MODULE *SASwap_ZmnSCPxj*

SASwap TLA+ specification (c) by Dmitry Petukhov (https://github.com/dgpv)
Licensed under a Creative Commons Attribution-ShareAlike 4.0 International
License <http://creativecommons.org/licenses/by-sa/4.0/>

EXTENDS *Naturals*, *Sequences*, *FiniteSets*, *TLC*

CONSTANT *PARTICIPANTS_IRRATIONAL* Can participants act irrational ?
ASSUME *PARTICIPANTS_IRRATIONAL* $\in$ BOOLEAN

CONSTANT *BLOCKS_PER_DAY*
More blocks per day means larger state space to check
ASSUME *BLOCKS_PER_DAY* ≥ 1

A transaction that has no deadline can be 'stalling',
i.e. not being sent while being enabled, for this number of days
CONSTANT *MAX_DAYS_STALLING*
More days allowed stalling means larger state space to check
ASSUME *MAX_DAYS_STALLING* ≥ 1

Is it possible for participants to send transactions
bypassing the *mempool* (give directly to the miner)
CONSTANT *STEALTHY_SEND_POSSIBLE*
When TRUE, the state space is increased dramatically.
ASSUME *STEALTHY_SEND_POSSIBLE* $\in$ BOOLEAN

Operator to create transaction instances
$Tx(id,\ ss,\ by,\ to,\ via) \triangleq$
$\quad [id \mapsto id,\ ss \mapsto ss,\ to \mapsto to,\ by \mapsto by,\ via \mapsto via]$

VARIABLE *blocks* $\langle \{Tx,\ \ldots\},\ \ldots \rangle$
VARIABLE *next_block* $\{Tx,\ \ldots\}$
VARIABLE *mempool* $\{Tx,\ \ldots\}$
VARIABLE *shared_knowledge* $\{Tx,\ \ldots\}$
VARIABLE *signers_map* $[participant \mapsto \{allowed\_sig,\ \ldots\}]$
VARIABLE *per_block_enabled* $\langle \{Tx,\ \ldots\},\ \ldots \rangle$
VARIABLE *wont_send* $\{id,\ \ldots\}$

$fullState \triangleq \langle blocks,\ next\_block,\ signers\_map,\ shared\_knowledge,\ mempool,$
$\qquad per\_block\_enabled,\ wont\_send \rangle$
$unchangedByMM \triangleq \langle blocks,\ signers\_map,\ shared\_knowledge,\ mempool,\ wont\_send \rangle$

A few generic operators

$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$
$Min(set) \triangleq \text{CHOOSE } x \in set : \forall\, y \in set : x \leq y$
$Max(set) \triangleq \text{CHOOSE } x \in set : \forall\, y \in set : x \geq y$

Various definitions that help to improve readability of the spec

$Alice \triangleq$ "Alice"
$Bob \triangleq$ "Bob"
$participants \triangleq \{Alice, Bob\}$

$sigAlice \triangleq$ "sigAlice"
$sigBob \triangleq$ "sigBob"
$secretAlice \triangleq$ "secretAlice"
$secretBob \triangleq$ "secretBob"
$all\_secrets \triangleq \{secretAlice, secretBob\}$
$all\_sigs \triangleq \{sigAlice, sigBob, secretAlice, secretBob\}$

$tx\_lock\_A \triangleq$ "tx_lock_A"
$tx\_lock\_B \triangleq$ "tx_lock_B"
$tx\_success \triangleq$ "tx_success"
$tx\_refund\_1 \triangleq$ "tx_refund_1"
$tx\_timeout \triangleq$ "tx_timeout"
$tx\_spend\_A \triangleq$ "tx_spend_A"
$tx\_spend\_B \triangleq$ "tx_spend_B"
$tx\_spend\_success \triangleq$ "tx_spend_success"
$tx\_spend\_refund\_1 \triangleq$ "tx_spend_refund_1"
$tx\_spend\_timeout \triangleq$ "tx_spend_timeout"

$nLockTime \triangleq$ "nLockTime"
$nSequence \triangleq$ "nSequence"
$NoTimelock \triangleq [days \mapsto 0, type \mapsto nLockTime]$

If blocks per day are low, the absolute locks need to be shifted,
otherwise not all contract paths will be reachable

$$ABS\_LK\_OFFSET \triangleq \begin{array}{lll} \text{CASE} & BLOCKS\_PER\_DAY = 1 & \to 2 \\ \square & BLOCKS\_PER\_DAY = 2 & \to 1 \\ \square & \text{OTHER} & \to 0 \end{array}$$

The map of the transactions, their possible destinations and timelocks.

