Message-passing and the Double Satisfaction Problem

In EUTxO model

Polina Vinogradova

EUTxO LEDGER

Transition system specification

$$\text{utxo}' := (\mathsf{getORefs}\ tx \ntriangleleft utxo) \cup \mathsf{mkOuts}\ tx$$

$$\text{ApplyTx}\ \frac{\mathsf{checkTx}\ (slot,\ utxo,\ tx)}{slot \vdash (utxo) \xrightarrow[\text{LEDGER}]{tx} (utxo')}$$

```
checkTx : Slot -> UTxO -> Tx -> Bool
```

Checks that the transaction is valid to apply to the given UTxO in the given slot, including :

```
slot ∈ validityInterval tx
∧
inputs tx ≠ { }
∧
sumValue (inputs tx) + mint tx =
    sumValue (outputs tx)

…
```

Structured contract

Structured contract $\mathtt{STRUC}$ consists of :

The transition system $\mathtt{STRUC}$ of the type :

$$\_ \vdash \_ \xrightarrow[\text{STRUC}]{\_} \_ \in \mathbb{P}\,(\star \times \mathsf{State}_{\mathsf{STRUC}} \times \mathsf{Input}_{\mathsf{STRUC}} \times \mathsf{State}_{\mathsf{STRUC}})$$

Together with :

- ledger representation π
- transaction representation π_{Tx}
- **proof obligation :**

$$\sim> \frac{slot \vdash (\,utxo\,) \xrightarrow[\text{LEDGER}]{tx} (\,utxo'\,)}{\star \vdash (\pi\ utxo) \xrightarrow[\text{STRUC}]{\pi_{\mathrm{Tx}}\ tx} (\pi\ \boldsymbol{utxo'})}$$

Stateful Contract Communication

- Is the **dependencies** of a state transition on **actions** of the carrying transaction
  - i.e. one script contains a constraint that another script validates in the same transaction, usually with specific inputs

- **e.g.** constraint that a payout must be made to some address :

$$\text{MakePayout}\ \frac{\begin{array}{c}\texttt{(otherScript, someAssets, \_)} \in \texttt{outputs tx} \\ \dots\end{array}}{* \vdash \texttt{s} \xrightarrow{\texttt{tx}} \texttt{s'}}$$

Current Communication Model

- “specifying other actions a carrying transaction must perform when updating contract state”
    - is **ad-hoc**
        - **no convention for modelling communication** within or between structured contracts
    - each dependency might have **its own set of dependencies**
        - which might have its own dependencies, etc.
    - no principled way to **associate action** satisfying a contract update with that **update**
        - The problem with **double satisfaction**

Motivation

- **Structured contracts framework** gives us a principled way to model **stateful computation** on the EUTxO ledger

- We want to be **principled about intra- and inter-contract communication**
    - to study its behaviour and properties!

- We want to have **control over the set of script dependencies**
    - At least limit sequences of dependencies longer than 1

Plan

1. What do messages look like?

2. How does message-passing work?

3. Usecases :
    i. Memoization with messages
    ii. Structured contracts that use message-passing

4. Double satisfaction

5. Payouts with messages

Message-passing

- Messages are **pieces of data and asset bundles** that have a producer/**sender** and consumer/**receiver**
    - With some notion of “sending”

- Standard approach to contract communication in **account-based ledger models**

- We propose a **message-passing scheme** in the EUTxO model
    - Allows us to formalize certain kinds of **asynchronous communication**
    - Implemented via certain kinds of dependencies

Record Containing Message Data

Msg

- **inUTxO** : OutputRef
- **msgIx** : Ix
- **msgTo** : Output
- **msgFrom** : Output
- **msgValue** : Value
- **msgData** : Data

- output reference that must be spent in order to mint the message
- index that, together with `inUTxO`, uniquely identifies the message
- recipient
- sender
- assets sent by message
- data sent by message

Messages in the UTxO

UTxO set

outputRef1 ⟼ (msgValidator, msg1 + assets1, _)

outputRef2 ⟼ (msgValidator, msg2 + assets2, _)

...

outputRefN ⟼ (msgValidator, msgN + assetsN, _)

NFT message tokens :

(msgPolicy, msg) ⟼ 1

msg is an **Msg** record encoded as Data

assets ≥ msgValue msg

Message-passing Specification

– (1) compute messages being sent and received by getting all redeemers which decode as pairs of an instruction and message data

$$\mathit{sndMsgs} := [\,(\mathit{msg},\, i) \mid i \leftarrow (\mathsf{toList}\ (\mathsf{inputs}\ \mathit{tx})),\ (\mathit{sr},\, \mathit{msg}) \leftarrow \mathsf{fromData}\ (\mathsf{redeemer}\ i),\ \mathit{sr} = \mathsf{send}\,]$$
$$\mathit{rcvMsgs} := [\,(\mathit{msg},\, i) \mid i \leftarrow (\mathsf{toList}\ (\mathsf{inputs}\ \mathit{tx})),\ (\mathit{sr},\, \mathit{msg}) \leftarrow \mathsf{fromData}\ (\mathsf{redeemer}\ i),\ \mathit{sr} = \mathsf{receive}\,]$$

– (2) check that no new messages are duplicates

$$\mathsf{noDups}\ (\mathsf{map}\ \mathsf{getMsgRef}\ \mathit{sndMsgs}\ +\!+\ \mathsf{map}\ \mathsf{getMsgRef}\ \mathit{rcvMsgs})$$
$$\mathsf{noDups}\ (\mathsf{map}\ \mathsf{getMsgRef}\ \mathit{sndMsgs}\ +\!+\ \mathsf{map}\ \mathsf{inUTxO}\ \mathit{msgs})$$

– (3) compute what new message-token containing outputs are being created

$$\mathit{newOuts} := \{\,(o,\, \mathit{msg}) \mid o \in \mathsf{outputs}\ \mathit{tx},\ \mathsf{msgTkn}\ \mathit{msg} \subseteq \mathsf{value}\ o\,\}$$

