MODULE *SASwap*

```
EXTENDS Naturals, Sequences, FiniteSets, TLC

CONSTANT PARTICIPANTS_IRRATIONAL   Can participants act irrational ?
ASSUME PARTICIPANTS_IRRATIONAL ∈ 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 ∈ BOOLEAN

 Operator to create transaction instances
Tx(id, ss, by, to, via) ≜
    [id ↦ id, ss ↦ ss, to ↦ to, by ↦ by, via ↦ via]

VARIABLE blocks               ⟨{Tx, ...}, ...⟩
VARIABLE next_block           {Tx, ...}
VARIABLE mempool              {Tx, ...}
VARIABLE shared_knowledge     {Tx, ...}
VARIABLE signers_map          [participant ↦ {allowed_sig, ...}]
VARIABLE per_block_enabled    ⟨{Tx, ...}, ...⟩

fullState  ≜ ⟨blocks, next_block, signers_map, shared_knowledge, mempool,
              per_block_enabled⟩
unchangedByMM  ≜ ⟨blocks, signers_map, shared_knowledge, mempool⟩
```

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

$Alice \triangleq \text{“Alice”}$
$Bob \triangleq \text{“Bob”}$
$participants \triangleq \{Alice, Bob\}$

$sigAlice \triangleq \text{“sigAlice”}$
$sigBob \triangleq \text{“sigBob”}$
$secretAlice \triangleq \text{“secretAlice”}$
$secretBob \triangleq \text{“secretBob”}$
$all\_secrets \triangleq \{secretAlice, secretBob\}$
$all\_sigs \triangleq \{sigAlice, sigBob, secretAlice, secretBob\}$

$tx\_lock\_A \triangleq \text{“tx\_lock\_A”}$
$tx\_lock\_B \triangleq \text{“tx\_lock\_B”}$
$tx\_success \triangleq \text{“tx\_success”}$
$tx\_refund\_1 \triangleq \text{“tx\_refund\_1”}$
$tx\_revoke \triangleq \text{“tx\_revoke”}$
$tx\_refund\_2 \triangleq \text{“tx\_refund\_2”}$
$tx\_timeout \triangleq \text{“tx\_timeout”}$
$tx\_spend\_A \triangleq \text{“tx\_spend\_A”}$
$tx\_spend\_B \triangleq \text{“tx\_spend\_B”}$
$tx\_spend\_success \triangleq \text{“tx\_spend\_success”}$
$tx\_spend\_refund\_1\_alice \triangleq \text{“tx\_spend\_refund\_1\_alice”}$
$tx\_spend\_refund\_1\_bob \triangleq \text{“tx\_spend\_refund\_1\_bob”}$
$tx\_spend\_revoke \triangleq \text{“tx\_spend\_revoke”}$
$tx\_spend\_refund\_2 \triangleq \text{“tx\_spend\_refund\_2”}$
$tx\_spend\_timeout \triangleq \text{“tx\_spend\_timeout”}$

$nLockTime \triangleq \text{“nLockTime”}$
$nSequence \triangleq \text{“nSequence”}$
$NoTimelock \triangleq [days \mapsto 0, type \mapsto nLockTime]$

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

$$ABS\_LK\_OFFSET \triangleq \text{CASE } BLOCKS\_PER\_DAY = 1 \rightarrow 2 \;\Box\; BLOCKS\_PER\_DAY = 2 \rightarrow 1 \;\Box\; \text{OTHER} \rightarrow 0$$

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.

$tx\_map \triangleq [$

'Contract' transactions – destinations are other transactions

$$tx\_lock\_A \mapsto [ds \mapsto \{tx\_success, tx\_refund\_1, tx\_revoke, tx\_spend\_A\}, ss \mapsto \{sigAlice\}],$$

$$tx\_lock\_B \mapsto [ds \mapsto \{tx\_spend\_B\}, ss \mapsto \{sigBob\}],$$

$$tx\_success \mapsto [ds \mapsto \{tx\_spend\_success\}, ss \mapsto \{sigAlice, sigBob, secretBob\}],$$