Adaptor signatures are modelled by an additional value in the required signature set – ss . For modelling purposes, the secret acts as just another signature.
ds stands for "destinations", and lk stands for "lock" (timelocks).
Only blockheight-based timelocks are modelled.

$$
\begin{array}{l}
tx\_map \triangleq [\\
\quad \text{'Contract' transactions – destinations are other transactions} \\
\quad tx\_lock\_A \mapsto [ds \mapsto \{tx\_success,\ tx\_refund\_1,\ tx\_timeout,\ tx\_spend\_A\}, \\
\qquad\qquad\qquad\quad ss \mapsto \{sigAlice\}], \\
\quad tx\_lock\_B \mapsto [ds \mapsto \{tx\_spend\_B\}, \\
\qquad\qquad\qquad\quad ss \mapsto \{sigBob\}], \\
\quad tx\_success \mapsto [ds \mapsto \{tx\_spend\_success\}, \\
\qquad\qquad\qquad\quad ss \mapsto \{sigAlice,\ sigBob,\ secretBob\}], \\
\quad tx\_refund\_1 \mapsto [ds \mapsto \{tx\_spend\_refund\_1\}, \\
\qquad\qquad\qquad\quad ss \mapsto \{sigAlice,\ sigBob,\ secretAlice\}, \\
\qquad\qquad\qquad\quad lk \mapsto [days \mapsto ABS\_LK\_OFFSET + 1,\ type \mapsto nLockTime]], \\
\quad tx\_timeout \mapsto [ds \mapsto \{tx\_spend\_timeout\}, \\
\qquad\qquad\qquad\quad ss \mapsto \{sigAlice,\ sigBob,\ secretBob\}, \\
\qquad\qquad\qquad\quad lk \mapsto [days \mapsto ABS\_LK\_OFFSET + 2,\ type \mapsto nLockTime]], \\
\quad \text{'Terminal' transactions – destinations are participants} \\
\quad tx\_spend\_A \mapsto [ds \mapsto \{Alice,\ Bob\}, \\
\qquad\qquad\qquad\quad ss \mapsto \{sigAlice,\ sigBob\}], \\
\quad tx\_spend\_B \mapsto [ds \mapsto \{Alice,\ Bob\}, \\
\qquad\qquad\qquad\quad ss \mapsto \{secretAlice,\ secretBob\}], \\
\quad tx\_spend\_success \mapsto [ds \mapsto \{Bob\}, \\
\qquad\qquad\qquad\quad ss \mapsto \{sigBob\}], \\
\quad tx\_spend\_refund\_1 \mapsto [ds \mapsto \{Alice\}, \\
\qquad\qquad\qquad\quad ss \mapsto \{sigAlice\}], \\
\quad tx\_spend\_timeout \mapsto [ds \mapsto \{Bob\}, \\
\qquad\qquad\qquad\quad ss \mapsto \{sigBob\}] \\
]
\end{array}
$$

$all\_transactions \triangleq \text{DOMAIN } tx\_map$

first_transaction defined so that miner's actions do not need to refer to any contract-specific info, and can just refer to *first_transaction* instead.

$first\_transaction \triangleq tx\_lock\_A$

$ConfirmedTransactions \triangleq \{tx.id : tx \in \text{UNION } Range(blocks)\}$

$NextBlockTransactions \triangleq \{tx.id : tx \in next\_block\}$

$NextBlockConfirmedTransactions \triangleq$
$\quad ConfirmedTransactions \cup NextBlockTransactions$

$MempoolTransactions \triangleq \{tx.id : tx \in mempool\}$

$SentTransactions \triangleq ConfirmedTransactions \cup MempoolTransactions$

$EnabledTransactions \triangleq \{tx.id : tx \in \text{UNION } Range(per\_block\_enabled)\}$

$SharedTransactions \triangleq \{tx.id : tx \in shared\_knowledge\}$

$ContractTransactions \triangleq$
$\quad \{id \in all\_transactions :$
$\quad\quad \forall\, d \in tx\_map[id].ds : d \in all\_transactions\}$

$TerminalTransactions \triangleq$
$\quad \{id \in all\_transactions :$
$\quad\quad \forall\, d \in tx\_map[id].ds : d \in participants\}$

$\text{ASSUME } \forall\, id \in all\_transactions : \lor\, id \in TerminalTransactions$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \lor\, id \in ContractTransactions$

In this contract each transaction has only one parent, so we can use simple mapping from *dep_id* to parent *id*

$dependency\_map \triangleq$
$\quad [dep\_id \in \text{UNION } \{tx\_map[id].ds : id \in ContractTransactions\}$
$\quad\quad \mapsto \text{CHOOSE } id \in ContractTransactions : dep\_id \in tx\_map[id].ds]$

Special destination for the case when funds will still be locked at the contract after the transaction is spent

$Contract \triangleq$ "Contract"

$DstSet(id) \triangleq$

IF $id \in ContractTransactions$ THEN $\{Contract\}$ ELSE $tx\_map[id].ds$

The CASE statement has no 'OTHER ' clause - only single dst is expected

$SingleDst(id) \triangleq$ CASE $id \in ContractTransactions \to Contract$
$\square$ $Cardinality(tx\_map[id].ds) = 1$
$\to$ CHOOSE $d \in tx\_map[id].ds$: TRUE

The set of transactions conflicting with the given transaction

$ConflictingSet(id) \triangleq$
IF $id \in$ DOMAIN $dependency\_map$
THEN $\{dep\_id \in$ DOMAIN $dependency\_map :$
$dependency\_map[dep\_id] = dependency\_map[id]\}$
ELSE $\{id\}$

Transaction also conflicts with itself

ASSUME $\forall\, id \in all\_transactions : id \in ConflictingSet(id)$

$ConfirmationHeight(id) \triangleq$
CHOOSE $bn \in$ DOMAIN $blocks : \exists\, tx \in blocks[bn] : tx.id = id$

All the transactions the given transaction depends on.
Because each transaction can only have one dependency in our model,
all dependencies form a chain, not a tree.