– (4) check that all the messages are correctly defined : correct sender, sender has correct redeemer, outputref is spent, one message per output, output with message has correct validator and sufficient value

$$\forall\,(o,\, \mathit{msg}) \in \mathit{newOuts},\ (\mathit{msg},\, (\mathit{inf}, \mathsf{msgFrom}\ \mathit{msg}, \_)) \in \mathit{sndMsgs}$$
$$\wedge\ \mathsf{inUTxO}\ \mathit{msg} = \mathit{inf}\ \wedge\ \{\,t \subseteq \mathsf{value}\ o \mid \mathsf{dom}\ t = \{\mathsf{msgsTT}\}\,\} = \mathsf{msgTkn}\ \mathit{msg}$$
$$\wedge\ \mathsf{validator}\ o = \mathsf{msgsVal}\ \wedge\ \mathsf{value}\ o \geq \mathsf{msgValue}\ \mathit{msg}$$

– (5) compute what message-outputs are being spent

$$\mathit{usedInputs} := \{\,(i,\, \mathit{msg}) \mid i \in \mathsf{inputs}\ \mathit{tx},\ \mathsf{msgTkn}\ \mathit{msg} \subseteq \mathsf{value}\ (\mathsf{output}\ i)\,\}$$

– (6) check that all the messages are correctly consumed : the receiver is correct, and has correct redeemer, and message exists

$$\forall\,(i,\, \mathit{msg}) \in \mathit{usedInputs},\ (\mathit{msg},\, (\_, \mathsf{msgTo}\ \mathit{msg}, \_)) \in \mathit{rcvMsgs}\ \wedge\ \mathit{msg} \in \mathit{msgs}$$

– (7) check minting and burning of message tokens :

$$\Sigma_{(\mathit{msg},\, \_)\, \in\, \mathit{sndMsgs}}\ \mathsf{msgTkn}\ \mathit{msg}\ +\ \Sigma_{(\mathit{msg},\, \_)\, \in\, \mathit{rcvMsgs}}\ (-1) * (\mathsf{msgTkn}\ \mathit{msg})$$
$$= \Sigma_{\mathsf{msgsTT} \mapsto \mathit{tkns}\, \in\, \mathsf{mint}\ \mathit{tx}}\, \mathsf{msgsTT} \mapsto \mathit{tkns}$$

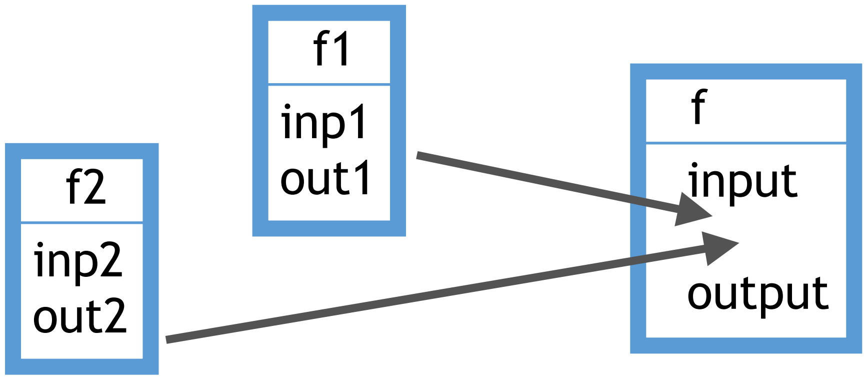
$$\text{Process}\ \frac{}{\vdash (\mathit{msgs}) \xrightarrow[\text{MSGS}]{\mathit{tx}} \big(\,(\mathit{msgs} \setminus (\mathsf{map}\ \mathsf{fst}\ \mathit{rcvMsgs})) \cup (\mathsf{map}\ \mathsf{fst}\ \mathit{sndMsgs})\,\big)} \quad (3)$$

Usecase : Memoization

- Technique used primarily to **speed up computer programs**

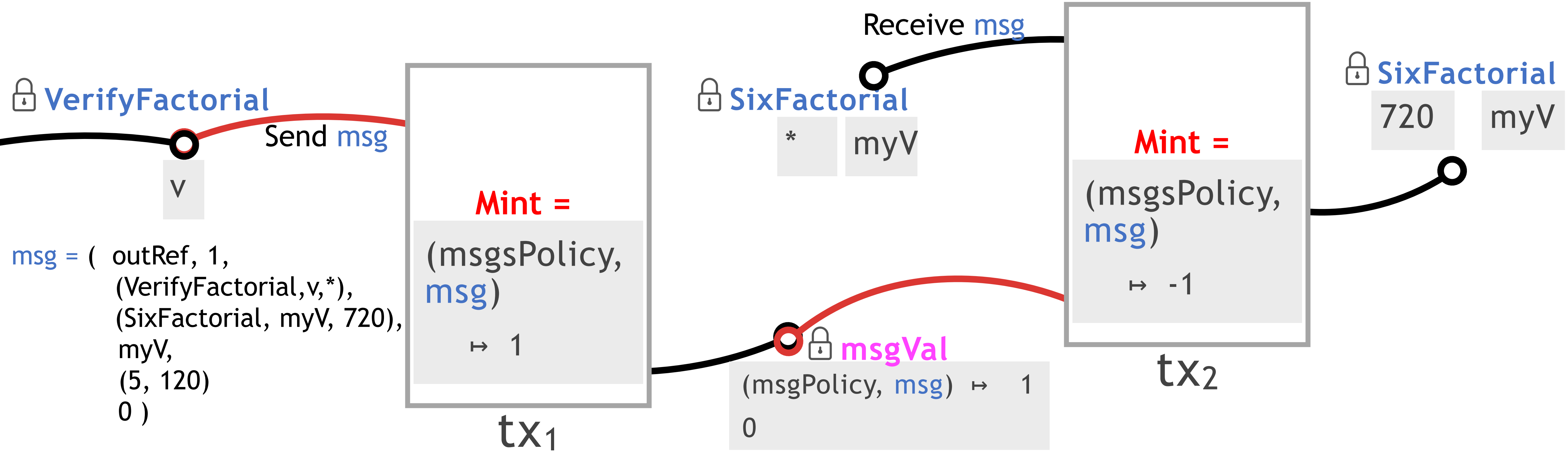
- Involves **storing the results of expensive function calls** to pure functions and returning the cached result when the same inputs occur again