$$tx\_refund\_1 \mapsto [ds \mapsto \{tx\_spend\_refund\_1\_bob, tx\_spend\_refund\_1\_alice\}, ss \mapsto \{sigAlice, sigBob, secretAlice\}, lk \mapsto [days \mapsto ABS\_LK\_OFFSET + 1, type \mapsto nLockTime]],$$

$$tx\_revoke \mapsto [ds \mapsto \{tx\_refund\_2, tx\_timeout, tx\_spend\_revoke\}, ss \mapsto \{sigAlice, sigBob\}, lk \mapsto [days \mapsto ABS\_LK\_OFFSET + 2, type \mapsto nLockTime]],$$

$$tx\_refund\_2 \mapsto [ds \mapsto \{tx\_spend\_refund\_2\}, ss \mapsto \{sigAlice, sigBob, secretAlice\}, lk \mapsto [days \mapsto 1, type \mapsto nSequence]],$$

$$tx\_timeout \mapsto [ds \mapsto \{tx\_spend\_timeout\}, ss \mapsto \{sigAlice, sigBob\}, lk \mapsto [days \mapsto 2, type \mapsto nSequence]],$$

'Terminal' transactions – destinations are participants

$$\begin{array}{ll}
\mathit{tx\_spend\_A} & \mapsto [\mathit{ds} \mapsto \{\mathit{Alice}, \mathit{Bob}\}, \\
& \quad \mathit{ss} \mapsto \{\mathit{sigAlice}, \mathit{sigBob}\}], \\
\mathit{tx\_spend\_B} & \mapsto [\mathit{ds} \mapsto \{\mathit{Alice}, \mathit{Bob}\}, \\
& \quad \mathit{ss} \mapsto \{\mathit{secretAlice}, \mathit{secretBob}\}], \\
\mathit{tx\_spend\_success} & \mapsto [\mathit{ds} \mapsto \{\mathit{Bob}\}, \\
& \quad \mathit{ss} \mapsto \{\mathit{sigBob}\}], \\
\mathit{tx\_spend\_refund\_1\_bob} & \mapsto [\mathit{ds} \mapsto \{\mathit{Bob}\}, \\
& \quad \mathit{ss} \mapsto \{\mathit{sigAlice}, \mathit{sigBob}\}], \\
\mathit{tx\_spend\_refund\_1\_alice} & \mapsto [\mathit{ds} \mapsto \{\mathit{Alice}\}, \\
& \quad \mathit{ss} \mapsto \{\mathit{sigAlice}\}, \\
& \quad \mathit{lk} \mapsto [\mathit{days} \mapsto 1, \mathit{type} \mapsto \mathit{nSequence}]], \\
\mathit{tx\_spend\_revoke} & \mapsto [\mathit{ds} \mapsto \{\mathit{Alice}, \mathit{Bob}\}, \\
& \quad \mathit{ss} \mapsto \{\mathit{sigAlice}, \mathit{sigBob}\}], \\
\mathit{tx\_spend\_refund\_2} & \mapsto [\mathit{ds} \mapsto \{\mathit{Alice}\}, \\
& \quad \mathit{ss} \mapsto \{\mathit{sigAlice}\}], \\
\mathit{tx\_spend\_timeout} & \mapsto [\mathit{ds} \mapsto \{\mathit{Bob}\}, \\
& \quad \mathit{ss} \mapsto \{\mathit{sigBob}\}]
\end{array}$$

]

$$\mathit{all\_transactions} \triangleq \textsc{domain}\ \mathit{tx\_map}$$

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.

$$\mathit{first\_transaction} \triangleq \mathit{tx\_lock\_A}$$

$$\mathit{ConfirmedTransactions} \triangleq \{\mathit{tx.id} : \mathit{tx} \in \textsc{union}\ \mathit{Range}(\mathit{blocks})\}$$

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

$$\mathit{NextBlockConfirmedTransactions} \triangleq \\ \quad \mathit{ConfirmedTransactions} \cup \mathit{NextBlockTransactions}$$

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

$$\mathit{SentTransactions} \triangleq \mathit{ConfirmedTransactions} \cup \mathit{MempoolTransactions}$$

