Partial Alloy specification to include into the report

Structural Alloy specification

Tinyman V2 protocol is very flexible in terms of imposed restrictions on a transaction group structure. This feature leads to some advantages and disadvantages. Obvious benefits are simplicity of an integration into other protocols.

Main disadvantage is complication of safety analysis, because of huge amount of possible configurations of a transaction group.

To tackle this problem Alloy analyzer was used. Alloy specification language was specifically designed to perform structural analysis in various domains. Also it is worth to mention that Alloy analyzer via SAT solvers enumerates and checks all possible structures up to specified bounds. So, if, for instance, we ask Alloy analyzer to check some property for up to 16 transactions in transaction group, up to 16 accounts, up to 16 assets, up to 16 pools, etc, then we may be rather confident that property is true for all possible configurations.

Specification was designed to model most important abstract qualitative structural properties of the protocol.

Using the specification we were able to do several things: first one is to manually review possible transaction groups under some conditions, and the second is to check that some properties are held in all possible configurations up to specified bounds.

Brief overview of the specification

Some important modelled items:

- Assets

Assets are modelled via ordered set `Asset` with one selected element `Algo`. Algo is always constrained to be the first element.

```
sig Asset{}
one sig Algo in Asset {} {Algo = oa/first}
```

- Accounts

Base signature for accounts:

```
sig Account{}
```

`Application` is some selected atom in `Account`:

```
one sig Application in Account {
```

with next relations (fields):

- `fee_setter` - points to one account that corresponds to a fee setter account:

```
  fee_setter: one Account,
```

- `fee_collector` - self-describing:

```
  fee_collector: one Account,
```

- `fee-manager` - self-describing:

```
  fee_manager: one Account
}
```

- Pools

Pools are selected set from `Account` set:

```
sig Pool in Account {
```

Assets of a pool:

```
  asset1 : one Asset,
  asset2 : one Asset,
  pool_token: one Asset,
```

Pool state regarding issued tokens:

```
  tokens_issued: Transaction -> one TokensIssuedState,
```

Lock state for each transaction:

```
  lock: Transaction -> one LockState
} {
```

State of the lock and tokens may change in some transaction in transaction group, so locks and property of issued tokens are modeled via realtions between `Transaction` and corresponding enumeration.

- Transactions

Transaction may be one of two types: transfer and application call:

```
enum TransactionType {
  Transfer,
  AppCall
}
```

```
sig Transaction {
```

Main fields of a `Transaction`:

```
  type: one TransactionType,
  sender: one Account,
  receiver: lone Account,
  asset: lone Asset,
  op: lone Operation,
```

- Operations

Operations correspond 1-to-1 to application calls.

Next list is the list of modelled operations:

```
enum Operation {
  ...
-- amm
  OpAddInitialLiquidity,
  OpAddLiquidity,
  OpRemoveLiquidity,
  OpSwap,
  OpFlashLoan,
  OpVerifyFlashLoan,
  OpFlashSwap,
  OpVerifyFlashSwap
}
```

Some important properties that were checked using the specification

- `Verify flash swap` is performed always under the lock:

```
verify_flash_swap_always_under_lock:
check {
  all vfs: all_transactions_for[OpVerifyFlashSwap]
  | AmmParams.pool_address[vfs].lock[vfs] = Locked
} for 16 but 6 int

--   No counterexample found. Assertion may be valid. 481047ms.
```

- `Verify flash swap` is the only possible operation for a locked pool:

```
only_verify_flash_swap_is_possible_for_locked_pool:
check {
  all t: amm_transactions
  | AmmParams.pool_address[t].lock[t] = Locked implies t.op =
OpVerifyFlashSwap
} for 16 but 6 int

-- No counterexample found. Assertion may be valid. 177ms.
```

- `Flash loan` and `Verify flash loan` always symmetrical and each of pair refers to the same pool and user address:

```
-- check flash loan configurations

flash_loan1:
check {
  OpFlashLoan not in Transaction.op
  implies OpVerifyFlashLoan not in Transaction.op
}
      for 16 but 6 int
-- No counterexample found. Assertion may be valid. 686ms.

flash_loan2:
check {
  OpVerifyFlashLoan not in Transaction.op
  implies OpFlashLoan not in Transaction.op
}
      for 16 but 6 int
-- No counterexample found. Assertion may be valid. 605ms.

flash_loan3:
check {
  all fl: all_transactions_for[OpFlashLoan] {
    let vfl = FlashLoanParams.verify_txn_index[fl] {
      MainParams.user_address[fl] = MainParams.user_address[vfl]
    }
  }
} for 16 but 6 int
-- No counterexample found. Assertion may be valid. 2665ms.

flash_loan4:
check {
  all fl: all_transactions_for[OpFlashLoan] {
    let vfl = FlashLoanParams.verify_txn_index[fl] {
      AmmParams.pool_address[fl] = AmmParams.pool_address[vfl]
    }
  }
} for 16 but 6 int
-- No counterexample found. Assertion may be valid. 2232ms.
```

- `Flash swap` and `Verify flash swap` always referto the same pool and user address:

```
-- check flash loan configurations
flash_swap1:
check {
  all fs: all_transactions_for[OpFlashSwap] {
    let vfs = FlashSwapParams.verify_txn_index[fs] {
      MainParams.user_address[fs] = MainParams.user_address[vfs]
    }
  }
} for 16 but 6 int
-- No counterexample found. Assertion may be valid. 4546ms.

flash_swap2:
check {
  all fs: all_transactions_for[OpFlashSwap] {
    let vfs = FlashSwapParams.verify_txn_index[fs] {
      AmmParams.pool_address[fs] = AmmParams.pool_address[vfs]
    }
  }
} for 16 but 6 int
-- No counterexample found. Assertion may be valid. 3849ms.
```