- **Memoization for ledger scripts** :
    - Large computations may be **over the limit** of allowable total **ExUnits**
    - **Divide up a large script** computation `f` into smaller functions `f1, …, fk`
    - Use **pre-computed results** of those functions to compute `f`
    - `f1, …, fk` and `f'` (version of `f` that uses pre-computed `f1, …, fk`) can be in **distinct transactions**
    - Can **re-use the cached computations**



Memoization with Message-Passing

Example : a message that provides a proof that `factorial 5 = 120` that SixFactorial may use



Memoization with Messages

Given some function :

```
myFunction : MyInput  MyOutput
```

`checkMyFunction` requires that `tx` mints a particular **message token** **`m`**

`m` must contain encoded `(fIn, fOut)`

$$
\begin{array}{l}
[\![\mathsf{checkMyFunction}]\!]\ (\_,\ r,\ (tx,\ i))\ := \\
\qquad\qquad \text{– If } r \text{ is a send-message redeemer sending } m \\
\qquad\qquad \text{–and message data is an input-output pair} \\
\text{if } \mathsf{fromData}\ r\ \neq\ \star\ \wedge\ \mathsf{fromData}_{IO}\ (\mathsf{msgData}\ m) \neq \star, \\
\qquad \text{– } m \text{ is uniquely identified by } i \\
\qquad \wedge\ \mathsf{inUTxO}\ m\ =\ \mathsf{outputRef}\ i \\
\qquad \text{– } m \text{ is from this script} \\
\qquad \wedge\ \mathsf{msgFrom}\ m\ =\ \mathsf{output}\ i \\
\qquad \text{– no minimum sent value} \\
\qquad \wedge\ \mathsf{msgValue}\ m\ =\ 0 \\
\qquad \text{– message token with message } m \text{ is minted} \\
\qquad \wedge\ \mathsf{msgTkn}\ m\ \subseteq\ \mathsf{mint}\ tx \\
\qquad \text{– check myFunction computation} \\
\qquad \wedge\ \mathsf{myFunction}\ fIn\ =\ fOut \\
\qquad \textbf{where} \\
\qquad\quad [(\mathsf{send},\ \mathsf{Msg}\ m)]\ =\ \mathsf{fromData}\ r \\
\qquad\quad (fIn,\ fOut)\ =\ \mathsf{fromData}_{IO}\ (\mathsf{msgData}\ m) \\
\text{else,} \\
\qquad \mathsf{False}
\end{array}
$$

Memoization with Messages

`useMyFunction` requires that `tx` burns **`m`**
i.e. `tx` **includes proof artefact**

Input-output pair `(fIn, fOut)` in `m` is used in compute `useMyFunction`

$[\![\mathsf{useMyFunction}]\!]\ (d,\ r,\ (tx,\ i)) :=$

– If r is a receive-message redeemer sending m

– and m-data from message is an input-output pair

if $\mathsf{fromData}\ r\ \neq\ \star\ \wedge\ \mathsf{fromData}_{IO}\ (\mathsf{msgData}\ m) \neq \star$,

– m is from the script computing myFunction

$\wedge\ \mathsf{msgFrom}\ m\ =\ \mathsf{checkMyFunction}$

– m is sent to this script

$\wedge\ \mathsf{msgTo}\ m\ =\ \mathsf{output}\ i$

– message token with message m is burned

$\wedge\ (-1) * (\mathsf{msgTkn}\ m)\ \subseteq\ \mathsf{mint}\ tx$

– use myFunction computation output from message t

$\wedge\ \mathsf{checkStuff}\ d\ r\ \ (tx,\ i)\ (fIn,\ fOut)$

where

$[(\mathsf{receive},\ \mathsf{Msg}\ m)]\ =\ \mathsf{fromData}\ r$

$(fIn,\ fOut)\ =\ \mathsf{fromData}_{IO}\ (\mathsf{msgData}\ m)$

else,

$\mathsf{checkOtherStuff}\ d\ r\ \ (tx,\ i)$

Memoization with Message-Passing

Formal statement of the result

Lemma (Verified input-output pairs). For any $(s,\ u,\ tx, u') \in \mathsf{LEDGER}$, with $\pi\, u \neq \star$ and $(i,\ (\mathsf{useMyFunction}, v, d),\ r) \in$ inputs tx, such that

$$\begin{aligned}
[(\mathsf{receive},\ \mathsf{Msg}\ m)] &= \mathsf{fromData}\ r \\
(fIn,\ fOut) &= \mathsf{fromData}_{IO}\ (\mathsf{msgData}\ m) \\
\mathsf{msgFrom}\ m &= (\mathsf{checkMyFunction}, _, _)
\end{aligned}$$

necessarily $\mathsf{myFunction}\ fIn = fOut$, and $\mathsf{msgTo}\ m = (\mathsf{useMyFunction}, v, d)$.