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

$$\begin{aligned}
&ContractTransactions \triangleq \\
&\quad \{id \in all\_transactions : \\
&\qquad \forall d \in tx\_map[id].ds : d \in all\_transactions\}
\end{aligned}$$

$$\begin{aligned}
&TerminalTransactions \triangleq \\
&\quad \{id \in all\_transactions : \\
&\qquad \forall d \in tx\_map[id].ds : d \in participants\}
\end{aligned}$$

$$\begin{aligned}
\text{ASSUME } \forall id \in all\_transactions : &\lor id \in TerminalTransactions \\
&\lor id \in ContractTransactions
\end{aligned}$$

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

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

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

$$Contract \triangleq \text{“Contract”}$$

$$\begin{aligned}
&DstSet(id) \triangleq \\
&\quad \text{IF } id \in ContractTransactions \text{ THEN } \{Contract\} \text{ ELSE } tx\_map[id].ds
\end{aligned}$$

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

$$\begin{aligned}
SingleDst(id) \triangleq \text{CASE } &id \in ContractTransactions \rightarrow Contract \\
\Box\ \ &Cardinality(tx\_map[id].ds) = 1 \\
&\quad \rightarrow \text{CHOOSE } d \in tx\_map[id].ds : \text{TRUE}
\end{aligned}$$

The set of transactions conflicting with the given transaction

$$\begin{aligned}
&ConflictingSet(id) \triangleq \\
&\quad \text{IF } id \in \text{DOMAIN } dependency\_map \\
&\quad\ \text{THEN } \{dep\_id \in \text{DOMAIN } dependency\_map : \\
&\qquad\qquad dependency\_map[dep\_id] = dependency\_map[id]\} \\
&\quad\ \text{ELSE } \{id\}
\end{aligned}$$

Transaction also conflicts with itself

$$\text{ASSUME } \forall id \in all\_transactions : id \in ConflictingSet(id)$$

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

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

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

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

All transactions that is not yet sent/confirmed, and have a chance to be.

```
$\mathit{RemainingTransactions} \triangleq$
    $((\mathit{all\_transactions} \setminus \mathit{ConfirmedTransactions}) \setminus \mathit{InvalidatedTransactions})$

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

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

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 $lk.type = nLockTime$
                 $\rightarrow lk.days * BLOCKS\_PER\_DAY$
             $\Box$  $lk.type = nSequence$
                 $\rightarrow$ 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 $hs \triangleq \{TimelockExpirationHeight(c\_id) :$
                  $c\_id \in ConflictingSet(id) \setminus \{id\}\}$
        $higher\_hs \triangleq \{h \in hs : h > TimelockExpirationHeight(id)\}$
     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 $id \in EnabledTransactions$
                  THEN (CHOOSE $en \in$ DOMAIN $per\_block\_enabled$ :
                            $\exists\, tx \in per\_block\_enabled[en] : tx.id = id)$
                        $+ MAX\_DAYS\_STALLING * BLOCKS\_PER\_DAY$
                  ELSE IF $h \neq UnreachableHeight$
                        THEN $h + MAX\_DAYS\_STALLING * BLOCKS\_PER\_DAY$
                        ELSE $0$
          ELSE $dl$
```

```
$SigsAvailable(id, sender, to) \triangleq$
    LET $secrets\_shared \triangleq$
            UNION $\{tx.ss \cap all\_secrets : tx \in shared\_knowledge\}$
        $sigs\_shared \triangleq$
            UNION $\{tx.ss : tx \in \{tx \in shared\_knowledge : \land tx.id = id$
                                                              $\land tx.to = to\}\}$
    IN  $sigs\_shared \cup secrets\_shared \cup signers\_map[sender]$

$DependencySatisfied(id, ids) \triangleq$
    $id \in$ DOMAIN $dependency\_map \Rightarrow dependency\_map[id] \in ids$

$IsSpendableTx(tx, other\_ids) \triangleq$
    $\land \{\} = ConflictingSet(tx.id) \cap other\_ids$
    $\land DependencySatisfied(tx.id, other\_ids)$
    $\land tx.ss \subseteq SigsAvailable(tx.id, tx.by, tx.to)$
    $\land Len(blocks) \geq TimelockExpirationHeight(tx.id)$
```

