8] all the mathematical proofs can be applied to the computer programs that is essential for the approach presented in the current document. Thus this proof assistant may be applied to the computer programs using the approaches described in the following sections.

d. Basic principles

The basic principles used to verify smart contracts implemented using imperative languages on Coq are:

- imperative languages eDSL implemented on pure functional languages using state monads[9] as it's a conventional way to represent imperative sequences in functional environment
- Types of imperative languages are represented not by specific Coq types but rather by type classes (isotypes) that correspond to some preconditions that allows to select a wide range of instantiations to, for example:
    - Get a basic proof environment using Coq *Z* type
    - Get an advanced proof environment using native TVM types
    - Prepare the code for extraction into Haskell or other general purpose languages

---

[2] https://ocaml.org/
[3] For example, ***apply*** tactic roughly means usage of some already known theorem or ***symmetry*** tactic utilizes the axiom that ***a=b*** is equivalent to ***b=a***
[4] https://hal.inria.fr/file/index/docid/76024/filename/RR-0530.pdf
[5] https://coq.inria.fr/distrib/current/refman/language/cic.html
[6] https://encyclopediaofmath.org/index.php?title=Mathematical_induction
[7] http://www.collegepublications.co.uk/logic/mlf/?00029
[8] Howard, William A. (September 1980), "The formulae-as-types notion of construction", in Seldin, Jonathan P.; Hindley, J. Roger (eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, pp. 479–490, ISBN 978-0-12-349050-6.
[9] https://wiki.haskell.org/State_Monad

- Actively use custom tactics to identify areas that can be selected for autoproving in future
- For the purposes of verification of the present contract isotyping to classic Coq *Z*, *bool* and similar types was selected. Maps are isotyped to list of key-value pairs rather than to Patricia tree as for the original TVM. Using of *Z* type allows to apply such powerful tactics as *lia* or *psatz*.

e. Preassumptions and limitations

The following presumptions are suggested for the verification of the present smart contract:

- Blockchain and TVM work strictly according to the specifications mentioned above
- Coq Proof Assistant or any other tool being used works correctly
- If something is not specified the assumptions based on common sense logic are applied
- Infinite sequences of similar elements (arrays) may exist
- Each cycle has fixed precalculated number of loops
- Elector system-level smart contract works correctly
- No inline recursions allowed
- Each recursion (direct or mutual) has fix precalculated number of reentrances

f. Deep vs. shallow embedding

There are two main approaches implementing DSL: deep and shallow embedding[10]. The former technology keeps all the domain-specific elements as abstract leaving the implementation to the external “observer” while the latter one immediately unfolds domain-specific parts and then operates with them as with simple values.

Generally speaking each of these technologies has some advantages and disadvantages. However, for the isotyping approach described above deep embedding looks more applicable as it keeps all the semantics abstract before the specific isotyping instance is selected. As a result deep embedding was selected for Pruvendo technology.

g. Syntactically equivalent programs

Ideally embedded DSL should achieve exactly the same syntax as a standalone program. However, practically achieving such an exact identity may be rather difficult or even impossible (due to syntax restrictions of the GPL where DSL is embedded). In this case it’s possible to introduce such relationships as syntactically equivalent programs. Roughly speaking, the programs are syntactically equivalent if:

- They can be translated into each other without losing any data (but comments , spaces, tabulation symbols etc.)
- At least one direction of translation may be done by “simple” tools such as regular expressions to illustrate that the generic syntax structure remains unchanged

[10] https://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/embedding-short.pdf

- The both programs should look close to each other and have the similar structure from human point of view

The approach selected for the contract being verified uses Coq eDSL code that is syntactically equivalent to the original Solidity source. All the details are discussed in the corresponding section in the Specification part.

2. Specification part

a. General notes

The present part is intended to provide formal specifications and function-level proofs for the *DePool* contract. The description of the *DePool* contract as well as a high level specification in the natural language provided in the Phase-1 specification document (see Submission 2).

All the issues that are not covered in the present document are to be discussed in the Phase-3 document to be developed within the incoming weeks.

All the specifications and proofs are based on the *DePool* commit *94bff38f9826a19a8ae55d5b48528912f21b3919* from *12/07/20*.

b. Code structure

In the present section the key files and directories mentioned and briefly described (most of them are thoroughly discussed in the following sections):

File/Directory	Brief description
Main repository	
src/DePoolClass.v	Ledger definition
src/DePoolFunc.v	Translation of all the Solidity functions into Coq DSL
src/SolidityNotations.vo	Notations used by DSL (binary only)
src/Lib/Proofs	*eval* and *exec* functions with corresponding proofs as described in the corresponding section
src/Scenarios/Scenario.v*	direct scenarios
reverse_translator/	*Script of the reverse "translator" (in Python) as well as results of the translation*
Scenario repository	
src/Scenarios/Projections	states, conditions and moves as described in the corresponding section (related to

	projection approach)
src/Scenarios/Correctness	correctness predicates and corresponding proofs
src/Scenarios/Common	common complex predicates with some corresponding proofs

c. DePool ledger

Ledger represents the full state of the system. Physically Ledger is a record of *LedgerC* type class that contains the following fields (some obsolete and deprecated fields are not mentioned):

№	Field	Description
2	`Ledger_ι_ValidatorBase`	*ValidatorBase* supercontract
3	`Ledger_ι_ProxyBase`	*ProxyBase* supercontract
5	`Ledger_ι_ParticipantBase`	*ParticinantBase supercontract*
7	`Ledger_ι_Errors`	“Named” errors such as `Errors_ι_IS_NOT_PROXY`
8	`Ledger_ι_InternalErrors`	“Unnamed” errors referenced by their index such as `InternalErrors_ι_ERROR511`
9	`Ledger_ι_DePoolLib`	*DePoolLib* supercontract
12	`Ledger_ι_DePoolContract`	*DePool* contract described by the present Ledger
16	`Ledger_ι_VMState`	Internal TVM variables
17	`Ledger_ι_LocalState`	Local variables state

Each of these fields is also a record and their fields may be accessed the shortcut as this: `↑12 ε DePoolContract_ι_m_poolClosed` . The example above gives access to the *m_poolClosed* field of the *DePool* contract, *12* is a number of the corresponding field in the table above and a construction `↑12 ε` states for monadic lifting of the underlying subcontract. It's worth mentioning that the field accessed in such a way is still wrapped into a monad. To unwrap it the following construction has to be used:

```
eval_state (↑12 ε DePoolContract_ι_m_poolClosed) l
```

The full description of Ledger is provided in the *src/DePoolClass.v* file.

Many underlying elements of the Ledger may be accessed through the getters defined in *src/DePoolFunc.v* file and discussed in the corresponding section. Roughly speaking they repeat corresponding getters in the Solidity source and may be used after the proving. For example, `ConfigParamsBase_Φ_getCurValidatorData` returns information about the current validator (wrapped inside a monad).

Another important way to alter fields is to use *with* and *With* notations as in the example below:

```
$ r2 with
    RoundsBase_ι_Round_ι_handledStakesAndRewards :=
    _returnOrReinvestNonValidatorRound l
       _returnOrReinvestForParticipant_round2_handledStakesAndRewards
r2 $}
    (RoundsBase_ι_Round_ι_handledStakesAndRewards r2)
    $}
```

or

```
{$ l With VMState_ι_messages := newMessage :: oldMessages $}
```

In the former example the value of the *RoundsBase_ι_Round_ι_handledStakesAndRewards* altered while usage of *With* notation allows to modify a field hidden inside a tree of records.

d. Specific Coq notations

To achieve maximum similarity between the contract code written in Coq DSL and Solidity a number of Coq notations applied. For the for simple local variable assignment the following notation is used:

```
Notation " 'U0!' l ':=' x ; t" := (do l ← x; t) (at level 33, right
associativity, t at level 50): solidity_scope.
```

Here `do l ← x ; t` is another notation that may be unfolded as `(x >>= fun l => t)` where `>>=` is a classic monadic operations (*bind*). As a result we get a pure functional construction on a rather imperative form such as:

```
U0! Л_timer := ↑ε6 DePoolHelper_ι_m_timer ;
```

In the expression above the local variable *timer* is assigned by the field *timer* of the *DePoolHelper* contract. Semantically this code corresponds to the following code in Solidity:

```
address timer = m_timer;
```

At the right side of the Coq statement above another example of notation was used - `↑ε6` that lifts subcontract *№6* (*DePoolHelper*) and thus gives access to its fields.

Notations cover virtually all the syntaxic cases of Solidity. The following example shows quite a complex notation

```
Notation " lift 'U2!' f ^^ p '[[' i ']]' '!+=' x" := (do x' ← x; do i' ← i ; lift (do f' ← ε f; let p' := xIntPlus ((f' ->> p) [ i' ]) x' in modify (fun r => {$ r with f := {$ f' with p := (f' ->> p) [ i' ] ← p' $} $} )); void!) (at level 35, right associativity, f at level 50): solidity_scope.
```

The notation above states for `f.p[ i ] := x` Solidity expression.

Thus the set of notations allows to create DSL code that looks rather close (and human readable) to Solidity code. However, these codes are not identical to each other and this issue is discussed below.