RECURSIVE $DependencyChain(\_)$
$DependencyChain(id) \triangleq$
IF $id \in$ DOMAIN $dependency\_map$
THEN $\{id\} \cup DependencyChain(dependency\_map[id])$
ELSE $\{id\}$

All the transactions that depend on the given transaction.
Dependants form a tree, but the caller is interested in just a set.

RECURSIVE $AllDependants(\_)$
$AllDependants(id) \triangleq$
LET $dependants \triangleq tx\_map[id].ds \setminus participants$
IN IF $dependants = \{\}$
THEN $\{id\}$
ELSE $dependants \cup$ UNION $\{AllDependants(d\_id) : d\_id \in dependants\}$

All transactions that cannot ever become valid because other, conflicting
transactions were confirmed befor them

$InvalidatedTransactions \triangleq$
    UNION $\{\{c\_id\} \cup AllDependants(c\_id) \quad : c\_id \in$
        UNION $\{ConflictingSet(id) \setminus \{id\} : id \in ConfirmedTransactions\}\}$

All transactions that is not yet sent/confirmed, and have a chance to be.
$RemainingTransactions \triangleq$
    $((all\_transactions \setminus ConfirmedTransactions) \setminus InvalidatedTransactions)$

$Timelock(id) \triangleq$ IF "lk" $\in$ DOMAIN $tx\_map[id]$ THEN $tx\_map[id].lk$ ELSE $NoTimelock$

$UnreachableHeight \triangleq 2^{30} + (2^{30} - 1)$

Calculate the height at which the timelock for the given transaction
expires, taking $BLOCKS\_PER\_DAY$ and dependencies confirmation into account
$TimelockExpirationHeight(id) \triangleq$
    LET $lk \triangleq Timelock(id)$
     IN CASE $lk.type = nLockTime$
                $\rightarrow lk.days * BLOCKS\_PER\_DAY$
          $\square$ $lk.type = nSequence$
                $\rightarrow$ IF $dependency\_map[id] \in ConfirmedTransactions$
                    THEN $ConfirmationHeight(dependency\_map[id])$
                        $+ lk.days * BLOCKS\_PER\_DAY$
                    ELSE $UnreachableHeight$

"Hard" deadline for transaction means that it is unsafe to publish
the transaction after the deadline
$Deadline(id) \triangleq$
    LET $hs \triangleq \{TimelockExpirationHeight(c\_id) :$
                $c\_id \in (ConflictingSet(id) \setminus \{id\}) \setminus wont\_send\}$
        $higher\_hs \triangleq \{h \in hs : h > TimelockExpirationHeight(id)\}$
     IN IF $higher\_hs = \{\}$
        THEN $UnreachableHeight$
        ELSE $Min(higher\_hs)$

"Soft" deadline for transaction means that after the deadline,
mining the transaction will mean that it was 'stalling' for too long
$SoftDeadline(id) \triangleq$
    LET $dl \triangleq Deadline(id)$
        $h \triangleq TimelockExpirationHeight(id)$
     IN IF $dl = UnreachableHeight$

```
            THEN IF id ∈ EnabledTransactions
                 THEN (CHOOSE en ∈ DOMAIN per_block_enabled :
                              ∃ tx      ∈ per_block_enabled[en] : tx.id = id)
                       + MAX_DAYS_STALLING * BLOCKS_PER_DAY
                 ELSE IF h ≠ UnreachableHeight
                      THEN h + MAX_DAYS_STALLING * BLOCKS_PER_DAY
                      ELSE 0
            ELSE dl

SigsAvailable(id, sender, to) ≜
    LET secrets_shared ≜
            UNION {tx.ss ∩ all_secrets : tx ∈ shared_knowledge}
        sigs_shared ≜
            UNION {tx.ss : tx ∈ {tx ∈ shared_knowledge : ∧ tx.id = id
                                                         ∧ tx.to = to}}
    IN  sigs_shared ∪ secrets_shared ∪ signers_map[sender]

DependencySatisfied(id, ids) ≜
    id ∈ DOMAIN dependency_map ⇒ dependency_map[id] ∈ ids

IsSpendableTx(tx, other_ids) ≜
    ∧ {} = ConflictingSet(tx.id) ∩ other_ids
    ∧ DependencySatisfied(tx.id, other_ids)
    ∧ tx.ss ⊆ SigsAvailable(tx.id, tx.by, tx.to)
    ∧ Len(blocks) ≥ TimelockExpirationHeight(tx.id)

Sending tx_spend_B does not actually expose secrets, because the secrets
are used as keys, and sigSecretBob would be exposed rather than secretBob.
Instead of introducing revealSecret < Alice | Bob > , sigSecret < Alice | Bob >
we simply filter out signatures of tx_spend_B before placing into shared knowledge
ShareKnowledge(knowledge) ≜
    LET knowledge_filtered ≜
            {IF tx.id ≠ tx_spend_B THEN tx ELSE [tx EXCEPT !.ss = {}] :
             tx ∈ knowledge}
        shared_knowledge may not change here, callers need to check if they care
    IN  shared_knowledge' = shared_knowledge ∪ knowledge_filtered

ShareTransactions(ids, by) ≜
    LET Ss(id) ≜ (tx_map[id].ss ∩ signers_map[by]) \ all_secrets
```

