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.

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

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

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

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

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

$SentTransactions \triangleq ConfirmedTransactions \cup MempoolTransactions$

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

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

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

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

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"

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

```
The CASE statement has no 'OTHER ' clause - only single dst is expected
SingleDst(id) ≜ CASE id ∈ ContractTransactions → Contract
                  □  Cardinality(tx_map[id].ds) = 1
                       → CHOOSE d ∈ tx_map[id].ds : TRUE

The set of transactions conflicting with the given transaction
ConflictingSet(id) ≜
    IF id ∈ DOMAIN dependency_map
     THEN {dep_id ∈ DOMAIN dependency_map :
             dependency_map[dep_id] = dependency_map[id]}
     ELSE {id}

Transaction also conflicts with itself
ASSUME ∀ id ∈ all_transactions : id ∈ ConflictingSet(id)

ConfirmationHeight(id) ≜
    CHOOSE bn ∈ DOMAIN blocks : ∃ tx ∈ 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) ≜
    IF id ∈ DOMAIN dependency_map
     THEN {id} ∪ 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) ≜
    LET dependants ≜ tx_map[id].ds \ participants
     IN   IF dependants = {}
            THEN {id}
            ELSE dependants ∪ UNION {AllDependants(d_id) : d_id ∈ dependants}

All transactions that cannot ever become valid because other, conflicting
transactions were confirmed befor them
InvalidatedTransactions ≜
    UNION {{c_id} ∪ AllDependants(c_id)      : c_id ∈
```

```
                UNION {ConflictingSet(id) \ {id} : id ∈ ConfirmedTransactions}}

 All transactions that is not yet sent/confirmed, and have a chance to be.
RemainingTransactions ≜
    ((all_transactions \ ConfirmedTransactions) \ InvalidatedTransactions)

Timelock(id) ≜ IF "lk" ∈ DOMAIN tx_map[id] THEN tx_map[id].lk ELSE NoTimelock

UnreachableHeight ≜ 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) ≜
    LET lk ≜ Timelock(id)
     IN  CASE lk.type = nLockTime
                  → lk.days * BLOCKS_PER_DAY
           □  lk.type = nSequence
                  → IF dependency_map[id] ∈ 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) ≜
    LET hs ≜ {TimelockExpirationHeight(c_id) :
                c_id ∈ (ConflictingSet(id) \ {id}) \ wont_send}
        higher_hs ≜ {h ∈ 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) ≜
    LET dl ≜ Deadline(id)
        h ≜ TimelockExpirationHeight(id)
     IN  IF dl = UnreachableHeight
         THEN IF id ∈ EnabledTransactions
                THEN (CHOOSE en ∈ 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 $\{tx.ss : tx \in \{tx \in shared\_knowledge : \wedge tx.id = id$
$\wedge tx.to = to\}\}$
IN $sigs\_shared \cup secrets\_shared \cup signers\_map[sender]$

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

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

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 $Ss(id) \triangleq (tx\_map[id].ss \cap signers\_map[by]) \setminus all\_secrets$
$txs \triangleq \{Tx(id, Ss(id), by, SingleDst(id),$ "direct"$) : id \in ids\}$
IN $\wedge ShareKnowledge(txs)$

$\land$ *shared_knowledge'* $\neq$ *shared_knowledge* not a new *knowledge* $\Rightarrow$ fail

Txs enabled at the current cycle, used to update *per_block_enabled* vector

$$
\begin{array}{l}
\mathit{NewlyEnabledTxs} \triangleq \\
\quad \{tx \in \\
\quad \text{UNION} \\
\quad \{\text{UNION} \\
\quad \{ \\
\quad \{ \\
\quad\quad \mathit{Tx}(\mathit{id},\ \mathit{tx\_map}[\mathit{id}].\mathit{ss},\ \mathit{sender},\ \mathit{to},\ \text{“enabled”}) : \mathit{to} \in \mathit{DstSet}(\mathit{id}) \\
\quad \} : \mathit{id} \in \mathit{RemainingTransactions} \\
\quad \} : \mathit{sender} \in \mathit{participants} \\
\quad \} : \land \neg\exists\, \mathit{etx} \in \text{UNION}\ \mathit{Range}(\mathit{per\_block\_enabled}) : \mathit{etx}.\mathit{id} = \mathit{tx}.\mathit{id} \\
\quad\quad \land \mathit{IsSpendableTx}(\mathit{tx},\ \mathit{ConfirmedTransactions}) \\
\quad \}
\end{array}
$$

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

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

$$
\begin{array}{l}
\mathit{SendTransactionToMiner}(\mathit{id},\ \mathit{sender},\ \mathit{to}) \triangleq \\
\quad \land \mathit{STEALTHY\_SEND\_POSSIBLE} \\
\quad \land \text{LET}\ \mathit{tx} \triangleq \mathit{Tx}(\mathit{id},\ \mathit{tx\_map}[\mathit{id}].\mathit{ss},\ \mathit{sender},\ \mathit{to},\ \text{“miner”}) \\
\quad\quad \text{IN} \quad \land \mathit{IsSpendableTx}(\mathit{tx},\ \mathit{NextBlockConfirmedTransactions}) \\
\quad\quad\quad\quad \land \mathit{next\_block}' = \mathit{next\_block} \cup \{\mathit{tx}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{SendTransaction}(\mathit{id},\ \mathit{sender},\ \mathit{to}) \triangleq \\
\quad \lor \land \mathit{SendTransactionToMempool}(\mathit{id},\ \mathit{sender},\ \mathit{to}) \\
\quad\quad \land \text{UNCHANGED}\ \mathit{next\_block} \\
\quad \lor \land \mathit{SendTransactionToMiner}(\mathit{id},\ \mathit{sender},\ \mathit{to})
\end{array}
$$