The full list of notations is available in *src/SolidityNotations.v* and partially in *src/DePoolFunc.v* .

e. Translation from Solidity to Coq

As it was discussed above the usage of notations helps to create a Coq program that is syntactically equivalent (and semantically identical, with some limitations) to the corresponding Solidity program. To automate this task a special translator has been developed (not published at the current stage). The example of the translation is provided below:

Solidity source:

```
function checkPureDePoolBalance() private returns (bool) {
        uint stakes = totalParticipantFunds(0);
        uint64 msgValue = uint64(msg.value);
        uint sum = CRITICAL_THRESHOLD + stakes + msgValue;
        if (address(this).balance < sum) {
            uint replenishment = sum - address(this).balance;
            emit TooLowDePoolBalance(replenishment);
            return false;
        }
        return true;
}
```

Coq DSL equivalent:

```
Definition DePoolContract_Φ_checkPureDePoolBalance' : LedgerT (XValueValue XBool) :=
       U0! Л_stakes := DePoolContract_Φ_totalParticipantFunds (! $xInt0 !) ;
    U0! Л_msgValue := msg_value ;
```

```
      U0! Л_sum := ↑ε12 DePoolContract_ι_CRITICAL_THRESHOLD !+ $
Л_stakes !+ $ Л_msgValue;
    If! ( tvm_balance () ?< $ Л_sum ) then {
        U0! Л_replenishment := $ Л_sum !- tvm_balance ();
        (->emit  TooLowDePoolBalance (!! $ Л_replenishment !!)) >>
        $ (xError xBoolFalse)
    }; $ xBoolTrue.
```

As one can see Solidity and Coq DSL versions are not identical but close to each other from human point of view. To verify that they are syntactically equivalent the reverse “translation” is required as it was discussed in the Common part.

The process of the reverse “translation” is described in the following section while the ultimate result of the direct translation may be found in *src/DePoolFunc.v* file where all the sources of the smart contract being verified were translated.

f. Reverse “translation” from Coq to Solidity

As it was discussed before the reverse translation is needed to ensure that Solidity and Coq DSL code received after the direct translation are the same. At the same time the “reverse” translation must be easily understandable and must not contain any complicated transformations to be confident that the original syntax and grammar is kept. To achieve this goal the reverse translation was implemented exclusively via regular expressions. Example of such a regular expression (using Python syntax) is below:

```
sub(r'XHMap ([a-zA-Z0-9ι_]*) ([a-zA-Z0-9ι_]*)', r'mapping (\1 => \2)')
```

The example above translates *XHMap* Coq structure into Solidity *mapping*. The full set of the regular expressions is provided in the Appendix A as well as at */reverse_translator* directory of the main repository.

g. Eval/Exec functions

Each function when called does two things : modifies a Ledger and returns some value. Talking about return value it may be a regular value, *void*, exception (it’s considered as a return value) or ‘no exception’ entity. The latter two are described by *XErrorValue* type (that has two constructors - one for success (with some regular value as a parameter) and one for exception (with *ErrorsP* or *InternalErrorsP* type class as a parameter)) while *void* is described by special type *True* that has only one constructor without any parameters.

The return value of the function *f* may be calculated as `eval_state f l` (unwrapped from monad) while the modified Ledger may be received as `exec_state f l` (unwrapped from monad as well).

The function `eval_state f l` is called *eval* function while the function `exec_state f l` is called as *exec* function.

The internal structure of the *eval* and *exec* functions may be pretty much complicated so their direct usage may be rather difficult. The provided solution is to prove equivalence of the *eval* and *exec* functions to their manually created “twins” thus converting calculations into a formula. This process is thoroughly discussed in the next section.

h. Proves of Eval/Exec functions

The lemma of equivalence has been created for each *eval* and for each *exec* function. The example of such lemmas is provided below:

```
Lemma DePoolContract_Φ_withdrawAll_eval : forall (l : Ledger),
let sender := eval_state msg_sender l in
let isInternalMessage : bool := negb (sender =? 0) in
let     isPoolClosed     :     bool     :=     eval_state     (↑ε12
DePoolContract_ι_m_poolClosed) l in
let         optParticipant         :=         eval_state         (↓
ParticipantBase_Φ_fetchParticipant sender) l in
let isEmptyParticipant : bool := negb (isSome optParticipant) in

eval_state DePoolContract_Φ_withdrawAll' l =
if isInternalMessage then
    if isPoolClosed then Value (Error I) else
        if isEmptyParticipant then Value (Error I)
    else Value (Value I)
else  Error (eval_state ( ↑7 ε Errors_ι_IS_EXT_MSG) l).

Lemma DePoolContract_Φ_withdrawAll_exec : forall (l : Ledger),
let sender := eval_state msg_sender l in
let isInternalMessage : bool := negb (sender =? 0) in
let     isPoolClosed     :     bool     :=     eval_state     (↑ε12
DePoolContract_ι_m_poolClosed) l in
let         optParticipant         :=         eval_state         (↓
ParticipantBase_Φ_fetchParticipant sender) l in
let isEmptyParticipant : bool := negb (isSome optParticipant) in
let participant := maybeGet optParticipant in
let      newParticipant      :=      {$      participant      with
(DePoolLib_ι_Participant_ι_reinvest, false) $} in
let           l_set           :=           exec_state           (↓
ParticipantBase_Φ__setOrDeleteParticipant  sender  newParticipant)  l
in
let           l_send           :=           exec_state           (↓
DePoolContract_Φ_sendAcceptAndReturnChange) l_set in
```

```
let      statusDepoolClosed      :=      eval_state      (↑12      ε
DePoolContract_ι_STATUS_DEPOOL_CLOSED) l in
let      statusNoParticipant      :=      eval_state      (↑12      ε
DePoolContract_ι_STATUS_NO_PARTICIPANT) l in
exec_state DePoolContract_Φ_withdrawAll' l =

if isInternalMessage then
     if isPoolClosed then exec_state (↓ DePoolContract_Φ__sendError
statusDepoolClosed 0) l else
                      if   isEmptyParticipant   then   exec_state   (↓
DePoolContract_Φ__sendError statusNoParticipant 0) l
        else l_send
else l.
```

The strategy of proving such lemmas is based on the *compute* tactic that makes the deepest reduction of the term thus ideally automatically achieving the equivalence of the both terms. However, it virtually never happens in real life and requires as wide usage of auxiliary tactics as well as to perform state-of-the-art activity to make some terms opaque to avoid overflow or inacceptable long calculations.

All the *eval* and *exec* functions may be found in the *src/Proofs* directory. Please note that the full compilation of all the files from this directory may take up to a few hours.

i. Projection approach

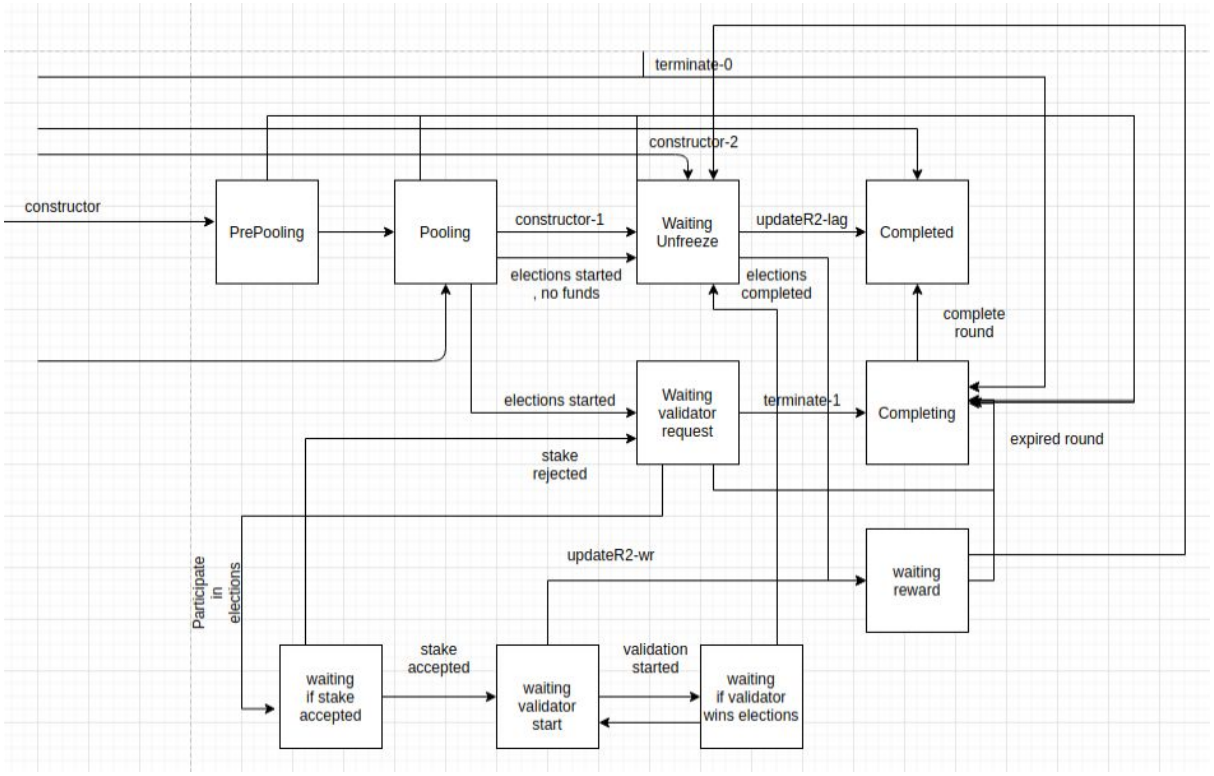
A finite state machine with (in some cases) temporal additions was selected as a basic tool for scenario building. However, the full set of states is too huge for any kind of handling so the projections of states to a rather small set of hypersurfaces is used for forming the scenarios. It's important to ensure that the selected set of hypersurfaces is full and covers all the dimensions of the original state. Thus the overall number of scenarios is a sum of scenarios for each hypersurface rather than multiplication as it would be for the complete state.