$txs \triangleq \{Tx(id,\ Ss(id),\ by,\ SingleDst(id),\ \text{“direct”}) : id \in ids\}$
IN $\land\ ShareKnowledge(txs)$
$\land\ shared\_knowledge' \neq shared\_knowledge$ not a new *knowledge* $\Rightarrow$ fail

Txs enabled at the current cycle, used to update *per_block_enabled* vector
$NewlyEnabledTxs \triangleq$
$\{tx \in$
UNION
{UNION
{
{
$Tx(id,\ tx\_map[id].ss,\ sender,\ to,\ \text{“enabled”}) : to \in DstSet(id)$
$\} : id \in RemainingTransactions$
$\} : sender \in participants$
$\} : \land \neg\exists\ etx \in \text{UNION}\ Range(per\_block\_enabled) : etx.id = tx.id$
$\land\ IsSpendableTx(tx,\ ConfirmedTransactions)$
$\}$

$SendTransactionToMempool(id,\ sender,\ to) \triangleq$
LET $tx \triangleq Tx(id,\ tx\_map[id].ss,\ sender,\ to,\ \text{“mempool”})$
IN $\land\ IsSpendableTx(tx,\ SentTransactions)$
$\land\ Len(blocks) < Deadline(id)$
$\land\ mempool' = mempool \cup \{tx\}$
$\land\ ShareKnowledge(\{tx\})$

Give *tx* directly to miner, bypassing global *mempool*
No *Deadline* check because information is not shared,
and after the block is mined, there's no possible contention
unless the block is orphaned. Orphan blocks are not modelled,
and therefore there's no need for additional restriction
as any state space restriction can possibly mask some other issue
$SendTransactionToMiner(id,\ sender,\ to) \triangleq$
$\land\ STEALTHY\_SEND\_POSSIBLE$
$\land$ LET $tx \triangleq Tx(id,\ tx\_map[id].ss,\ sender,\ to,\ \text{“miner”})$
IN $\land\ IsSpendableTx(tx,\ NextBlockConfirmedTransactions)$
$\land\ next\_block' = next\_block \cup \{tx\}$