Usecase : Using Message-Passing

- **Given a structured contract implemented by script `s`, substitute**

  constraint on the **execution of a script `s`** with inputs `ins`

  ↓

  constraint on **consuming a message** with sender `s` with inputs `ins`

- **Combine** a stateful contract **`STRUC`** and the **`MSGS`** contract into a **single contract**

- **Express** a constraint such as **`msgTkn ∈ mint tx`** on the **transaction** as a constraint on the **update to the message state**

Using Message-Passing

- To construct messages correctly, contracts need to **express constraints** on the specific **scripts** implementing them, their **output** references, and their **inputs**

- So, we talk about only a **certain class of structured contracts**

- Let `F : Output → Bool`
  - `c : UTxO → Bool`

$$\mathsf{State} := \{i \mapsto o \in u \mid i \in \mathsf{Input},\ o \in \mathsf{F}, u \in \mathsf{UTxO}, \mathsf{c}\ u\}$$

$$\pi_{\mathsf{F,c}}\ u := \begin{cases} \{i \mapsto o \in u \mid o \in \mathsf{F}\} & \text{if } \mathsf{c}\ u \\ \star & \text{otherwise} \end{cases}$$

$$\pi_{\mathsf{Tx}} := \mathsf{id}$$

Using Message-Passing

- Construct a contract with combined message and STRUC state :

$$
\begin{aligned}
\pi_{\mathsf{State}-M} &:= (\pi_{\mathsf{F,c}}, \pi_{\mathsf{Msg}}) \\
\pi_{\mathsf{Tx}-M} &:= \mathsf{id}_{\mathsf{Tx}} \\
\mathsf{STRUC}_{\mathsf{MSGS}} &:= \{\, (\star, (s,m), tx, (s',m')) \;|\; (\star, s, tx, s') \in \mathsf{STRUC}, (\star, m, tx, m') \in \mathsf{MSGS} \,\}
\end{aligned}
$$

Definition (uses message-passing)

- **STRUC** uses message passing whenever one of these sets is nonempty for some step :

$$
\begin{aligned}
\mathsf{getFromSTRUCmsgs} &: \mathbb{P}\ \mathsf{Msg} \to \mathbb{P}\ \mathsf{Msg} \\
\mathsf{getFromSTRUCmsgs}\ \mathit{msgs} &:= \{\ m \ \mid\ m\ \in \mathit{msgs},\ \mathsf{F}\ (\mathsf{msgFrom}\ m)\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{getToSTRUCmsgs} &: \mathbb{P}\ \mathsf{Msg} \to \mathbb{P}\ \mathsf{Msg} \\
\mathsf{getToSTRUCmsgs}\ \mathit{msgs} &:= \{\ m \ \mid\ m\ \in \mathit{msgs},\ \mathsf{F}\ (\mathsf{msgTo}\ m)\ \}
\end{aligned}
$$

Definition : Payouts

Payouts are special kinds of messages :

$$\text{getPayouts}\,(\star,(s,m),tx,(s'm')) \;:=\; \{msg \in \text{getFromSTRUCmsgs}\,(m' \setminus m) \;\mid\; \text{msgValue}\,msg > 0 \land \neg\,(\text{F}\,(\,\text{msgTo}\,msg))\,\}$$

- `STRUC` **makes payouts** when above set is not empty for some step
- `getPayouts` is a function of the message state `m, m′` and `F` **only**

Example : Payouts

Does not affect PAYOUT state, messages sent/ received as usual

$$\frac{\_ \vdash (m) \xrightarrow[\text{MSGS}]{tx} (\boldsymbol{m'}) \qquad (i, o, \_) \notin \mathsf{inputs}\ tx}{\vdash \begin{pmatrix} \{i \mapsto o\} \\ m \end{pmatrix} \xrightarrow[\text{PAYOUT-MSGS}]{tx} \begin{pmatrix} \{i \mapsto o\} \\ \boldsymbol{m'} \end{pmatrix}} \ \boxed{\text{MSGSOnly}}$$

Sent messages must include payout & remove amount from contract

$$ms := (i, 1, (\mathsf{recipient}, \mathsf{v}, \star), (\mathsf{payout}, \mathsf{NFT} + a, \star), v, \star)$$

$$ms \in m' \qquad ms \notin m \qquad \mathsf{v} \leq a$$

$$(tx, ix) \mapsto (\mathsf{payout}, \mathsf{NFT} + (a - \mathsf{v}), \star) \in \mathsf{mkOuts}\ tx$$

$$\exists\, inp \in \mathsf{inputs}\ tx, \mathsf{outputRef}\ inp = i \ \wedge\ \mathsf{redeemer\ inp} = \{(\mathsf{send}, ms)\}$$

$$\frac{\_ \vdash (m) \xrightarrow[\text{MSGS}]{tx} (\boldsymbol{m'})}{\vdash \begin{pmatrix} \{i \mapsto (\mathsf{payout}, \mathsf{NFT} + a, \star)\} \\ m \end{pmatrix} \xrightarrow[\text{PAYOUT-MSGS}]{tx} \begin{pmatrix} \{(tx, ix) \mapsto (\mathsf{payout}, \mathsf{NFT} + (a - \mathsf{v}), \star)\} \\ \boldsymbol{m'} \end{pmatrix}} \ \boxed{\text{PayoutV}}$$

Example : Payouts

Constraints on the `MSGS` state

$$\text{MSGSOnly}\ \frac{\_ \vdash (m) \xrightarrow[\text{MSGS}]{tx} (m') \qquad (i, o, \_) \notin \mathsf{inputs}\ tx}{\vdash \begin{pmatrix} \{i \mapsto o\} \\ m \end{pmatrix} \xrightarrow[\text{PAYOUT-MSGS}]{tx} \begin{pmatrix} \{i \mapsto o\} \\ m' \end{pmatrix}}$$

$$ms := (i, 1, (\mathsf{recipient}, \mathsf{v}, \star), (\mathsf{payout}, \mathsf{NFT} + a, \star), v, \star)$$

$$ms \in m' \qquad ms \notin m \qquad v \leq a$$

$$(tx, ix) \mapsto (\mathsf{payout}, \mathsf{NFT} + (a - \mathsf{v}), \star) \in \mathsf{mkOuts}\ tx$$

$$\exists\, inp \in \mathsf{inputs}\ tx, \mathsf{outputRef}\ inp = i \ \wedge\ \mathsf{redeemer\ inp} = \{(\mathsf{send}, ms)\}$$

$$\text{PayoutV}\ \frac{\_ \vdash (m) \xrightarrow[\text{MSGS}]{tx} (m')}{\vdash \begin{pmatrix} \{i \mapsto (\mathsf{payout}, \mathsf{NFT} + a, \star)\} \\ m \end{pmatrix} \xrightarrow[\text{PAYOUT-MSGS}]{tx} \begin{pmatrix} \{(tx, ix) \mapsto (\mathsf{payout}, \mathsf{NFT} + (a - \mathsf{v}), \star)\} \\ m' \end{pmatrix}}$$