For each hypersurface the conventional way of finite state machine representation is used where squares illustrate the different state, arrows - possible transitions between states and titles for these arrows - conditions for transitions. It is worth noting that conditions have heterogeneous form and consist of an external event (in most cases) as well as a logical condition (also, in most cases).

Additionally some states have attributes (such as "balance") and its evolution during transitions may be represented as the second title for arrows.

Temporal epochs (for example, divided by a round switch) are separated by a vertical dashed line while the corresponding transitions of attributes are illustrated by a dotted arrow.

Example of such a projection (“round step” projection) is provided below while the full list of them is available in Appendix B:



The projection approach suggests a way to describe a complete set of business-level scenarios using a following methodology:

- Each way without loops from the beginning to any dead end is a scenario
- Each loop is a scenario itself, one round only

For a full list of scenarios please refer to the [Depool contract verification report (Phase-1)] (see Submission 2).

When the projections are created the next steps to prove the contract at the business level are:

- Formally specify conditions/moves (arrows in projections) (see below)
- Prove that all the scenarios are reachable (see below)
- Prove that no other ways to change the state of each projection is possible (Phase 3)
- Define correct state (see below)
- Prove that any move from correct state goes into correct state (see below)
- Prove that the program can not stuck in any state but the terminal one

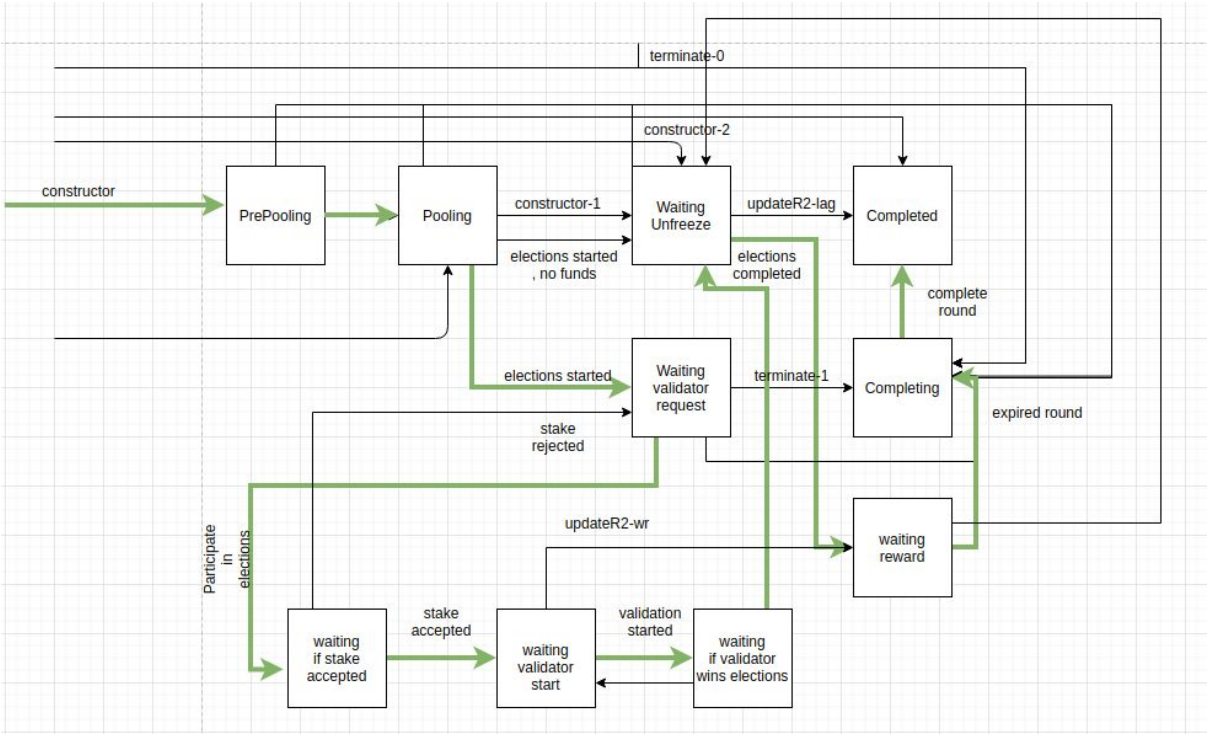
j. Direct scenarios

For the subset of scenarios discussed in the previous section the formal definitions were implemented, the list of such scenarios is provided below:

- Scenario 1: victory scenario, ordinary stake from validator that is the only participant, one full round loop (4 round jumps) after constructor
- Scenario 2: defeat scenario, ordinary stake from validator that is the only participant, one full round loop after constructor

- Scenario 3: victory scenario, ordinary and vesting stake from validator that is the only participant, one full round loop after constructor
- Scenario 4: defeat scenario, ordinary and vesting stake from validator that is the only participant, one full round loop after constructor
- Scenario 5: victory scenario, ordinary stake from validator + ordinary stake from participant, one full round loop after constructor
- Scenario 6: defeat scenario, ordinary stake from validator + ordinary stake from participant, one full round loop after constructor
- Scenario 7: victory scenario, ordinary stake from validator + vesting stake from participant, one full round loop after constructor
- Scenario 8: defeat scenario, ordinary stake from validator + vesting stake from participant, one full round loop after constructor
- Scenario 9: too low ordinary stake from validator + ordinary stake from participant, one full round loop after constructor

The described scenarios roughly follow the paradigm described above, as an example Scenario 1 may be illustrated by the following diagram:



Examples of the formal definitions of the scenarios can not be included into the present document because all them are too large but all the scenarios may be found at the root directory under *src/Scenarios*.

k. Formal specification for condition/moves

State - all the possible values of the projections is a set that is a union of states. States don't intersect with each other. Usually state is determined by the value of one of the variables

with a finite possible number of values (not only finite but also rather small, less than 15 or so). At the diagrams changes are drawn as rectangles.
Conditions - are conditions that have to be met to change the projection (with or without changing the *state*). At the diagrams changes sometimes drawn as titles for the arrows (moves)
Moves - changes of the projection that happen when the certain *condition* is met. At the diagrams moves are drawn as arrows.

All the *states*, *conditions* and *moves* are located at the *src/Projections* directory. Each projection has its own subdirectory. *States* are usually located in the *States.v* file while *conditions* and *moves* may be found in *Conditions.v* . Below the examples of formal specifications of *states*, *conditions* and *moves* are provided:

```
Definition     getProjectionRoundCompletionReasonState     (r     :
RoundsBase_ι_Round) :=
    RoundsBase_ι_Round_ι_completionReason r.
```

It's a quite typical definition of *state* - with simple binding of state to the value of the specific variable. Now let's move to the example of condition:

```
Definition
projection_round_steps_waiting_validator_request_waiting_if_stake_ac
cepted_condition
    (l : Ledger) (r : RoundsBase_ι_Round) :=
    participateInElectionsCalled /\
    onlyValidatorContract l /\
    poolClosed l = false /\
    checkDePoolBalance l (msgValue l) (balance l) /\
    stakeAt = RoundsBase_ι_Round_ι_supposedElectedAt r /\
    r = round1 l /\
                      getProjectionRoundStepState r =
RoundsBase_ι_RoundStepP_ι_WaitingValidatorRequest.
```

It should be noted that *condition* typically is a conjunction of a few predicates that can be roughly splitted into three types:

- simple predicates - such as `poolClosed l = false`
- complex predicates - such as *checkDePoolBalance*. Such predicates may have pretty much complicated internal structure. For example, *checkDePoolBalance* discussed here is defined by the following expression:

```
Definition checkDePoolBalance (l : Ledger)(msgValue balance : Z) :=
     (CRITICAL_THRESHOLD l + totalParticipantsStake l + msgValue <=
balance)%Z.
```

At the same time *totalParticipantStake* is also a complex predicate that in its turn contains a complex predicate *totalStake*.

- caller predicates - equal to *True* when and only when the calling message (internal or external) equals to some function. As an example *participateInElectionsCalled* predicate equals to *True* when *participateInElection* message is being handled. During Phase-3 such predicates will be defined through the *VMState_ι_messages* field of the *Ledger_ι_VMState* (field №16) of the Ledger.

Taking into account complex predicates may be quite complicated and consequently contain some errors proving that they are reasonable and non-contradictory becomes necessary. Example of such a lemma is provided below:

```
Lemma      cutWithdrawalValueMove_lastWithdrawalTime_strictly_later:
forall l i,
    investParamsCorrectLocally l i ->
    0 < withdrawal l i ->
             RoundsBase_ι_InvestParams_ι_lastWithdrawalTime   i   <
newLastWithdrawalTime l i.
```

The lemma above states that under certain circumstances *lastWithdrawalTime* of *InvestParams* object is increased after each partial withdrawal that is definitely reasonable. Such proofs and internal complex predicates are mostly located in *src/Scenarios/Common* directory.