$SendTransaction(id,\ sender,\ to) \triangleq$
$\lor\ \land\ SendTransactionToMempool(id,\ sender,\ to)$

```
        ∧ UNCHANGED next_block
    ∨ ∧ SendTransactionToMiner(id, sender, to)
      ∧ UNCHANGED ⟨mempool, shared_knowledge⟩

SendSomeTransaction(ids, sender) ≜
  LET SendSome(filtered_ids) ≜
        ∃ id ∈ filtered_ids \ wont_send :
        ∃ to ∈ (IF id ∈ ContractTransactions
                  THEN {Contract}
                  ELSE tx_map[id].ds ∩ {sender}) :
          SendTransaction(id, sender, to)
      terminal_ids ≜ ids ∩ TerminalTransactions
  IN  CASE PARTICIPANTS_IRRATIONAL
            → SendSome(ids)  Irrational participants do no prioritization
        □ ENABLED SendSome(terminal_ids)
            → SendSome(terminal_ids)  Can send terminal tx ⇒ do it immediately
        □ OTHER
            → SendSome(ids \ terminal_ids)

HasCustody(ids, participant) ≜
  ∃ id ∈ ids : ∃ tx ∈ UNION Range(blocks) : tx.id = id ∧ tx.to = participant

 Sharing secrets or keys has to occur before deadline to send tx_success
TooLateToShare ≜ Len(blocks) ≥ Deadline(tx_success)

Participant actions

 Helper operators to declutter the action expressions
NoSending ≜ UNCHANGED ⟨mempool, next_block⟩
NoKeysShared ≜ UNCHANGED signers_map
NoKnowledgeShared ≜ UNCHANGED shared_knowledge

AliceAction ≜
  LET Send(ids) ≜ SendSomeTransaction(ids, Alice)
      Share(ids) ≜ ShareTransactions(ids, Alice)
      SafeToSend(id) ≜
        CASE PARTICIPANTS_IRRATIONAL
                → TRUE  Unsafe txs are OK for irrational Alice
          □ secretAlice ∈ tx_map[id].ss
```

```
                    Once Alice shared tx_success, should never send out secretAlice
                  → ∨ tx_success ∉ {tx.id : tx ∈ shared_knowledge}
                    ∨ id = tx_spend_B  unless this is a transaction to get B
                                       which does not in fact expose secrets
          □ OTHER → TRUE
IN    ∨ ∧ Send({id ∈ RemainingTransactions : SafeToSend(id)})
        ∧ NoKeysShared
      ∨ ∧ {tx_lock_A, tx_lock_B} ⊆ ConfirmedTransactions
        ∧ ¬({tx_success, tx_timeout} ⊆ SharedTransactions)
        ∧ tx_refund_1 ∉ SentTransactions
        ∧ Share({tx_success, tx_timeout})
        ∧ NoSending ∧ NoKeysShared
      ∨ ∧ {tx_lock_A, tx_lock_B} ⊆ ConfirmedTransactions
        ∧ {tx_success, tx_refund_1, tx_timeout} ⊆ SharedTransactions
        ∧ secretBob ∈ signers_map[Alice]  Bob gave Alice his secret
        ∧ sigAlice ∉ signers_map[Bob]     Alice did not yet gave Bob her key
        ∧ tx_success ∉ ConfirmedTransactions  Swap went on-chain ⇒ no key sharing
        ∧ ¬TooLateToShare
        ∧ signers_map' = [signers_map   Give Alice's key to Bob
                          EXCEPT ![Bob] = @ ∪ {sigAlice}]
        ∧ NoSending ∧ NoKnowledgeShared

BobAction ≜
  LET Send(ids) ≜ SendSomeTransaction(ids, Bob)
      Share(ids) ≜ ShareTransactions(ids, Bob)
      tx_success_sigs ≜ SigsAvailable(tx_success, Bob, Contract)
  IN  ∨ ∧ Send(RemainingTransactions)
        ∧ NoKeysShared ∧ UNCHANGED wont_send
      ∨ ∧ tx_refund_1 ∉ SharedTransactions
        ∧ tx_success ∉ SentTransactions
        ∧ tx_timeout ∉ SentTransactions
        ∧ Share({tx_refund_1})
        ∧ wont_send' = wont_send ∪ {tx_timeout}
        ∧ NoSending ∧ NoKeysShared
      ∨ ∧ {tx_lock_A, tx_lock_B} ⊆ ConfirmedTransactions
        ∧ {tx_success, tx_refund_1, tx_timeout} ⊆ SharedTransactions
        ∧ tx_success ∉ SentTransactions
```

$\land\ tx\_timeout \notin SentTransactions$
$\land\ secretAlice \notin tx\_success\_sigs$
$\land\ secretBob \notin signers\_map[Alice]$
$\land\ \neg TooLateToShare$
$\land\ signers\_map' = [signers\_map$ Give *secretBob* to *Alice*
EXCEPT $![Alice] = @ \cup \{secretBob\}]$
$\land\ wont\_send' = wont\_send \setminus \{tx\_timeout\}$
$\land\ NoSending \land NoKnowledgeShared$

$MempoolMonitorActionRequired \triangleq$
$\exists\, tx \in mempool :\ \land\ Len(blocks) + 1 = Deadline(tx.id)$
$\land\ tx.id \notin NextBlockTransactions$

We update *next_block* directly rather than having to deal with fees and prioritization. What we want to model is the behavior of participants where once they have sent the transaction, they do anything possible to meet the deadline set by the protocol to confirm the transaction. Failure to do so before the deadline is out of scope, even though it could be caused by some unexpected *mempool* behavior.

Exact *mempool* behavior is too low-level and is better modelled separately to check that high-level constraints can be met. Although if we were to have more complex model where the amounts available for each participant are tracked, it might make sense to include the fees and *mempool* behavior into the model of the contract to catch the cases when participants just can't bump fees anymore, for example.

We could just not model the *mempool* monitoring, and constrain state space such that states with late *txs* are invalid, to express that we don't care about the cases when participants fail to get their *txs* confirmed in time. But maybe there could be some interesting behaviors to be modelled if more elaborate monitor action is implemented

$MempoolMonitorAction \triangleq$
LET $tx \triangleq$ CHOOSE $tx \in mempool : Len(blocks) + 1 = Deadline(tx.id)$
$txs\_to\_bump \triangleq \{tx\} \cup \{dptx \in mempool :$
$\land\ tx.id \in$ DOMAIN $dependency\_map$
$\land\ dptx.id = dependency\_map[tx.id]$
$\land\ dptx.id \notin NextBlockTransactions\}$
IN $next\_block' =$
$\{nbtx \in next\_block :$ conflicting *txs* are expunged from *next_block*
$\{\} = DependencyChain(nbtx.id) \cap$

$$\text{UNION } \{\mathit{ConflictingSet}(\mathit{bmptx.id}) : \mathit{bmptx} \in \mathit{txs\_to\_bump}\}$$
$$\cup \{[\mathit{bmptx} \text{ EXCEPT } !.\mathit{via} = \text{“fee-bump”}] : \mathit{bmptx} \in \mathit{txs\_to\_bump}\}$$