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 $revealSecret < Alice \mid Bob >$, $sigSecret < Alice \mid Bob >$
we simply filter out signatures of *tx_spend_B* before placing into shared knowledge

```
$ShareKnowledge(knowledge) \triangleq$
    LET $knowledge\_filtered \triangleq$
            $\{$IF $tx.id \neq tx\_spend\_B$ THEN $tx$ ELSE $[tx$ EXCEPT $!.ss = \{\}] :$
              $tx \in knowledge\}$
          shared_knowledge may not change here, callers need to check if they care
    IN  $shared\_knowledge' = shared\_knowledge \cup knowledge\_filtered$

$ShareTransactions(ids, by) \triangleq$
    LET $Ss(id) \triangleq (tx\_map[id].ss \cap signers\_map[by]) \setminus all\_secrets$
        $txs \triangleq \{Tx(id, Ss(id), by, SingleDst(id), \text{“direct”}) : id \in ids\}$
    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 ≜
    {tx ∈
     UNION
     {UNION
       {
         {
           Tx(id, tx_map[id].ss, sender, to, "enabled") : to ∈ DstSet(id)
         } : id ∈ RemainingTransactions
       } : sender ∈ participants
     } :  ∧ ¬∃ etx ∈ UNION Range(per_block_enabled) : etx.id = tx.id
          ∧ IsSpendableTx(tx, ConfirmedTransactions)
    }

SendTransactionToMempool(id, sender, to) ≜
    LET tx ≜ Tx(id, tx_map[id].ss, sender, to, "mempool")
     IN   ∧ IsSpendableTx(tx, SentTransactions)
          ∧ Len(blocks) < Deadline(id)
          ∧ mempool' = mempool ∪ {tx}
          ∧ 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) ≜
     ∧ STEALTHY_SEND_POSSIBLE
     ∧ LET tx ≜ Tx(id, tx_map[id].ss, sender, to, "miner")
        IN   ∧ IsSpendableTx(tx, NextBlockConfirmedTransactions)
             ∧ next_block' = next_block ∪ {tx}
```

```
$SendTransaction(id, sender, to) \triangleq$
      $\lor \land SendTransactionToMempool(id, sender, to)$
         $\land$ UNCHANGED $next\_block$
      $\lor \land SendTransactionToMiner(id, sender, to)$
         $\land$ UNCHANGED $\langle mempool, shared\_knowledge \rangle$

$SendSomeTransaction(ids, sender) \triangleq$
     LET $SendSome(filtered\_ids) \triangleq$
                 $\exists\, id \in filtered\_ids :$
                 $\exists\, to \in ($IF $id \in ContractTransactions$
                            THEN $\{Contract\}$
                            ELSE $tx\_map[id].ds \cap \{sender\}) :$
                     $SendTransaction(id, sender, to)$
           $terminal\_ids \triangleq ids \cap TerminalTransactions$
      IN   CASE $PARTICIPANTS\_IRRATIONAL$
                    $\rightarrow SendSome(ids)$  Irrational participants do no prioritization
               $\Box$   ENABLED $SendSome(terminal\_ids)$
                    $\rightarrow SendSome(terminal\_ids)$  Can send terminal $tx \Rightarrow$ do it immediately
               $\Box$   OTHER
                    $\rightarrow SendSome(ids \setminus terminal\_ids)$

$HasCustody(ids, participant) \triangleq$
     $\exists\, id \in ids : \exists\, tx \in$ UNION $Range(blocks) : tx.id = id \land tx.to = participant$

 Sharing secrets or keys has to occur before deadline to send $tx\_success$
$TooLateToShare \triangleq Len(blocks) \geq Deadline(tx\_success)$
```

Participant actions

Transactions *Alice* initially shares signatures on

$phase0\_to\_share\_Alice \triangleq \{tx\_revoke,\ tx\_timeout\}$

Transactions *Bob* initially shares signatures on

$phase0\_to\_share\_Bob \triangleq \{tx\_refund\_1,\ tx\_revoke,\ tx\_refund\_2,\ tx\_timeout\}$