Double Satisfaction

Consider the following examples :

(i) **Authorization tokens** : the transaction must contain in its inputs a special token, the ability to spend which constitutes proof that a particular contract state update is authorized

- **OK** to present **one authorization** token to satisfy multiple actions controlled by scripts that require it in a single transaction

(ii) **Payouts** : to perform a specified state update, the transaction must make a payout to a : Script by including an output containing value v, with address a

- **NOT OK** to make **one payout** output `(a, v, _)` to satisfy multiple actions controlled by scripts that require payout in a single transaction

Double Satisfaction

Define function : $\mathsf{s}\,\mathsf{STRUC} = \{\ (s, s')\ \mid\ \exists\, i,\ (\star, s, i, s') \in \mathsf{STRUC}\ \}$

Definition (transition constraint). A *constraint* of a transition system STRUC is a subset

$$\mathsf{C} \subseteq \{\star\} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State}$$

such that $\mathsf{STRUC} \subseteq \mathsf{C}$. A constraint is *strict* when $\mathsf{STRUC} \subsetneq \mathsf{C}$.

Definition (double satisfaction). A system STRUC is *vulnerable to double satisfaction* with respect to a strict constraint C whenever there exists another contract STRUC', with $\mathsf{STRUC} \subseteq \mathsf{STRUC}'$ and $\mathsf{s}\,\mathsf{STRUC} = \mathsf{s}\,\mathsf{STRUC}'$,

such that $\mathsf{STRUC} \subseteq \mathsf{STRUC}' \cap \mathsf{C} \subsetneq \mathsf{STRUC}'$.

Double Satisfaction Example

Example (TOGGLE *with extra constraint*). Consider the following specification of a TOGGLE contract, with $\mathsf{State} = \mathbb{B}$, and $\mathsf{Input} = (\mathsf{toggle} \cup \{\star\}) \times \mathsf{Interval}[\mathsf{Slot}]$.

$$\text{DoNothing}\ \frac{}{\vdash (\,x\,) \xrightarrow[\text{TOGGLE}]{(\star,\_)} (\,x\,)}$$

$$\text{Toggle}\ \frac{5 \leq k < j \leq 9}{\vdash (\,x\,) \xrightarrow[\text{TOGGLE}]{(toggle,[j,k])} (\,\neg\, \boldsymbol{x}\,)}$$

Double Satisfaction Example

Define a constraint `C` :

```
C (*,_,(t,[j,k]),_) := (t = toggle) ⇒ (5 ≤ j < k ≤ 9)
```

Compute

```
s STRUC = { x ↦ x, x ↦ ¬ x }
```

Define `STRUC'` by removing `(5 ≤ j < k ≤ 9)` from rule `Toggle` of `STRUC`

We have : `STRUC ⊆ STRUC'`

```
STRUC = STRUC' ∩ C ⊊ STRUC'
```

⇒ **`STRUC` is vulnerable to DS with respect to `C`**

Double Satisfaction Example

Counterintuitive

Standard solution approach to DS is the following constraint :

> "**No other contracts** are allowed to be in the transaction except for the validator locking a given state machine"

This is **vulnerable to double satisfaction!**

It is **incidentally not possible to include other scripts** in the transaction that can be satisfied by this constraint

This approach excludes some good usecases anyway : (

Double Satisfaction Lemma

DS-free contracts

A system `STRUC` is **not vulnerable to double satisfaction** with respect to any constraint whenever for any `(s, i)`, there exists an s′ such that `(⋆, s, i, s′ ) ∈ STRUC`

Message Payouts Constraint

$$\texttt{Pays}\ (\star,\ (s,\ m),\ tx,\ (s'm')) := \forall\ \texttt{STRUC}' \supseteq \texttt{STRUC},\ s\ \texttt{STRUC} = s\ \texttt{STRUC}',$$
$$\quad \forall(\star,\ (s,\ m),\ tx',\ (s',\ m')) \in \texttt{STRUC}',$$

$$\quad \texttt{getPayoutsSTRUC}(\star,(s,m),tx,(s',m'))$$
$$\qquad = \texttt{getPayoutsSTRUC}'\ (\star,(s,m),tx',(s',m'))$$

`Pays` says :

“Payout messages required for a transition between two states are the same for any contract and its more permissive version”

$\forall$ `step, Pays step`

- Because `getPayoutsSTRUC` is a function of `s, s'`, and `F` only
- $\texttt{STRUC}' \cap \texttt{Pays} = \texttt{STRUC}'$

Message Payouts Lemma

Given a filter $\mathtt{F}$, a UTxO constraint $\mathtt{c}$, and a structured contract $(\pi_{\mathtt{F},\mathtt{c}},\ \pi_{\mathtt{Tx}},\ \mathtt{STRUC})$, $\mathtt{STRUC}_{\mathtt{MSGS}}$ is not vulnerable to DS with respect to the constraint $\mathtt{Pays}$.