Miner action

$$\mathit{IncludeTxIntoBlock} \triangleq$$
$$\land \exists\, \mathit{tx} \in \mathit{mempool} :$$
$$\land \{\} = \mathit{ConflictingSet}(\mathit{tx.id}) \cap \mathit{NextBlockConfirmedTransactions}$$
$$\land \mathit{DependencySatisfied}(\mathit{tx.id},\ \mathit{NextBlockConfirmedTransactions})$$
$$\land \mathit{next\_block}' = \mathit{next\_block} \cup \{\mathit{tx}\}$$
$$\land \text{UNCHANGED } \langle \mathit{blocks},\ \mathit{mempool},\ \mathit{shared\_knowledge}\rangle$$

Needed to restrict the state space, so that model checking is feasible

$$\mathit{CanMineEmptyBlock} \triangleq$$
$$\land \mathit{first\_transaction} \in \mathit{ConfirmedTransactions}$$
$$\land \text{LET } \mathit{soft\_dls} \triangleq \{\mathit{SoftDeadline}(\mathit{id}) : \mathit{id} \in \mathit{RemainingTransactions}\}$$
$$\text{IN } \mathit{soft\_dls} \neq \{\} \land \mathit{Len}(\mathit{blocks}) + 1 < \mathit{Max}(\mathit{soft\_dls})$$

$$\mathit{MineTheBlock} \triangleq$$
$$\text{IF } \mathit{next\_block} = \{\}$$
$$\text{THEN } \land \mathit{CanMineEmptyBlock}$$
$$\land \mathit{blocks}' = \mathit{Append}(\mathit{blocks},\ \{\})$$
$$\land \text{UNCHANGED } \langle \mathit{mempool},\ \mathit{next\_block},\ \mathit{shared\_knowledge}\rangle$$
$$\text{ELSE } \land \mathit{blocks}' = \mathit{Append}(\mathit{blocks},\ \mathit{next\_block})$$
$$\land \mathit{mempool}' =$$
$$\{\mathit{tx} \in \mathit{mempool} : \text{ conflicting } \mathit{txs} \text{ are expunged from } \mathit{mempool}$$
$$\{\} = \mathit{DependencyChain}(\mathit{tx.id}) \cap$$
$$\text{UNION } \{\mathit{ConflictingSet}(\mathit{nbtx.id}) : \mathit{nbtx} \in \mathit{next\_block}\}\}$$
$$\land \mathit{next\_block}' = \{\}$$
$$\land \mathit{ShareKnowledge}(\mathit{next\_block} \setminus \mathit{mempool})$$

$$\mathit{MinerAction} \triangleq \mathit{IncludeTxIntoBlock} \lor \mathit{MineTheBlock}$$

Auxiliary action for soft-deadline tracking

$$\mathit{UpdateEnabledPerBlock} \triangleq$$
$$\mathit{per\_block\_enabled}' =$$
$$\text{IF } \mathit{Len}(\mathit{per\_block\_enabled}) < \mathit{Len}(\mathit{blocks}) + 1$$
$$\text{THEN } \mathit{Append}(\mathit{per\_block\_enabled},\ \mathit{NewlyEnabledTxs})$$
$$\text{ELSE } [\mathit{per\_block\_enabled} \text{ EXCEPT } ![\mathit{Len}(\mathit{blocks}) + 1] = @ \cup \mathit{NewlyEnabledTxs}]$$

High-level contract spec

$$
\begin{aligned}
&\mathit{AliceLostByMisbehaving} \triangleq \\
&\quad \land \mathit{HasCustody}(\{\mathit{tx\_spend\_B}\}, \mathit{Bob}) \\
&\quad \land \mathit{HasCustody}(\{\mathit{tx\_spend\_success}, \mathit{tx\_spend\_timeout}\}, \mathit{Bob})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{BobLostByBeingLateOnSuccess} \triangleq \\
&\quad \land \mathit{HasCustody}(\{\mathit{tx\_spend\_B}\}, \mathit{Alice}) \\
&\quad \land \mathit{HasCustody}(\{\mathit{tx\_spend\_refund\_1}\}, \mathit{Alice})
\end{aligned}
$$

$$\mathit{SwapUnnaturalEnding} \triangleq \mathit{BobLostByBeingLateOnSuccess} \lor \mathit{AliceLostByMisbehaving}$$

The normal, 'natural' cases.

$$
\begin{aligned}
&\mathit{SwapSuccessful} \triangleq \\
&\quad \land \mathit{HasCustody}(\{\mathit{tx\_spend\_B}\}, \mathit{Alice}) \\
&\quad \land \mathit{HasCustody}(\{\mathit{tx\_spend\_A}, \mathit{tx\_spend\_success}, \mathit{tx\_spend\_timeout}\}, \mathit{Bob})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{SwapAborted} \triangleq \\
&\quad \land \mathit{HasCustody}(\{\mathit{tx\_spend\_A}, \mathit{tx\_spend\_refund\_1}\}, \mathit{Alice}) \\
&\quad \land \ \lor \mathit{HasCustody}(\{\mathit{tx\_spend\_B}\}, \mathit{Bob}) \\
&\qquad \lor \mathit{tx\_lock\_B} \notin \mathit{SentTransactions}
\end{aligned}
$$