Conditions to divide the contract execution into phases according to original spec

$$\begin{aligned}
&Phase\_3\_cond \triangleq tx\_lock\_B \in ConfirmedTransactions\\
&Phase\_2\_cond \triangleq tx\_lock\_A \in ConfirmedTransactions\\
&Phase\_1\_cond \triangleq\\
&\quad \land \forall\, id \in phase0\_to\_share\_Alice :\\
&\qquad \exists\, tx \in shared\_knowledge : tx.id = id \land sigAlice \in tx.ss\\
&\quad \land \forall\, id \in phase0\_to\_share\_Bob :\\
&\qquad \exists\, tx \in shared\_knowledge : tx.id = id \land sigBob \in tx.ss
\end{aligned}$$

$$\begin{aligned}
&InPhase\_3 \triangleq\\
&\quad \land Phase\_3\_cond
\end{aligned}$$

$$\begin{aligned}
&InPhase\_2 \triangleq\\
&\quad \land Phase\_2\_cond\\
&\quad \land \neg Phase\_3\_cond
\end{aligned}$$

$$\begin{aligned}
&InPhase\_1 \triangleq\\
&\quad \land Phase\_1\_cond\\
&\quad \land \neg Phase\_2\_cond\\
&\quad \land \neg Phase\_3\_cond
\end{aligned}$$

$$\begin{aligned}
&InPhase\_0 \triangleq\\
&\quad \land \neg Phase\_1\_cond\\
&\quad \land \neg Phase\_2\_cond\\
&\quad \land \neg Phase\_3\_cond
\end{aligned}$$

Helper operators to declutter the action expressions

$$\begin{aligned}
&NoSending \triangleq \text{UNCHANGED}\ \langle mempool,\ next\_block\rangle\\
&NoKeysShared \triangleq \text{UNCHANGED}\ signers\_map\\
&NoKnowledgeShared \triangleq \text{UNCHANGED}\ shared\_knowledge
\end{aligned}$$

```
$AliceAction \triangleq$
    LET $Send(ids) \triangleq SendSomeTransaction(ids, Alice)$
        $Share(ids) \triangleq ShareTransactions(ids, Alice)$
        $SafeToSend(id) \triangleq$
            CASE $PARTICIPANTS\_IRRATIONAL$
                  $\rightarrow$ TRUE  Unsafe $txs$ are $OK$ for irrational $Alice$
              $\square$ $id = tx\_refund\_1$  Do not send $refund\_1$ if $tx\_success$ was shared
                  $\rightarrow tx\_success \notin \{tx.id : tx \in shared\_knowledge\}$
              $\square$ $secretAlice \in tx\_map[id].ss$
                  Once $Alice$ received $secretBob$, should never send out $secretAlice$
                  $\rightarrow \lor\ secretBob \notin signers\_map[Alice]$
                      $\lor\ id = tx\_spend\_B$  unless this is a transaction to get $B$
                                          which does not in fact expose secrets
              $\square$ OTHER $\rightarrow$ TRUE
    IN  $\lor \land InPhase\_0$
          $\land Share(phase0\_to\_share\_Alice)$
          $\land NoSending \land NoKeysShared$
        $\lor \land InPhase\_1$
          $\land Send(\{tx\_lock\_A\})$
          $\land NoKeysShared$
        $\lor \land InPhase\_2$  Just waiting for $Bob$ to lock $B$
          $\land NoSending \land NoKnowledgeShared \land NoKeysShared$
        $\lor \land InPhase\_3$
          $\land \lor \land secretBob \in signers\_map[Alice]$  Bob gave $Alice$ his secret
                 $\land sigAlice \notin signers\_map[Bob]$  $Alice$ did not yet gave $Bob$ her key
                 $\land \neg TooLateToShare$
                 $\land signers\_map' = [signers\_map$  Give $Alice$'s key to $Bob$
                                      EXCEPT $![Bob] = @ \cup \{sigAlice\}]$
                 $\land NoSending \land NoKnowledgeShared$
             $\lor \land tx\_refund\_1 \notin SentTransactions$
                 $\land \neg TooLateToShare$
                 $\land Share(\{tx\_success\})$  $refund\_1$ not sent yet, can share
                 $\land NoSending \land NoKeysShared$
             $\lor \land Send(\{id \in RemainingTransactions : SafeToSend(id)\})$
                 $\land NoKeysShared$
```

```
$BobAction \triangleq$
    LET $Send(ids) \triangleq SendSomeTransaction(ids, Bob)$
        $Share(ids) \triangleq ShareTransactions(ids, Bob)$
        $tx\_success\_sigs \triangleq SigsAvailable(tx\_success, Bob, Contract)$
    IN  $\lor \land InPhase\_0$
          $\land Share(phase0\_to\_share\_Bob)$
          $\land NoSending \land NoKeysShared$
        $\lor \land InPhase\_1$ Just waiting for *Alice* to lock A
          $\land NoSending \land NoKnowledgeShared \land NoKeysShared$
        $\lor \land \lor InPhase\_2$
            $\lor InPhase\_3$
          $\land \lor \land sigAlice \in tx\_success\_sigs$
                    If *Bob* already knows *secretAlice*, he doesn't need to share *secretBob*
               $\land secretAlice \notin tx\_success\_sigs$
               $\land secretBob \notin signers\_map[Alice]$
               $\land \neg TooLateToShare$
               $\land signers\_map' = [signers\_map$ Give *secretBob* to *Alice*
                                    EXCEPT $![Alice] = @ \cup \{secretBob\}]$
               $\land NoSending \land NoKnowledgeShared$
            $\lor \land Send(RemainingTransactions)$
               $\land NoKeysShared$
```