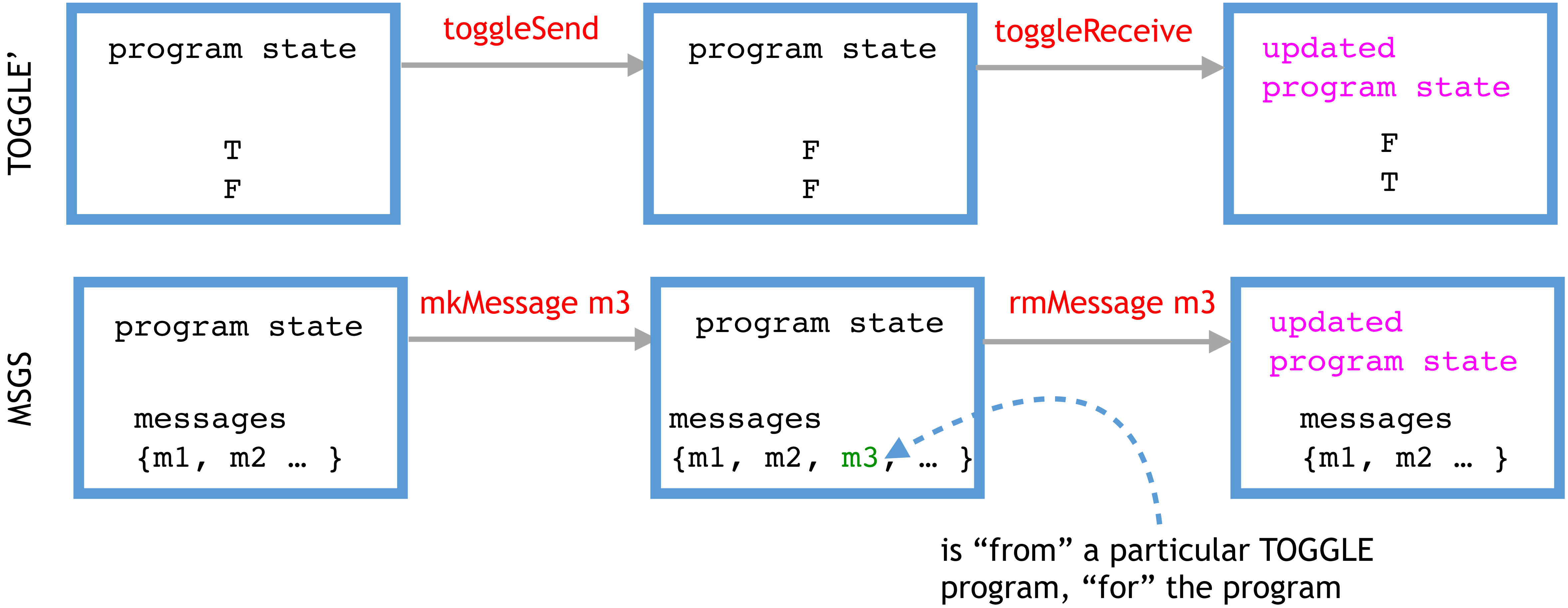
Conclusion

- Message passing is a model of script and structured contract communication
    - Alternative to to script interaction
    - More principled/amenable to formal verification
    - Asynchronous
    - Concurrent (multiple messages can be produced/consumed by one or more contracts in a single transaction)
    - Kind of a “flat” UTxO ledger in itself

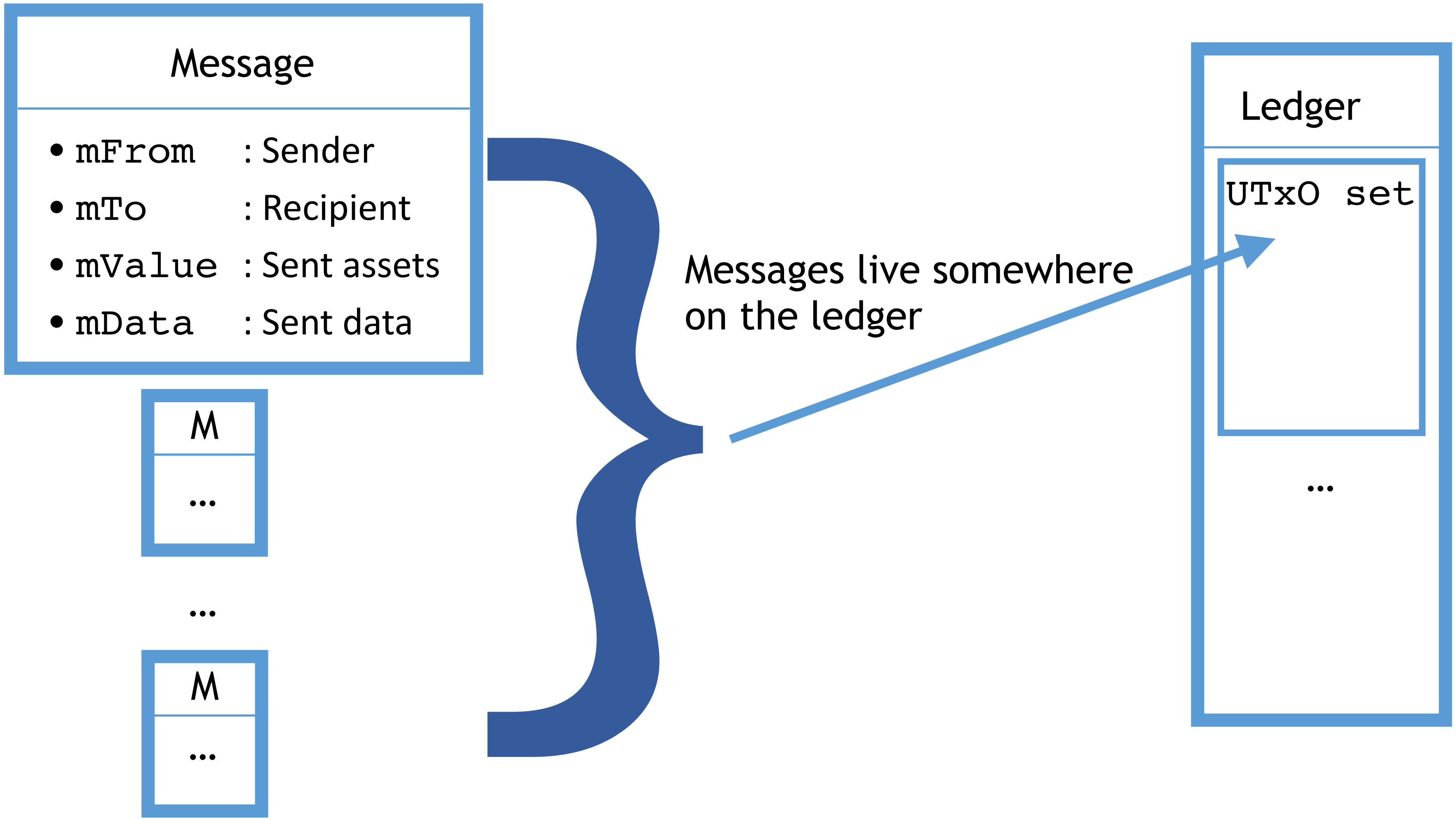
- Double satisfaction is a problem of associating a constraint with a contract that imposes that constraint
    - State is already associated with a specific contract
    - Constraints on state are therefore associated with the specific contract