Specification of *moves* is very similar to the specification of *conditions*. The typical example is below:

```
Definition
projection_round_completion_reason_validator_stake_is_too_small_move
    (ol nl : Ledger) (or nr : RoundsBase_ι_Round) :=

projection_round_completion_reason_validator_stake_is_too_small_cond
ition ol or /\
    roundIn nl nr /\
    roundsSame or nr /\
                getProjectionRoundCompletionReasonState    nr    =
RoundsBase_ι_CompletionReasonP_ι_ValidatorStakeIsTooSmall.
```

It worth mentioning that the typical *move* predicate consists on:

- *condition* predicate
- *move* itself - changes in the projection
- additional requirements - in this particular case they are:
    - round should stay in the ledger (be one of four active rounds)
    - modified round should have the same id as the original one (roughly speaking, should be the “same” round as long as “same” is applicable for immutable objects of functional languages)

I. Invariants and correctness

Many popular technologies of formal verification use invariants as a key entity they build their approach around. It's not a case for Pruvendo approach, at least, explicitly but invariants, in the form of conjunction of predicates that must be always equal to *True*, still are very important as it's necessary to verify that any *move* (described above) from the correct state comes into the correct state.

The correctness predicate is a tree of predicates (mostly joined by conjunction). Such a root predicate looks as follows:

```
Definition ledgerCorrectGlobally (l : Ledger) :=
    ledgerCorrectLocally l /\
    (forall r, roundIn l r -> roundCorrectGlobally l) /\
    (forall p, participantIn l p -> participantCorrectGlobally l).
```

At this point it's worth mentioning that correctness may be local and global. Local correctness must be kept at any point of execution while global correctness must be kept only before and after handling of any message and may be violated inside the handling (including points of execution before and after calling of inline functions as well as TVM subroutines introduced by EXECUTE or similar primitives). Local correctness should always be a subset of global correctness. An example of a local correctness statement is below:

```
Definition roundCorrectLocally (l : Ledger)(r : RoundsBase_ι_Round)
:=
    _roundCorrectNonNegative r /\
    _roundCorrectStakeIsTheMost r /\
    _roundCorrectStakeSum r /\
    _roundCorrectStakes l r /\
    _roundValidatorRemainingStake r /\
    _roundNoDupStakes r /\
    _roundValidatorRemainingStakeLessOrEqualValidatorStake l r /\
    ledgerCorrectLocally l /\
    _roundCorrectStepsAndCompletionReasons l r.
```

As one can see the correctness predicates may contain complex predicates and so such statements can contain errors and require proving. An example of such a theorem (with proof, as in this particular case it's short enough and may be included into the present document) is provided below:

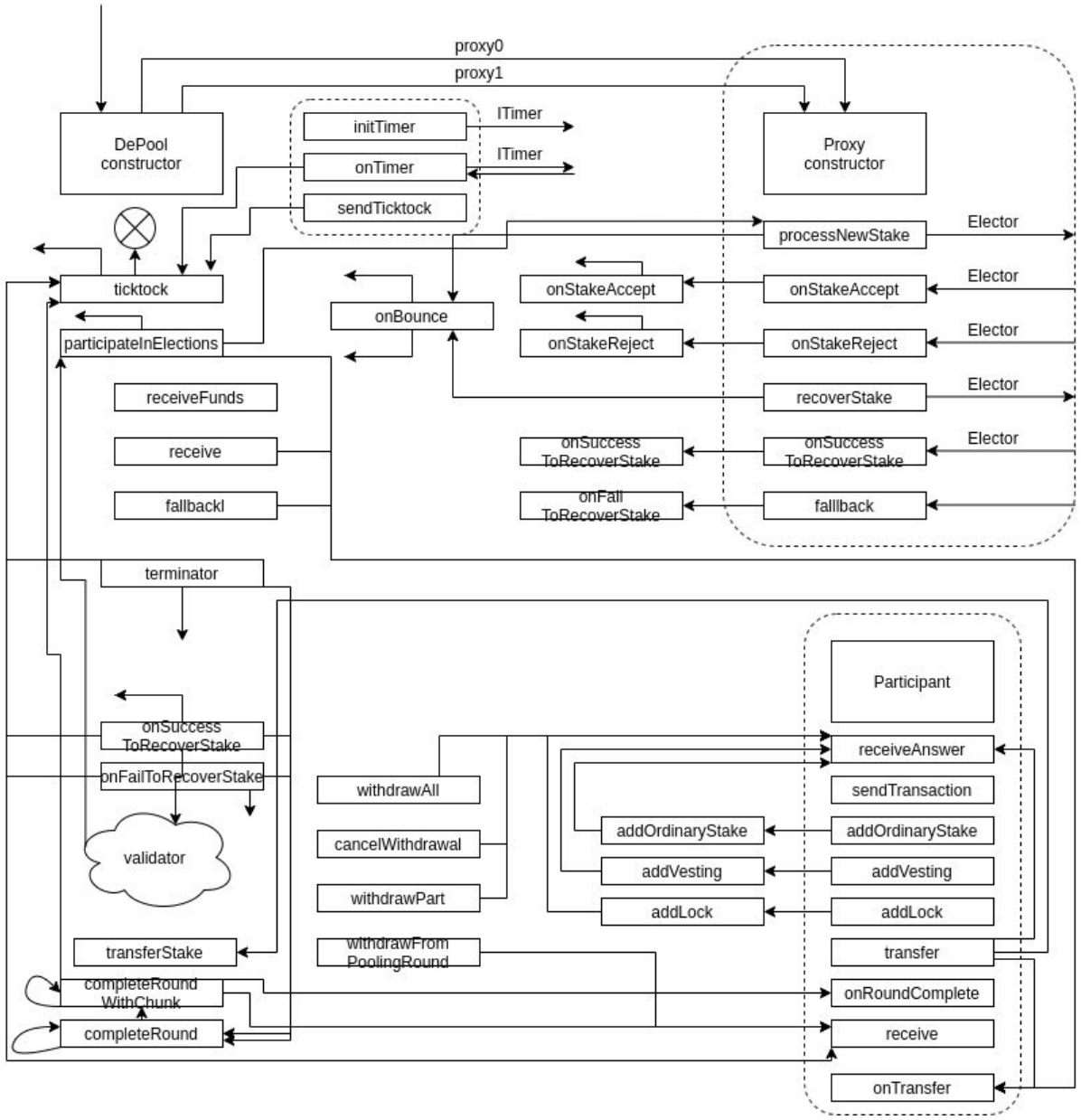
```
Theorem notValidator_vaidator_remaining_stake_less_stake :
    forall l r ,
    roundCorrectLocally l r ->
    allStakesAreNotEmpty l r ->
    isNotValidatorInRound l r ->
    RoundsBase_ι_Round_ι_validatorRemainingStake r <
RoundsBase_ι_Round_ι_stake r.
Proof.
```

```
intros. remember H as RCL. clear HeqRCL. unfold roundCorrectLocally
in H. decompose [and] H.
clear H. unfold
_roundValidatorRemainingStakeLessOrEqualValidatorStake in H8.
assert (stakeSum (validatorStake l r) < RoundsBase_ι _Round_ι _stake
r).
apply not_validator_stake_less_stakes ; assumption. lia.
Qed.
```

All the files related to correctness as well as required lemmas (with their proofs) may be found at *src/Scenarios/Correctness* directory.

m. Call tree

A possible issue is that at some point a sequence of sent messages will lead to infinite direct or mutual recursion. To investigate this possibility the following call graph that describes the calling of all the messages has been created:



The next steps in this investigation (to be done at Phase-3) are:

- prove that the provided call graph is correct
- ensure and prove that messages sent to external contracts (their handling may be arbitrary and even, in some cases, those external contracts can turn to be internal contract (so internal address may be used for external contract))

n. Axioms

The axioms represent the relationships that can not be described using correctness predicates such as time-related ones. Example of such an axiom is that validator hashes never repeat themself (strictly speaking, it's not true but probability of such a repeat is extremely low and it's assumed it never happens). Formally this axiom has been written in the following form:

```
Axiom validatorsNeverSame : forall l1 l2 l3,
    now l1 < now l2 ->
    now l2 < now l3 ->
    currentValidator l2 <> currentValidator l3 ->
    currentValidator l1 <> currentValidator l3.
```

The full list of axioms may be found in *src/Scenarios/Common/Axioms.v* file.

3. Summary

a. Achieved results

As a result of the present Phase-2 the preparation for the final and ultimate proving of the DePool contract has been mostly done with a few exceptions that may be considered as a debt for Phase-3. In particular, the following tasks has been completed or almost completed:

- Solidity -> Coq DSL translator
- Coq DSL -> Solidity reverse “translator”
- Generation of Ledger
- Generation of *eval* and *exec* functions with their proofs
- Modified and corrected projections (initially introduced at the Phase-1)
- Definition of direct scenarios (almost completed)
- Formal specification for *states*, *conditions* and *moves* with corresponding proofs (almost completed)
- Formal specification for correctness with corresponding proofs (almost completed)
- Call tree

The phase outcome can be described as follows:

- DePool contract has been mapped into Coq DSL
- As **no bugs were found** high (but not ultimate yet) level of confidence the current implementation if the contract is reliable has been achieved
- Contract has been proved at the functional level
- Made all the specification-related preparations to finally prove the contract at Solidity cross-functional (business) level as a result of Phase-3

b. Activity for Phase-3

The goal of the Phase-3 is to finally prove the DePool contract at the cross-functional (business) level. To achieve this goal the following activity is planned:

- Complete all the “almost completed” tasks from Phase-2
- Prove availability and reachability of the direct scenarios
- Prove that no *moves* are possible but already declared
- Prove that correct state *moves* into correct state
- Prove that the program can not get stuck inside the non-terminal projection *state*
- Prove that no infinite mutual recursion is possible

Gas related and TVM related issues are currently not planned for Phase-3 (they are considered for Phase-4) however can be moved there upon governance decision.

c. Company information

Pruvendo team has been actively involved into the formal verification based on *Coq* for the last six years. During this time a number of formal verification projects have been completed, mostly in the finance and banking industry.