```
        ∧ 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}
```

$\lor\ id = tx\_spend\_B$ unless this is a transaction to get B which does not in fact expose secrets

$\square$ OTHER $\rightarrow$ TRUE

IN $\lor \land Send(\{id \in RemainingTransactions : SafeToSend(id)\})$
$\land NoKeysShared$
$\lor \land \{tx\_lock\_A,\ tx\_lock\_B\} \subseteq ConfirmedTransactions$
$\land \neg(\{tx\_success,\ tx\_timeout\} \subseteq \{tx.id : tx \in shared\_knowledge\})$
$\land Share(\{tx\_success,\ tx\_timeout\})$
$\land NoSending \land NoKeysShared$
$\lor \land \{tx\_lock\_A,\ tx\_lock\_B\} \subseteq ConfirmedTransactions$
$\land \{tx\_success,\ tx\_refund\_1,\ tx\_timeout\} \subseteq \{tx.id : tx \in shared\_knowledge\}$
$\land secretBob \in signers\_map[Alice]$ 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 $![Bob] = @ \cup \{sigAlice\}]$
$\land NoSending \land NoKnowledgeShared$

$BobAction \triangleq$
LET $Send(ids) \triangleq SendSomeTransaction(ids,\ Bob)$
$Share(ids) \triangleq ShareTransactions(ids,\ Bob)$
$tx\_success\_sigs \triangleq SigsAvailable(tx\_success,\ Bob,\ Contract)$
IN $\lor \land Send(RemainingTransactions)$
$\land NoKeysShared \land$ UNCHANGED $wont\_send$
$\lor \land tx\_refund\_1 \notin \{tx.id : tx \in shared\_knowledge\}$
$\land tx\_success \notin SentTransactions$
$\land tx\_timeout \notin SentTransactions$
$\land Share(\{tx\_refund\_1\})$
$\land wont\_send' = wont\_send \cup \{tx\_timeout\}$
$\land NoSending \land NoKeysShared$
$\lor \land \{tx\_lock\_A,\ tx\_lock\_B\} \subseteq ConfirmedTransactions$
$\land \{tx\_success,\ tx\_refund\_1,\ tx\_timeout\} \subseteq \{tx.id : tx \in shared\_knowledge\}$
$\land tx\_success \notin SentTransactions$
$\land tx\_timeout \notin SentTransactions$
$\land secretAlice \notin tx\_success\_sigs$
$\land secretBob \notin signers\_map[Alice]$
$\land \neg TooLateToShare$

$$
\begin{aligned}
&\land\ signers\_map' = [signers\_map \ \text{Give } secretBob \text{ to } Alice \\
&\qquad\qquad \text{EXCEPT } ![Alice] = @ \cup \{secretBob\}] \\
&\land\ wont\_send' = wont\_send \setminus \{tx\_timeout\} \\
&\land\ NoSending \land NoKnowledgeShared
\end{aligned}
$$

$$
\begin{aligned}
&MempoolMonitorActionRequired \triangleq \\
&\quad \exists\, tx \in mempool :\ \land\ Len(blocks) + 1 = Deadline(tx.id) \\
&\qquad\qquad\qquad\qquad\ \land\ tx.id \notin NextBlockTransactions
\end{aligned}
$$

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

$$
\begin{aligned}
&MempoolMonitorAction \triangleq \\
&\quad \text{LET } tx \triangleq \text{CHOOSE } tx \in mempool : Len(blocks) + 1 = Deadline(tx.id) \\
&\qquad txs\_to\_bump \triangleq \{tx\} \cup \{dptx \in mempool : \\
&\qquad\qquad\qquad\qquad \land\ tx.id \in \text{DOMAIN } dependency\_map \\
&\qquad\qquad\qquad\qquad \land\ dptx.id = dependency\_map[tx.id] \\
&\qquad\qquad\qquad\qquad \land\ dptx.id \notin NextBlockTransactions\} \\
&\quad \text{IN} \quad next\_block' = \\
&\qquad \{nbtx \in next\_block : \ \text{conflicting } txs \text{ are expunged from } next\_block \\
&\qquad\ \{\} = DependencyChain(nbtx.id) \cap \\