Applications

Asynchronous or **partial** contract execution.

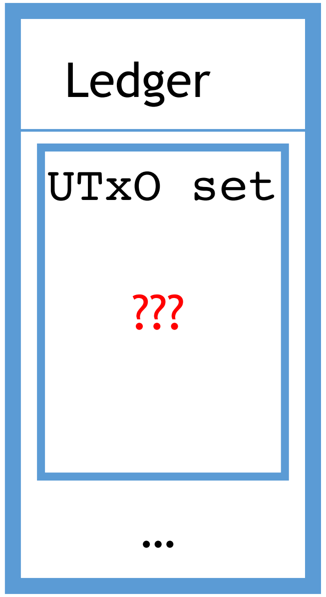


What is a message?



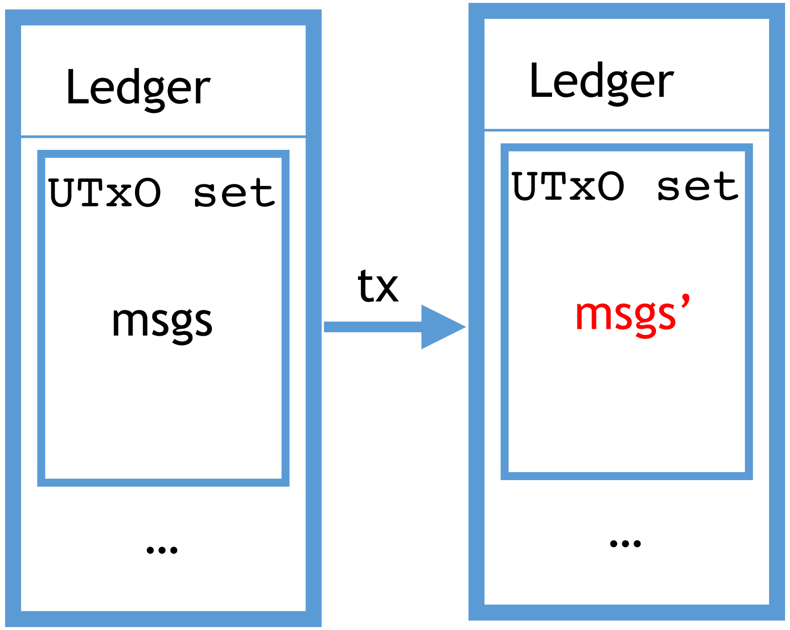
What we need to specify

1. How are messages recorded on the ledger?



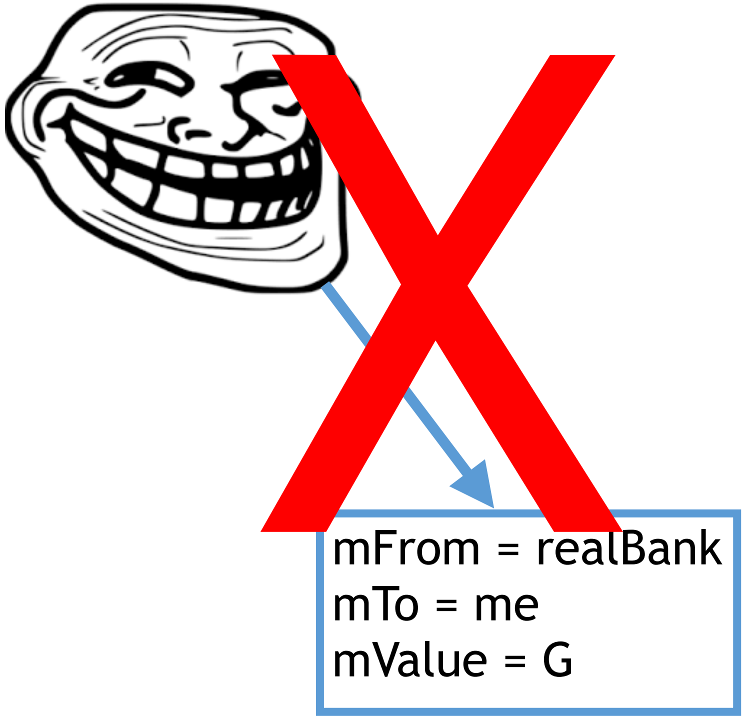
2. How to update the collection of messages?

Produce some messages
Consume some messages



Looks like a state machine!

3. How do we ensure message legitimacy?



1. How can we record messages?

UTxO Set

...

```
txOutRef   ↦   ( messageUTxOLockingScriptHash,
                 mValue (msg) +  (messageMintingPolicy ↦ (msg ↦ 1)),
                 _ )
```

...