The team is a pioneer in mathematical justification of the proof-of-stake consensus[11], implemented the prototype of the blockchain of the formally verified code, many-years active participant of different blockchain communities.

For the last year the team has concentrated on the *TON* project and successfully proved a *Multisig* contract introducing the whole bunch of new technologies and know-hows.

Currently the team obtains a unique set of tools that lets it to quickly formally verify any kind of *TON* smart contract.

In case of any questions feel free to contact us at team@pruvendo.com .

[11] https://consensusresearch.org/

Appendix A. Reverse “translation” from Coq eDSL into Solidity

```
1'. ↑ε[0-9]* -->
2'. \bε\b -->
3'. U[0-9]! -->
4'. D[0-9]! -->
5'. ↑+[0-9]+ -->
6'. ς\s+return#.*?\. --> .
7'.
\(\s*ξ\$\s+(?:[A-Za-z0-9_]+?_ι_)?([_A-Za-z0-9]*?)P?_ι_([_A-Za-z0-9
]*?)\s\) --> \1__DOT__\2
8'.
ξ(?:\s*\$)?\s+(?:[A-Za-z0-9_]+?_ι_)?([_A-Za-z0-9]*?)P?_ι_([_A-Za-z
0-9]*?) --> \1__DOT__\2
9'. ^\s*initial.*?>> -->
10'.
\(\s*LocalState_ι_[_A-Za-z0-9]*?_Л_([_A-Za-z0-9]*?)\s*:=\s*\$\s*Л_
\1\s*\) -->
11'. LocalState_ι_.*?_Л_(.*?)\b --> \1
12'. \:\s*[A-Za-z0-9_]+?_ι_([A-Za-z0-9_]+?) --> : \1
13'. ([A-Za-z0-9]+_И_[A-Za-z0-9]+)F --> \1
14'. [A-Za-z0-9_]+_ι_([A-Za-z0-9_]+?)P?_ι_([A-Za-z0-9_]+?)\b --> \2
15'. (?:(?!__DOT__)[A-Za-z0-9_])+_ι_([A-Za-z0-9_]+?) --> \1
16'.
declareLocal\s+([a-zA-ZА-Яа-я\'_α-ωA-Ω][a-zA-ZА-Яа-я0-9_\'α-ω
A-Ω]*)\s*:>:\s*([A-Za-z0-9_]+)\s+:=\s*{\|\|((?:.|\n)*?)\|\|}  ;  -->
\2 \1 = \2 ( { \3 __RCURLY__ ) __SEMICOLON__
17'.
declareLocal\s+(?!{\()(.*?)\s*(?:\:>\:|::::)\s*\(?(.*?)\)?\s*\?*:=
--> \2 \1 =
18'. declareLocal\s+({\(.*?\)})\s*\?*:= --> \1 =
19'.
declareLocal\s+([a-zA-ZА-Яа-я\'_α-ωA-Ω][a-zA-ZА-Яа-я0-9_\'α-ω
A-Ω]*) --> \1
20'.
\(\s*declareGlobal\!?\s+(?!{\()(.*?)\s*\:>\:\s*\((.*?)\)\s*:=((.|\n)
*?)\)\s*>> --> \2 \1 = \3;
21'.
\(\s*declareGlobal\!?\s+(?!{\()(.*?)\s*\:>\:\s*(.*?):=((.|\n)*?)\)\s
*>> --> \2 \1 = \3;
22'.
declareGlobal\s+([a-zA-ZА-Яа-я\'_α-ωA-Ω][a-zA-ZА-Яа-я0-9_\'α-
ωA-Ω]*) --> \1
23'. \(.*?declareInit.*? >> -->
```

```
24'.
([a-zA-ZА-Я а-я\'_α-ωΑ-Ω][a-zA-ZА-Я а-я0-9_\'α-ωΑ-Ω]*)\s*(?:\:
>\:|::::)\s*(\([A-Za-z0-9 ]+\)|(?:(?!\)\})[^,;)])*) --> \2 \1
25'. XHMap ([a-zA-Z0-9ι _]*) ([a-zA-Z0-9ι _]*) --> mapping (\1 => \2)
26'. XList\s+XInteger8 --> bytes
27'.
XMaybe\s+\(([a-zA-ZА-Я а-я\'_α-ωΑ-Ω][a-zA-ZА-Я а-я0-9_\'α-ωΑ-Ω
]*)\) --> optional(\1)
28'.
XMaybe\s+([a-zA-ZА-Я а-я\'_α-ωΑ-Ω][a-zA-ZА-Я а-я0-9_\'α-ωΑ-Ω]*
) --> optional(\1)
29'. XMaybe --> optional
30'.
do(?:\s*_)\s*←\s*\(\s*ForIndexedE\s*\(\s*xListCons\s+(.*?)\s*\(xList
Cons\s+xInt1\s*xListNil\s*\)\s*\)\s+do\s*\(\s*fun\s+\((.*?):\s+(.*?)
\)\s+=> --> for ( \3 \2 = \1; \2 < 2; ++ \2) {
31'. \) >>= fun r => return! \(xProdSnd r\) \) \?\?; --> }
__SEMICOLON__
32'.
\(\s*ForIndexed\s*\(\s*xListCons\s+(.*?)\(xListCons\s+xInt1\s*xListN
il\s*\)\s*\)\s+do\s*\(\s*fun\s+\((.*?):\s+(.*?)\)\s+=> --> for ( \3
\2 = \1; \2 < 2; ++ \2) {
33'.
\(\s*([a-zA-ZА-Я а-я\'_α-ωΑ-Ω][a-zA-ZА-Я а-я0-9_\'α-ωΑ-Ω]*)\s*
:\s*(optional\s*\(\s*[a-zA-ZА-Я а-я\'_α-ωΑ-Ω][a-zA-ZА-Я а-я0-9_
\'α-ωΑ-Ω]*\s*\)|mapping\s*\(\s*[a-zA-ZА-Я а-я\'_α-ωΑ-Ω][a-zA-ZА
-Я а-я0-9_\'α-ωΑ-Ω]*\s*=>\s*[a-zA-ZА-Я а-я\'_α-ωΑ-Ω][a-zA-ZА-
Я а-я0-9_\'α-ωΑ-Ω]*\s*\)|[a-zA-ZА-Я а-я\'_α-ωΑ-Ω][a-zA-ZА-Я а-
я0-9_\'α-ωΑ-Ω]*)\s*\) --> \2 \1,
34'. XInteger --> uint
35'. XBool --> bool
36'. XAddress --> address
37'. \(\s*xValue\s+I\s*\) -->
38'. [Xx]Value\s+I -->
39'. XValueValue -->
40'. XArray\s+([A-Za-z0-9]+) --> \1[]
41'. ::= --> :
42'.              Definition\s+([_a-zA-Z0-9Φ]*_[Cc]onstructor[0-9]+)
((?:.|\n)*),\s+:\s+.*?:= (?:\s|\n)*New ([A-Za-z0-9]+)[_ΦA-Za-z0-9]+
(.*?) >> --> constructor (\2) \3 \4 {
43'.     Definition\s+([_a-zA-Z0-9Φ]*_[Cc]onstructor[0-9]+)     -->
constructor (
44'. Definition\s+([_a-zA-Z0-9Φ]*)\'? --> function \1 (
45'. \. --> ;}
46'. ,\s*: --> :
```