&\qquad\qquad\qquad \text{UNION } \{ConflictingSet(bmptx.id) : bmptx \in txs\_to\_bump\}\} \\
&\qquad \cup \{[bmptx \text{ EXCEPT } !.via = \text{“fee-bump”}] : bmptx \in txs\_to\_bump\}
\end{aligned}
$$

Miner action

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

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

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

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

$MinerAction \triangleq IncludeTxIntoBlock \vee MineTheBlock$

Auxiliary action for soft-deadline tracking

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

High-level contract spec

$BobLostByBeingLateOnSuccess \triangleq$
$\quad \wedge HasCustody(\{tx\_spend\_B\},\ Alice)$

$\land$ $HasCustody(\{tx\_spend\_refund\_1\}, Alice)$

$SwapUnnaturalEnding \triangleq BobLostByBeingLateOnSuccess$

The normal, 'natural' cases.

$SwapSuccessful \triangleq$
$\land$ $HasCustody(\{tx\_spend\_B\}, Alice)$
$\land$ $HasCustody(\{tx\_spend\_A, tx\_spend\_success, tx\_spend\_timeout\}, Bob)$

$SwapAborted \triangleq$
$\land$ $HasCustody(\{tx\_spend\_A, tx\_spend\_refund\_1\}, Alice)$
$\land$ $\lor$ $HasCustody(\{tx\_spend\_B\}, Bob)$
$\lor$ $tx\_lock\_B \notin SentTransactions$

All possible endings of the contract
$ContractFinished \triangleq \lor SwapSuccessful$
$\lor SwapAborted$
$\lor PARTICIPANTS\_IRRATIONAL \land SwapUnnaturalEnding$

Actions in the contract when it is not yet finished. Separated into dedicated operator to be able to test ENABLED *ContractAction*
$ContractAction \triangleq$
$\lor AliceAction$ $\land$ UNCHANGED $\langle blocks, wont\_send \rangle$
$\lor BobAction$ $\land$ UNCHANGED $blocks$
$\lor$ IF $MempoolMonitorActionRequired$
THEN $MempoolMonitorAction$ $\land$ UNCHANGED $unchangedByMM$
ELSE $MinerAction$ $\land$ UNCHANGED $\langle signers\_map, wont\_send \rangle$

Invariants

$TypeOK \triangleq$
LET $TxConsistent(tx, vias) \triangleq \land tx.id \in all\_transactions$
$\land tx.ss \subseteq tx\_map[tx.id].ss$
$\land tx.to \in DstSet(tx.id)$
$\land tx.by \in participants$
$\land tx.via \in vias$
$AllSigsPresent(tx) \triangleq tx.ss = tx\_map[tx.id].ss$
$SigConsistent(sig) \triangleq \land sig.id \in all\_transactions$
$\land sig.s \in all\_sigs$

$$
\begin{array}{ll}
 & \land\, sig.ds \subseteq participants \\
 & \quad \cup \text{DOMAIN } dependency\_map \\
\text{IN} & \land \forall\, tx \in \text{UNION } Range(blocks) : \\
 & \quad \lor \land TxConsistent(tx, \{\text{“mempool”, “miner”, “fee-bump”}\}) \\
 & \quad\quad \land AllSigsPresent(tx) \\
 & \quad \lor Print(\langle \text{“}\sim\text{TypeOK blocks”}, tx\rangle, \text{FALSE}) \\
 & \land \forall\, tx \in \text{UNION } Range(per\_block\_enabled) : \\
 & \quad \lor \land TxConsistent(tx, \{\text{“enabled”}\}) \\
 & \quad\quad \land AllSigsPresent(tx) \\
 & \quad \lor Print(\langle \text{“}\sim\text{TypeOK blocks”}, tx\rangle, \text{FALSE}) \\
 & \land \forall\, tx \in next\_block : \\
 & \quad \lor \land TxConsistent(tx, \{\text{“mempool”, “miner”, “fee-bump”}\}) \\