Deadlock is possible by design.
See https://*lists.linuxfoundation.org*/pipermail/bitcoin-dev/2020-June/017918.*html*

$$
\begin{aligned}
&\mathit{SwapDeadlocked} \triangleq \\
&\quad \text{LET } \mathit{stx} \triangleq \mathit{Tx}(\mathit{tx\_success}, \mathit{tx\_map}[\mathit{tx\_success}].\mathit{ss}, \mathit{Bob}, \mathit{Contract}, \text{“test”}) \\
&\qquad\ \ \mathit{rtx} \triangleq \mathit{Tx}(\mathit{tx\_refund\_1}, \mathit{tx\_map}[\mathit{tx\_refund\_1}].\mathit{ss}, \mathit{Alice}, \mathit{Contract}, \text{“test”}) \\
&\quad \text{IN} \quad \land \mathit{ConfirmedTransactions} = \{\mathit{tx\_lock\_A}, \mathit{tx\_lock\_B}\} \\
&\qquad\quad \land \mathit{IsSpendableTx}(\mathit{stx}, \mathit{SentTransactions}) \\
&\qquad\quad \land \mathit{IsSpendableTx}(\mathit{rtx}, \mathit{SentTransactions}) \\
&\qquad\quad \land \neg\text{ENABLED } \mathit{AliceAction} \\
&\qquad\quad \land \neg\text{ENABLED } \mathit{BobAction}
\end{aligned}
$$

All possible endings of the contract

$$
\begin{aligned}
\mathit{ContractFinished} \triangleq\ & \lor \mathit{SwapSuccessful} \\
& \lor \mathit{SwapAborted} \\
& \lor \mathit{SwapDeadlocked} \\
& \lor \mathit{PARTICIPANTS\_IRRATIONAL} \land \mathit{SwapUnnaturalEnding}
\end{aligned}
$$

Actions in the contract when it is not yet finished. Separated into

dedicated operator to be able to test ENABLED *ContractAction*

$$
\begin{array}{l}
ContractAction \triangleq \\
\quad \lor AliceAction \qquad\qquad\qquad \land \text{UNCHANGED } \langle blocks,\ wont\_send \rangle \\
\quad \lor BobAction \qquad\qquad\qquad\ \ \land \text{UNCHANGED } blocks \\
\quad \lor \text{IF } MempoolMonitorActionRequired \\
\qquad \text{THEN } MempoolMonitorAction \land \text{UNCHANGED } unchangedByMM \\
\qquad \text{ELSE } MinerAction \qquad\qquad\ \ \land \text{UNCHANGED } \langle signers\_map,\ wont\_send \rangle
\end{array}
$$

Invariants

$$
\begin{array}{l}
TypeOK \triangleq \\
\quad \text{LET } TxConsistent(tx,\ vias) \triangleq \land\ tx.id \in all\_transactions \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \land\ tx.ss \subseteq tx\_map[tx.id].ss \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \land\ tx.to\ \in DstSet(tx.id) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \land\ tx.by \in participants \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \land\ tx.via \in vias \\
\qquad\quad AllSigsPresent(tx) \triangleq\ tx.ss = tx\_map[tx.id].ss \\
\qquad\quad SigConsistent(sig) \triangleq\ \land\ sig.id \in all\_transactions \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \land\ sig.s \in all\_sigs \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \land\ sig.ds \subseteq participants \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cup \text{DOMAIN } dependency\_map \\
\quad \text{IN} \quad \land \forall\, tx \in \text{UNION } Range(blocks) : \\
\qquad\qquad \lor\ \land\ TxConsistent(tx,\ \{\text{“mempool”},\ \text{“miner”},\ \text{“fee-bump”}\}) \\
\qquad\qquad\quad\ \ \land\ AllSigsPresent(tx) \\
\qquad\qquad \lor\ Print(\langle \text{“}\sim\text{TypeOK blocks”},\ tx\rangle,\ \text{FALSE}) \\
\qquad\quad \land \forall\, tx \in \text{UNION } Range(per\_block\_enabled) : \\
\qquad\qquad \lor\ \land\ TxConsistent(tx,\ \{\text{“enabled”}\}) \\
\qquad\qquad\quad\ \ \land\ AllSigsPresent(tx) \\
\qquad\qquad \lor\ Print(\langle \text{“}\sim\text{TypeOK blocks”},\ tx\rangle,\ \text{FALSE}) \\
\qquad\quad \land \forall\, tx \in next\_block : \\
\qquad\qquad \lor\ \land\ TxConsistent(tx,\ \{\text{“mempool”},\ \text{“miner”},\ \text{“fee-bump”}\}) \\
\qquad\qquad\quad\ \ \land\ AllSigsPresent(tx) \\
\qquad\qquad \lor\ Print(\langle \text{“}\sim\text{TypeOK next\_block”},\ tx\rangle,\ \text{FALSE}) \\
\qquad\quad \land \forall\, tx \in mempool : \\
\qquad\qquad \lor\ \land\ TxConsistent(tx,\ \{\text{“mempool”}\}) \\
\qquad\qquad\quad\ \ \land\ AllSigsPresent(tx) \\
\qquad\qquad \lor\ Print(\langle \text{“}\sim\text{TypeOK mempool”},\ tx\rangle,\ \text{FALSE}) \\
\qquad\quad \land \forall\, tx \in shared\_knowledge :
\end{array}
$$