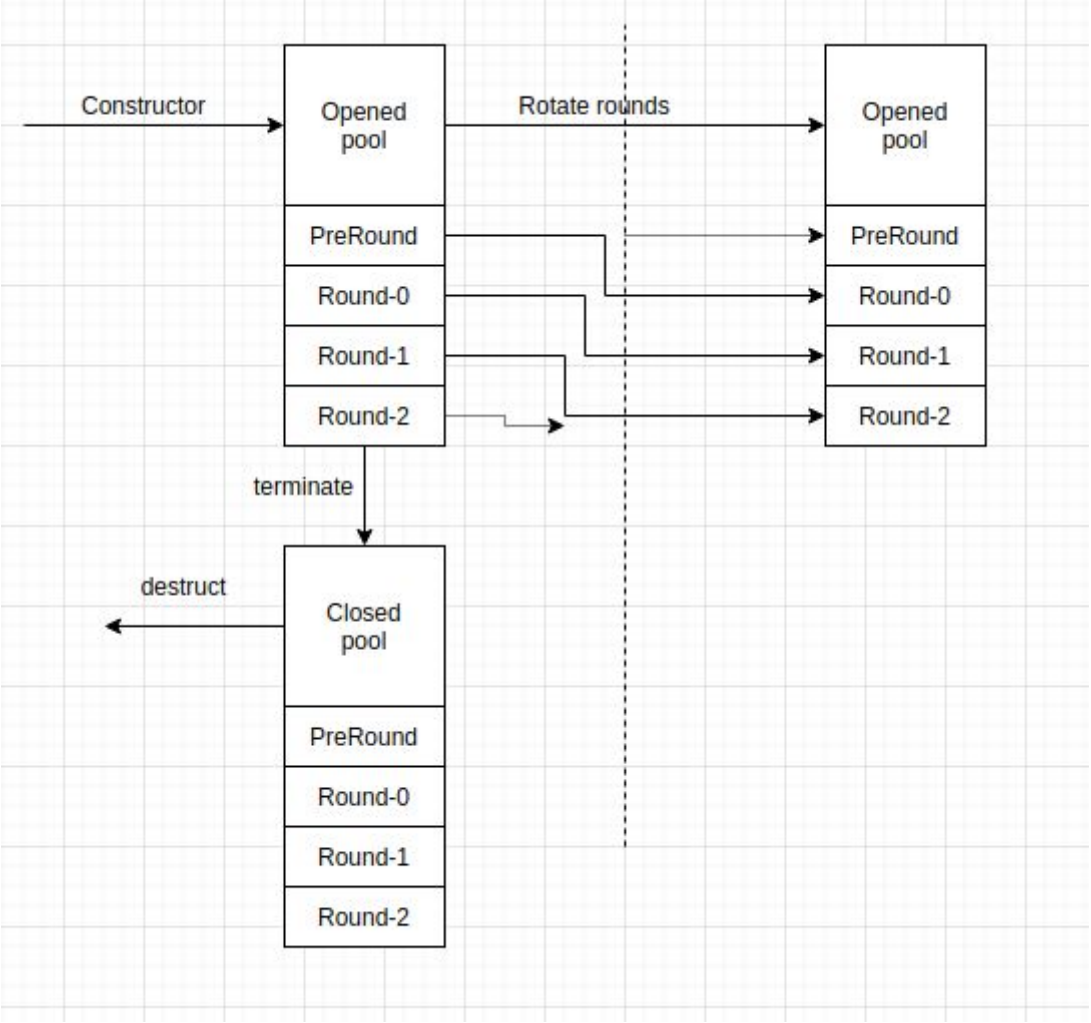
```
47'. \s*:\s*LedgerT(.*):=(?:\s|\n)*returns\s+(.*)>> --> ) returns \2
{
48'. LedgerT\s+([A-Za-z0-9]+)\s*:= --> LedgerT (\1) :=
49'. :\s*LedgerT --> ) returns
50'. \s*XErrorValue(.*)uint\s*\) -->  \1 )
51'. return# --> return
52'. # --> ,
53'.  function ((?:.|\n)*)returns ((?:[^{]|\n)*?):= --> function \1
returns \2 {
54'. \bTrue\b -->
55'. xBool(True|False) --> \1
56'. True --> true
57'. False --> false
58'. xInt([a-z0-9_]*) --> \1
59'. [a-zA-Z0-9_]*Ф_([a-zA-Z0-9_]*) --> \1
60'. xError\s*\( --> (
61'. (?<!{)\$\s*I\b -->
62'. (?<!{)\$(?!}) -->
63'. Require2? {{\$?\s*((.|\n)*?)\s*}} --> require ( \1 )
64'. ->store\s+(.+?)\s+([^\s;]+) --> .store(\1, \2)
65'. tvm_address\s*\(\s*\) --> address ( this )
66'. tvm_now\s*\(\s*\) --> now
67'.          tvm_rawConfigParam_([0-9]+)(\s*\(\s*\))?          -->
tvm.rawConfigParam(\1)
68'. tvm_configParam_([0-9]+)(\s*\(\s*\))? --> tvm.configParam(\1)
69'. \(\s*tvm_functionId (.*?)F?\s*\) --> tvm_functionId(\1)
70'. tvm_revert --> revert
71'. tvm_exit\s*\(\s*\) --> tvm.exit();
72'. tvm_balance\s*\(\s*\) --> address(this).balance
73'. tvm_([a-zA-Z]+) --> tvm.\1
74'. \bmessageValue\b --> value
75'. \bmessageBounce\b --> bounce
76'. \bmessageFlag\b --> flag
77'.
this->sendMessage\s*\(.*?(?:И_)?([A-Za-z0-9_]*?)F?\s+\(\!\!((?:.|\n
)*?)\!\!\)\s*\)\s*with\s*\{\|\|((.|\n)*?)\|\|\}    -->    this.\1{\3
__RCURLY__ (\2)
78'.
this->sendMessage\s*\(.*?(?:И_)?([A-Za-z0-9_]*?)F?\s+\s*\)\s*with\s
*\{\|\|((.|\n)*?)\|\|\} --> this.\1{\2 __RCURLY__ ()
79'.
(?:\"(.*?)\"|([A-Za-z0-9]+))\s+of\s+\(\s*(.*?)\)\s+->sendMessage\s*\
(.*?(?:И_)?([A-Za-z0-9_]*?)F?\s+\(\!\!((?:.|\n)*?)\!\!\)\s*\)[\s\n]
*with\s*\{\|\|((?:.|\n)*?)\|\|\} --> \1\2(\3).\4{\6 __RCURLY__ (\5)
80'. (?<![A-Za-z_9])If2?!* --> if
81'. then -->
```

```
82'. \?\?:= --> =
83'. _\s+\?:= -->
84'. := --> =
85'. \b_\b -->
86'. Л_ -->
87'. \s0\s+!- -->  !-
88'. \?([!<>=]+) --> \1
89'. !([+\-*/%]) --> \1
90'. !¬ --> !
91'. !& --> &&
92'. !\| --> ||
93'. ::: --> :
94'. \[\[ --> [
95'. \]\] --> ]
96'. \[\( --> (
97'. \)\] --> )
98'. \{\( --> (
99'. \)\} --> )
100'. \^\^ --> .
101'. ->min(?![0-9]) --> .min()
102'. math->min[0-9]+ --> math.min
103'. \(\s*if\b --> if
104'. (?<!})}\s*\) --> ; }
105'. ->set\s*([A-Za-z0-9]+\)) --> ->set (\1
106'. ->toCell --> .toCell()
107'. ->set --> .set
108'. \(\s*->emit ((.|\n)*?)\)(\s|\n)*(;|>>) --> ->emit \1;
109'. ->emit\s*\(((.|\n)*?)\)\s*; --> ->emit \1;
110'. ->emit --> emit
111'. ->selfdestruct --> selfdestruct
112'.
->(fetch|next|exists|delete|push)\s+([a-zA-ZА-Яа-я\'_α-ωΑ-Ω][a-z
A-ZА-Яа-я0-9_\'α-ωΑ-Ω]*) --> .\1(\2)
113'. ->(get|hasValue|empty|reset) --> .\1()
114'. ->>? --> .
115'. \bDePoolClosed --> DePoolClosed()
116'.  DePoolLib__DOT__RequestC((\s+[a-zA-Z0-9]+)*) -->  Request( \1
__ENDREQUEST__
117'. (?=([A-Za-z ]+)__ENDREQUEST__)([A-Za-z]+) --> \2,
118'. ,\s*__ENDREQUEST__ --> )
119'. msg_sender\s*\(\s*\) --> msg.sender
120'. msg_value\s*\(\s*\) --> msg.value
121'. msg_value --> msg.value
122'. msg_pubkey\s*\(\s*\) --> msg.pubkey ()
123'.     \s*\(\s*optional\s*\(([A-Za-z0-9]+)\)\s*\)(?!\s*{)     -->
optional(\1)
```

```
124'. >>=(.|\n)*\?; --> ;
125'. \(+\s*While ((.|\n)*?)\s+do(\s|\n)+\( --> while \1 {
126'. do\s+←\s*\(\s*WhileE\s+((?:.|\n)*?)\s+do --> while \1 {
127'. continue!(\s+I)? -->
128'. ^\s*(\)\s*)+>> --> ;}
129'. >> --> ;
130'. {\|\| --> {
131'. \|\|} --> __RCURLY__
132'. returns\s*\(\s*\((.*?)\)\s*\) --> returns (\1)
133'. \(\s*delMin(.*?)\) --> \1.delMin().get()
134'. completionReason2uint --> uint8
135'. emit\s+round --> emit Round
136'. \breturns(\s*\(\s*\))?\s*[{=] --> {
137'. _И_ --> .
138'. return!{1,3} --> return
139'. } --> ; }
140'. ;\s*; --> ;
141'. \(!+ --> (
142'. !+\) --> )
143'. }\s*; --> }
144'. (xError false) --> false
145'. ([0-9]+)\s*\*\s*x1_day --> (\1 days)
146'. =\s*default -->
147'.
(if\s+\(\s*(isRound[0-9]\s*\(\s*roundId\s*\)|isProcessNewStake)\s*\)
\s*{\s*)\( --> \1
148'. \(\s*delete\s+([a-zA-Z0-9_\[\]\.\n ]+)\) --> delete \1
149'.                                                    (?<![A-Za-z0-9_
\t])\s*\(\s*([A-Za-z0-9_]+(\s*\.\s*[A-Za-z0-9_]+)*)\s*\)\s* --> \1
150'. {\s*(;\s*)+} --> { }
151'. __DOT__ --> .
152'. __RCURLY__ --> }
153'. __SEMICOLON__ --> ;
154'. \(\s*xError\s+I\s*\) -->
```