 & \quad\quad \land AllSigsPresent(tx) \\
 & \quad \lor Print(\langle \text{“}\sim\text{TypeOK next\_block”}, tx\rangle, \text{FALSE}) \\
 & \land \forall\, tx \in mempool : \\
 & \quad \lor \land TxConsistent(tx, \{\text{“mempool”}\}) \\
 & \quad\quad \land AllSigsPresent(tx) \\
 & \quad \lor Print(\langle \text{“}\sim\text{TypeOK mempool”}, tx\rangle, \text{FALSE}) \\
 & \land \forall\, tx \in shared\_knowledge : \\
 & \quad \lor TxConsistent(tx, \{\text{“mempool”, “miner”, “fee-bump”, “direct”}\}) \\
 & \quad \lor Print(\langle \text{“}\sim\text{TypeOK shared\_knowledge”}, tx\rangle, \text{FALSE}) \\
 & \land \forall\, p \in \text{DOMAIN } signers\_map : \\
 & \quad \lor p \in participants \land \forall\, sig \in signers\_map[p] : sig \in all\_sigs \\
 & \quad \lor Print(\langle \text{“}\sim\text{TypeOK signers\_map”}, p\rangle, \text{FALSE})
\end{array}
$$

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

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

$$
NoConflictingTransactions \triangleq
$$

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

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

$$
\begin{array}{l}
\mathit{TransactionTimelocksEnforced} \triangleq \\
\quad \land \forall\, \mathit{tx} \in \mathit{mempool} : \mathit{Len}(\mathit{blocks}) \geq \mathit{TimelockExpirationHeight}(\mathit{tx}.\mathit{id}) \\
\quad \land \mathit{STEALTHY\_SEND\_POSSIBLE} \\
\qquad \Rightarrow \forall\, \mathit{tx} \in \mathit{next\_block} : \mathit{Len}(\mathit{blocks}) \geq \mathit{TimelockExpirationHeight}(\mathit{tx}.\mathit{id})
\end{array}
$$

$$
\begin{array}{l}
\mathit{ExpectedStateOnAbort} \triangleq \\
\quad \mathit{SwapAborted} \\
\quad \Rightarrow \text{LET } \mathit{ids\_left} \triangleq \text{IF ENABLED } \mathit{ContractAction} \text{ THEN } \{\mathit{tx\_lock\_B}\} \text{ ELSE } \{\} \\
\qquad \text{IN} \quad \mathit{RemainingTransactions} \subseteq \{\mathit{tx\_spend\_B}\} \cup \mathit{ids\_left}
\end{array}
$$

$$
\begin{array}{l}
\mathit{ExpectedStateOnSuccess} \triangleq \\
\quad \mathit{SwapSuccessful} \Rightarrow \land \neg\text{ENABLED } \mathit{ContractAction} \lor \mathit{Print}(\langle\text{ENABLED } \mathit{AliceAction}, \text{ ENABLED } B \\
\qquad\qquad\qquad\qquad \land \mathit{RemainingTransactions} = \{\} \\
\qquad\qquad\qquad\qquad \land \mathit{mempool} = \{\} \\
\qquad\qquad\qquad\qquad \land \mathit{next\_block} = \{\}
\end{array}
$$

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

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

Temporal properties

$$\mathit{ContractEventuallyFinished} \triangleq \Diamond \mathit{ContractFinished}$$

Init & Next

$$
\begin{array}{l}
\mathit{Init} \triangleq \\
\quad \land \mathit{blocks} = \langle\rangle
\end{array}
$$

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

$$
\begin{aligned}
Next\ \triangleq\ &\lor\ \land\ ContractAction \\
&\qquad\ \land\ UpdateEnabledPerBlock \\
&\quad\ \lor\ ContractFinished \land \text{UNCHANGED}\ fullState
\end{aligned}
$$

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