$\lor\ TxConsistent(tx,\ \{\text{“mempool”},\ \text{“miner”},\ \text{“fee-bump”},\ \text{“direct”}\})$
$\lor\ Print(\langle\text{“}\sim\text{TypeOK shared\_knowledge”},\ tx\rangle,\ \text{FALSE})$
$\land\ \forall\, p \in \text{DOMAIN}\ signers\_map :$
$\lor\ p \in participants \land \forall\, sig \in signers\_map[p] : sig \in all\_sigs$
$\lor\ Print(\langle\text{“}\sim\text{TypeOK signers\_map”},\ p\rangle,\ \text{FALSE})$

$OnlyWhenParticipantsAreRational\ \triangleq$
$PARTICIPANTS\_IRRATIONAL$
$\Rightarrow\ Assert(\text{FALSE},\ \text{“Not applicable when participants are not rational”})$

$NoConcurrentSecretKnowledge\ \triangleq$
$\land\ OnlyWhenParticipantsAreRational$
$\land\ \text{LET}\ SecretsShared\ \triangleq$
$(all\_secrets \cap \text{UNION}\ \{tx.ss : tx \in shared\_knowledge\})$
$\cup\ (\{secretBob\} \cap signers\_map[Alice])$
$\cup\ (\{secretAlice\} \cap signers\_map[Bob])$
$\text{IN}\quad Cardinality(SecretsShared) \leq 1$

$NoConflictingTransactions\ \triangleq$
$\text{LET}\ ConflictCheck(txs)\ \triangleq$
$\text{LET}\ ids\ \triangleq\ \{tx.id : tx \in txs\}$
$\text{IN}\quad \land\ Cardinality(ids) = Cardinality(txs)$
$\land \forall\, id \in ids : ConflictingSet(id) \cap ids = \{id\}$
$\text{IN}\quad \land\ ConflictCheck(\text{UNION}\ Range(blocks) \cup next\_block)$
$\land\ ConflictCheck(\text{UNION}\ Range(blocks) \cup mempool)$

$NoSingleParticipantTakesAll\ \triangleq$
$\land\ OnlyWhenParticipantsAreRational$
$\land \forall\, p \in participants :$
$\text{LET}\ txs\_to\_p\ \triangleq\ \{tx \in \text{UNION}\ Range(blocks) : tx.to = p\}$
$\text{IN}\quad Cardinality(\{tx.id : tx \in txs\_to\_p\}) \leq 1$

$TransactionTimelocksEnforced\ \triangleq$
$\land \forall\, tx \in mempool : Len(blocks) \geq TimelockExpirationHeight(tx.id)$
$\land\ STEALTHY\_SEND\_POSSIBLE$
$\Rightarrow \forall\, tx \in next\_block : Len(blocks) \geq TimelockExpirationHeight(tx.id)$

$ExpectedStateOnAbort\ \triangleq$
$SwapAborted$
$\Rightarrow\ \text{LET}\ ids\_left\ \triangleq\ \text{IF ENABLED}\ ContractAction\ \text{THEN}\ \{tx\_lock\_B\}\ \text{ELSE}\ \{\}$

$$\text{IN} \quad RemainingTransactions \subseteq \{tx\_spend\_B\} \cup ids\_left$$

$$ExpectedStateOnSuccess \triangleq$$
$$SwapSuccessful \Rightarrow \land \neg \text{ENABLED } ContractAction \lor Print(\langle \text{ENABLED } AliceAction, \text{ ENABLED } B$$
$$\land RemainingTransactions = \{\}$$
$$\land mempool = \{\}$$
$$\land next\_block = \{\}$$

Can use this invariant to check if certain state can be reached.
If the CounterExample invariant is violated, then the state has been reached.

$$CounterExample \triangleq \text{TRUE} \land \ldots$$

Temporal properties

$$ContractEventuallyFinished \triangleq \Diamond ContractFinished$$

Init & Next

$$Init \triangleq$$
$$\land blocks = \langle\rangle$$
$$\land per\_block\_enabled = \langle\rangle$$
$$\land next\_block = \{\}$$
$$\land mempool = \{\}$$
$$\land shared\_knowledge = \{\}$$
$$\land wont\_send = \{\}$$
$$\land signers\_map = [Alice \mapsto \{sigAlice, secretAlice\},$$
$$Bob \mapsto \{sigBob, secretBob\}]$$

$$Next \triangleq \lor \land ContractAction$$
$$\land UpdateEnabledPerBlock$$
$$\lor ContractFinished \land \text{UNCHANGED } fullState$$

$$Spec \triangleq Init \land \Box[Next]_{fullState} \land \text{WF}_{fullState}(Next)$$