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

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*
             $\{\} = DependencyChain(nbtx.id) \cap$
                    UNION $\{ConflictingSet(bmptx.id) : bmptx \in txs\_to\_bump\}\}$
            $\cup \{[bmptx$ EXCEPT $!.via =$ "fee-bump"$] : bmptx \in txs\_to\_bump\}$
```

Miner action

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

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

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

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

$MinerAction \triangleq IncludeTxIntoBlock \lor MineTheBlock$

Auxiliary action for soft-deadline tracking

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

High-level contract spec

First, the 'unnatural' cases.

For all transactions defined by the original spec
to be covered by the model, we need to also model the case
where *Alice* misbehaves by sending transactions containing her
secret after she gave $tx\_success\ to\ Bob. This\ behavior$
$also\ enables\ Bob\ to\ misbehave\ by\ failing\ to\ punish\ Alice\ s$
misbehavior, which results in *Bob* losing B.
The following four actions are needed to express all that.

$$
\begin{aligned}
&AliceLostByMisbehaving \triangleq \\
&\quad \land HasCustody(\{tx\_spend\_B\}, Bob) \\
&\quad \land HasCustody(\{tx\_spend\_refund\_1\_bob\}, Bob)
\end{aligned}
$$

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

$$
\begin{aligned}
&BobLostByBeingLateOnRefund\_2 \triangleq \\
&\quad \land HasCustody(\{tx\_spend\_B\}, Alice) \\
&\quad \land HasCustody(\{tx\_spend\_refund\_2\}, Alice)
\end{aligned}
$$

$$
\begin{aligned}
&SwapUnnaturalEnding \triangleq \\
&\quad \lor AliceLostByMisbehaving \\
&\quad \lor BobLostByBeingLateOnRefund\_1 \\
&\quad \lor BobLostByBeingLateOnRefund\_2
\end{aligned}
$$

The normal, 'natural' cases.