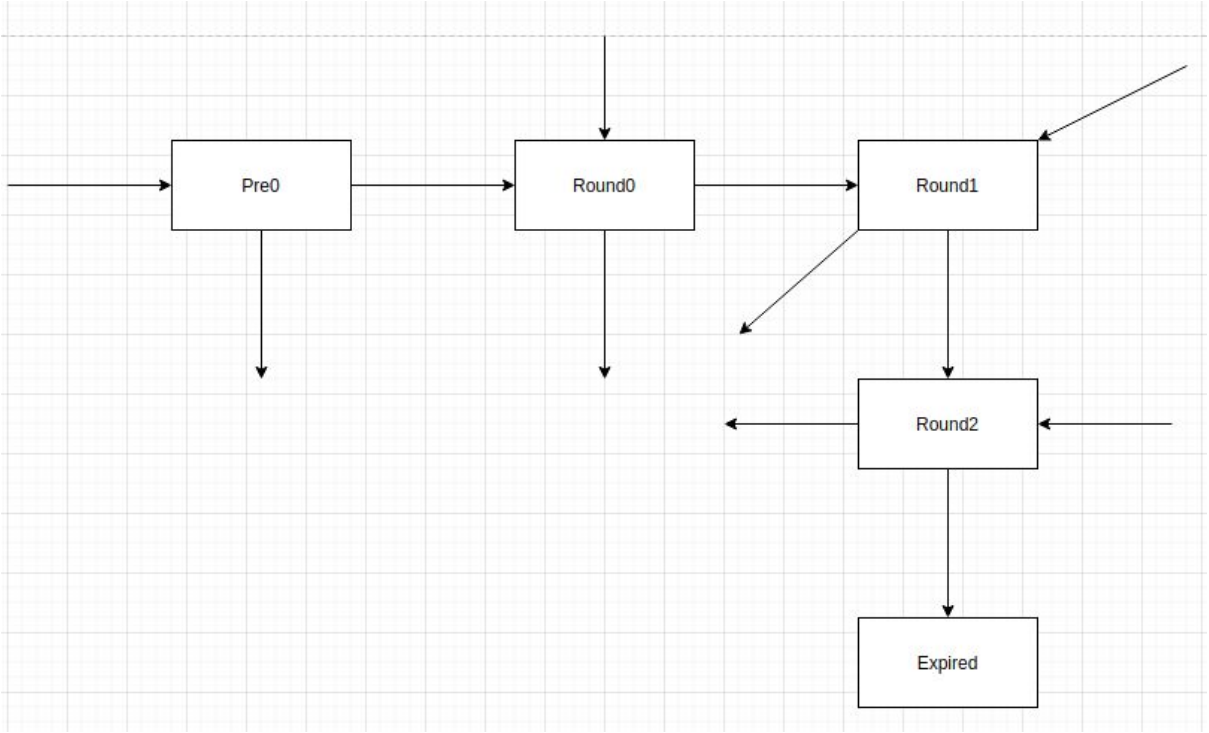
Appendix B : The diagrams of all the projections

Pool state projection:



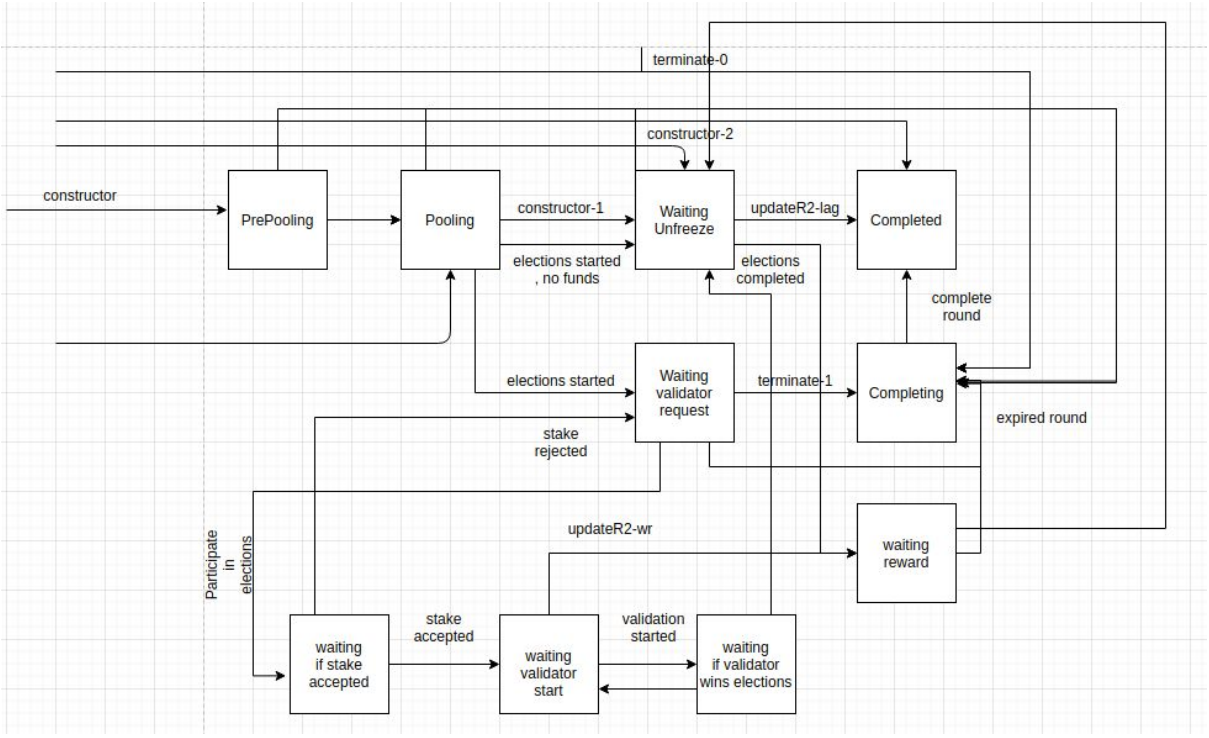
This is a projection of the DePool contract into *m_closed* and *m_rounds* variables.

Round pool projection:



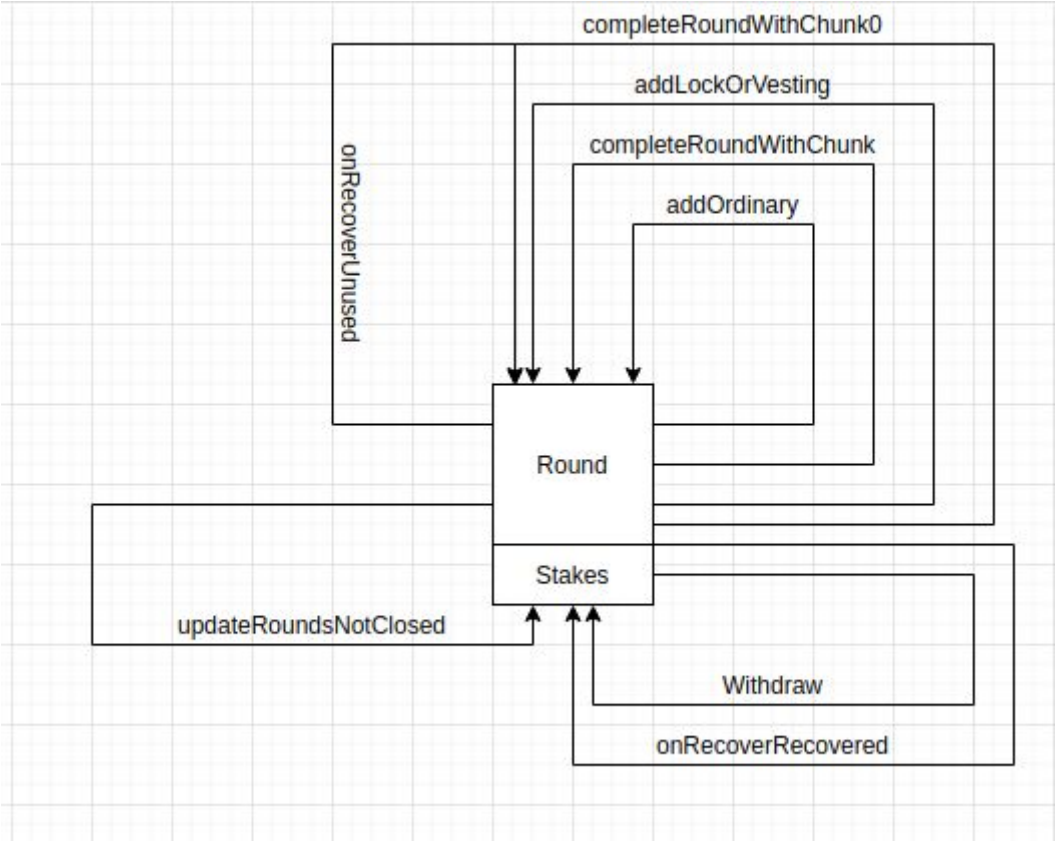
Evolution of particular round inside a pool

Round steps projection:



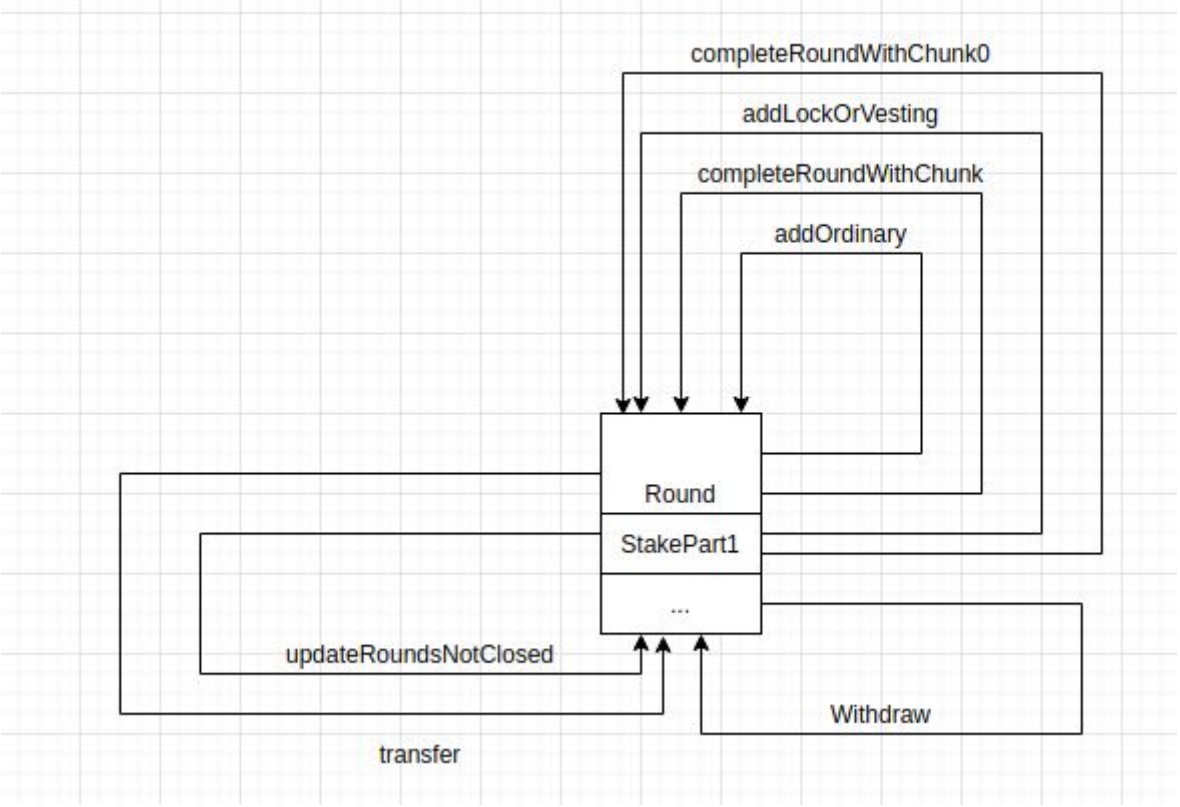
Projection of the particular round into *step* variable.

Round stakes projection:



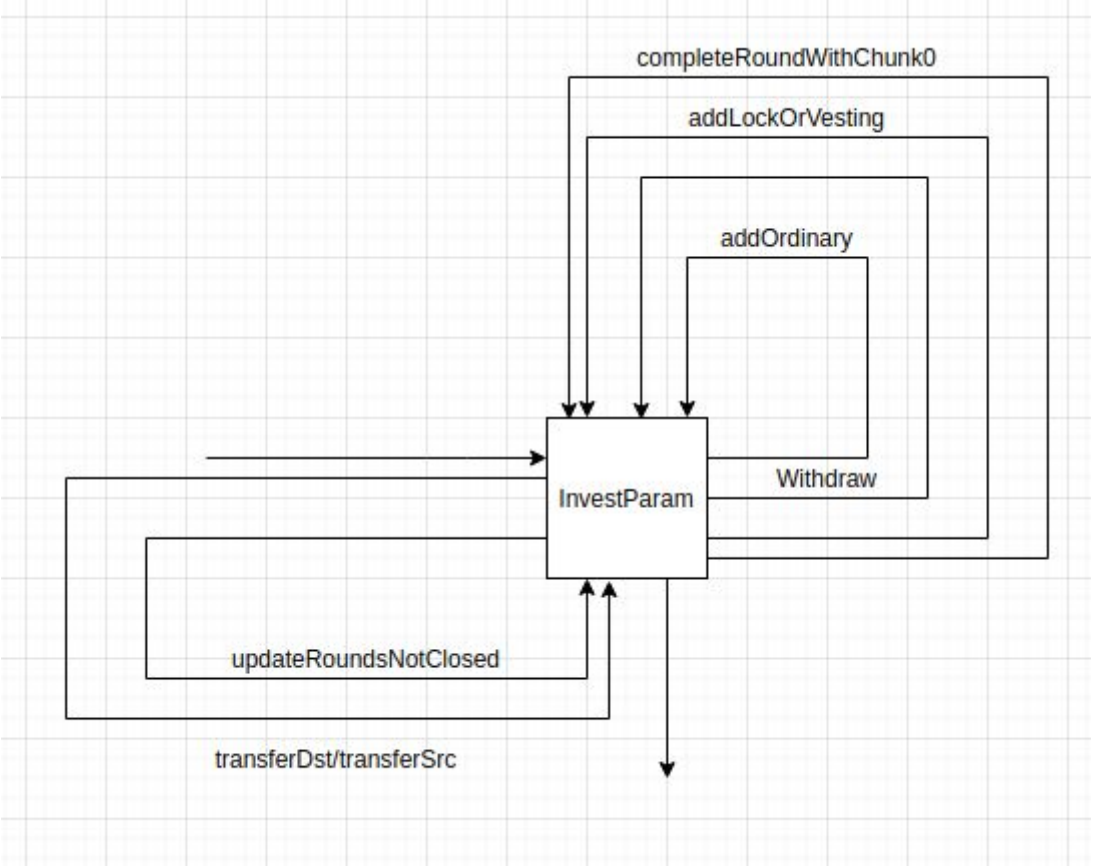
Projection of the particular round into the following variables: *stake, recoveredStake, unused, isValidatorStakeCompleted, rewards, validatorStake, handledStakeAndRewards*.

Round stake list projection:



Projection of the particular round into *stakes* variable.

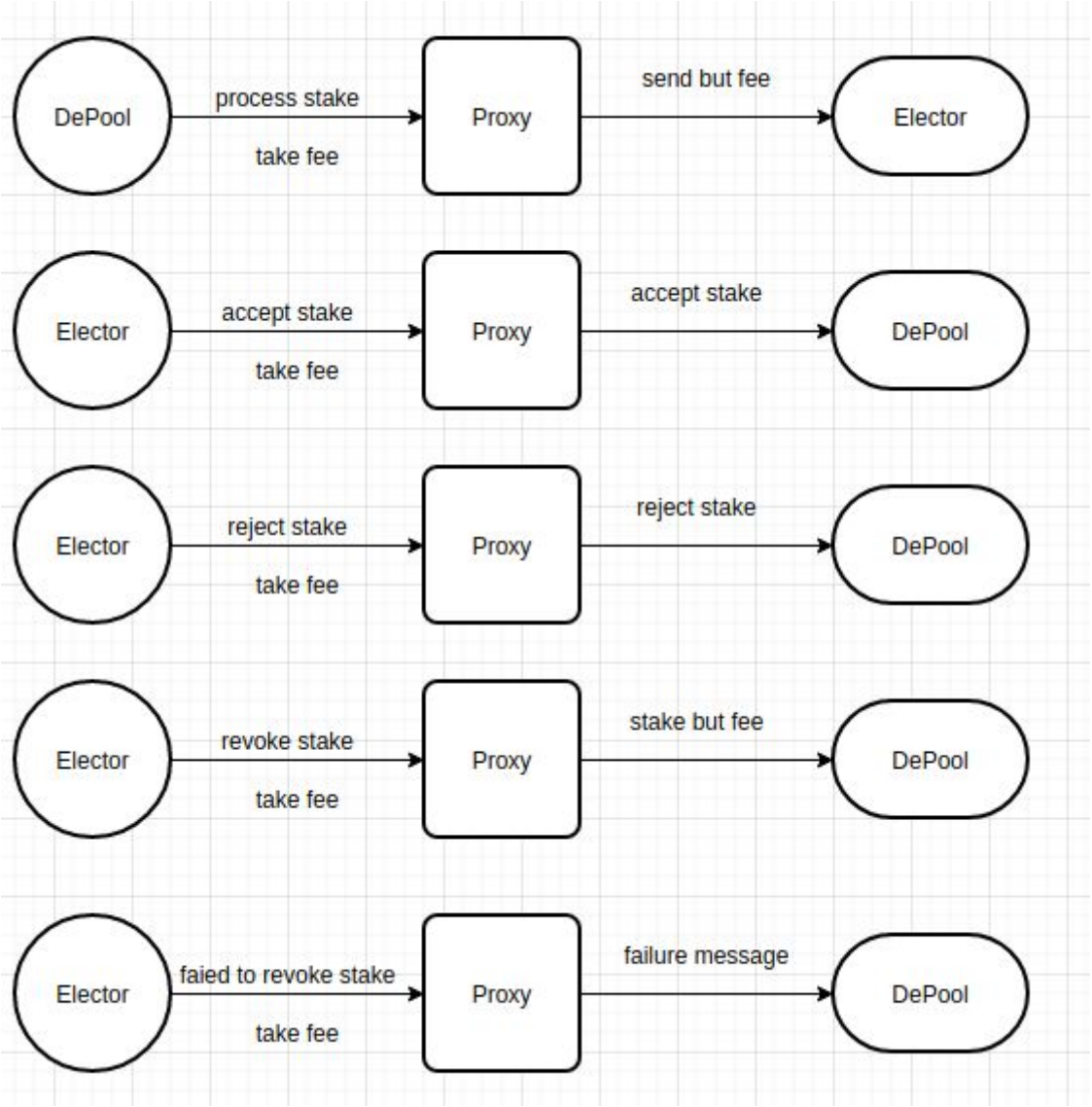
InvestParams projection:



Evolution of a single InvestParams within one round.

Participant projection has the same diagram as above and so not redrawn here.

Proxy stateless workflow diagram:


DePool
process stake
take fee
Proxy
send but fee
Elector
Elector
accept stake
take fee
Proxy
accept stake
DePool
Elector
reject stake
take fee
Proxy
reject stake
DePool
Elector
revoke stake
take fee
Proxy
stake but fee
DePool
Elector
faied to revoke stake
take fee
Proxy
failure message
DePool