```
$SwapSuccessful \triangleq$
    $\land HasCustody(\{tx\_spend\_B\}, Alice)$
    $\land \lor HasCustody(\{tx\_spend\_A, tx\_spend\_success,$
                        $tx\_spend\_timeout, tx\_spend\_revoke\}, Bob)$
      $\lor \land PARTICIPANTS\_IRRATIONAL$
        $\land HasCustody(\{tx\_spend\_refund\_1\_bob\}, Bob)$

$SwapAborted \triangleq$
    $\land HasCustody(\{tx\_spend\_A, tx\_spend\_refund\_1\_alice, tx\_spend\_refund\_2\}, Alice)$
    $\land HasCustody(\{tx\_spend\_B\}, Bob)$

$SwapTimedOut \triangleq$
    $\land tx\_spend\_timeout \in ConfirmedTransactions$
      Alice can't claim tx_spend_B on timeout
    $\land secretBob \notin signers\_map[Alice]$
    $\land secretBob \notin \text{UNION } \{tx.ss : tx \in shared\_knowledge\}$
```

All possible endings of the contract

```
$ContractFinished \triangleq \lor SwapSuccessful$
                    $\lor SwapAborted$
                    $\lor SwapTimedOut$
                    $\lor PARTICIPANTS\_IRRATIONAL \land SwapUnnaturalEnding$
```

Actions in the contract when it is not yet finished. Separated into dedicated operator to be able to test ENABLED *ContractAction*

```
$ContractAction \triangleq$
    $\lor AliceAction$                $\land \text{UNCHANGED } blocks$
    $\lor BobAction$                  $\land \text{UNCHANGED } blocks$
    $\lor \text{IF } MempoolMonitorActionRequired$
      $\text{THEN } MempoolMonitorAction \land \text{UNCHANGED } unchangedByMM$
      $\text{ELSE } MinerAction$          $\land \text{UNCHANGED } signers\_map$
```

Invariants

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

```
$ConsistentPhase \triangleq$
    $\text{LET } phases \triangleq \langle InPhase\_0, InPhase\_1, InPhase\_2, InPhase\_3 \rangle$
     $\text{IN } \quad Cardinality(\{i \in \text{DOMAIN } phases : phases[i]\}) = 1$

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

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

$NoUnexpectedTransactions \triangleq$
     $\land OnlyWhenParticipantsAreRational$
     $\land tx\_spend\_refund\_1\_bob \notin SentTransactions$

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

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

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

```
$ExpectedStateOnTimeout \triangleq$
    $SwapTimedOut \Rightarrow RemainingTransactions \subseteq \{tx\_lock\_B,\ tx\_spend\_B\}$

$ExpectedStateOnFinish \triangleq$
    $ContractFinished \Rightarrow$
        $\textsc{if}\ SwapTimedOut$
         $\textsc{then let}\ ids\_left \triangleq \textsc{if enabled}\ ContractAction\ \textsc{then}\ \{tx\_lock\_B\}\ \textsc{else}\ \{\}$
                $\textsc{in}$   $\land RemainingTransactions = \{tx\_spend\_B\} \cup ids\_left$
                      $\land MempoolTransactions \subseteq ids\_left$
                      $\land NextBlockTransactions \subseteq ids\_left$
         $\textsc{else}$  $\land \neg\textsc{enabled}\ ContractAction$
               $\land RemainingTransactions = \{\}$
               $\land mempool = \{\}$
               $\land next\_block = \{\}$

$NoTransactionsRemaining\_iff\_NotTimedOut \triangleq$
    $\{\} = RemainingTransactions \equiv ContractFinished \land \neg SwapTimedOut$
```

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

```
$CounterExample \triangleq \textsc{true}\ \land \ldots$
```

Temporal properties

```
$ContractEventuallyFinished \triangleq \Diamond ContractFinished$
```

Init & Next

$$
\begin{aligned}
&Init \triangleq \\
&\quad \land\ blocks = \langle\rangle \\
&\quad \land\ per\_block\_enabled = \langle\rangle \\
&\quad \land\ next\_block = \{\} \\
&\quad \land\ mempool = \{\} \\
&\quad \land\ shared\_knowledge = \{\} \\
&\quad \land\ signers\_map = [Alice \mapsto \{sigAlice,\ secretAlice\}, \\
&\qquad\qquad\qquad\qquad\ \ Bob \ \ \mapsto \{sigBob,\ secretBob\}]
\end{aligned}
$$

$$
\begin{aligned}
Next \triangleq\ &\lor \land\ ContractAction \\
& \land\ UpdateEnabledPerBlock \\
&\lor\ ContractFinished \land \textsc{unchanged}\ fullState
\end{aligned}
$$

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