The **token name** is the **`msg`** (really, it's the hash of the data in the message)

The `messageMintingPolicy` together with `messageUTxOLockingScriptHash` ensure that all legitimate message tokens on the ledger are in this kind of a UTxO entry

A **legitimate message** on the ledger is any token that has this minting policy

2. How do we update messages?

Producing messages

Tx

Mint

```
(messageMintingPolicy ↦ (msg1 ↦ 1))
 …
```

…

Consuming messages

Tx

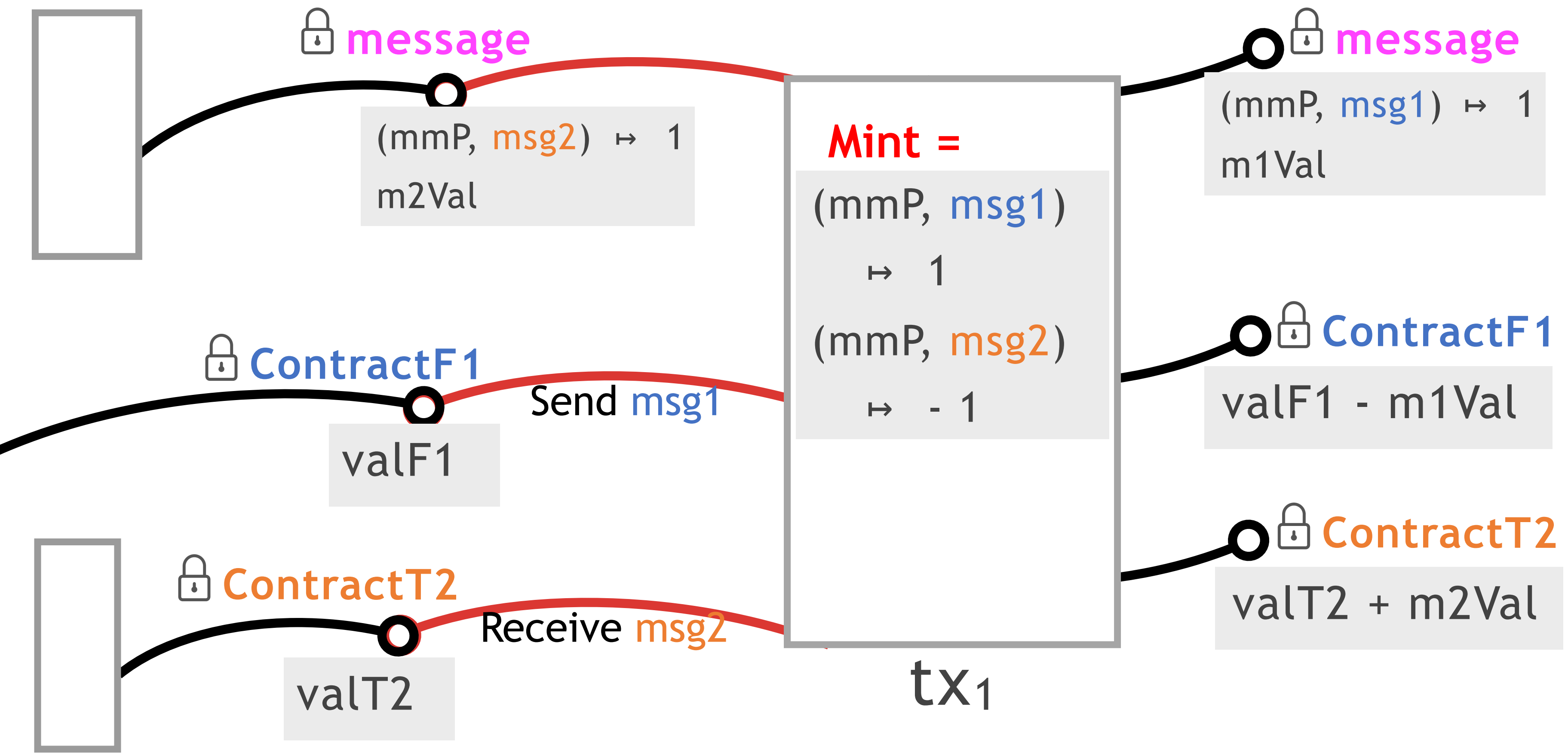
Mint

```
(messageMintingPolicy ↦ (msg2 ↦ -1))
 …
```

…

`messageMintingPolicy` does all the hard work of making sure these tokens got placed in the right kind of UTxOs

Message-updating transaction



mmP = messageMintingPolicy

message =
messageUTxOLockingScriptHash

msg1 = (ContractF1,
ContractT1, m1Val, _)

msg2 = (ContractF2,
ContractT2, m2Val, _)

3. Message Send/Receive validation

- Message sending/receiving must be "**validated by the owner of the `mFrom/mTo` credentials**"
  - This is not a well-defined concept, but let us see what we can do

- In the case that the credential is a **PK**, this validation is a **signature** on the transaction

- In the **general** case that the credential is a **script**, there is no way to identify any "owner in control of the credentials"
  - this makes the general case of (possibly non-unique) scripts hard for message-passing
  - could do something with `TxOutRef`'s as unique identifiers here

- For **persistent contracts with unique identifiers** - eg. state machines with unique output-locking hashes (and thread tokens),
  - **Validation** can be defined as "**taking a step with a specific redeemer**" (eg. `Send msg`)
  - Same idea for receiving/consuming a message

3. Message Send/Receive validation

- Message are enforced to be **unique across the transaction** in which they are produced/consumed

- This lets us make an association **via the `Send msg` redeemer** between :
  - Message `msg` being minted
    <-> Step (`startingStateOfmFrom` $\xrightarrow{\texttt{Send msg}}$ `stateAfterSendStepOfmFrom`)

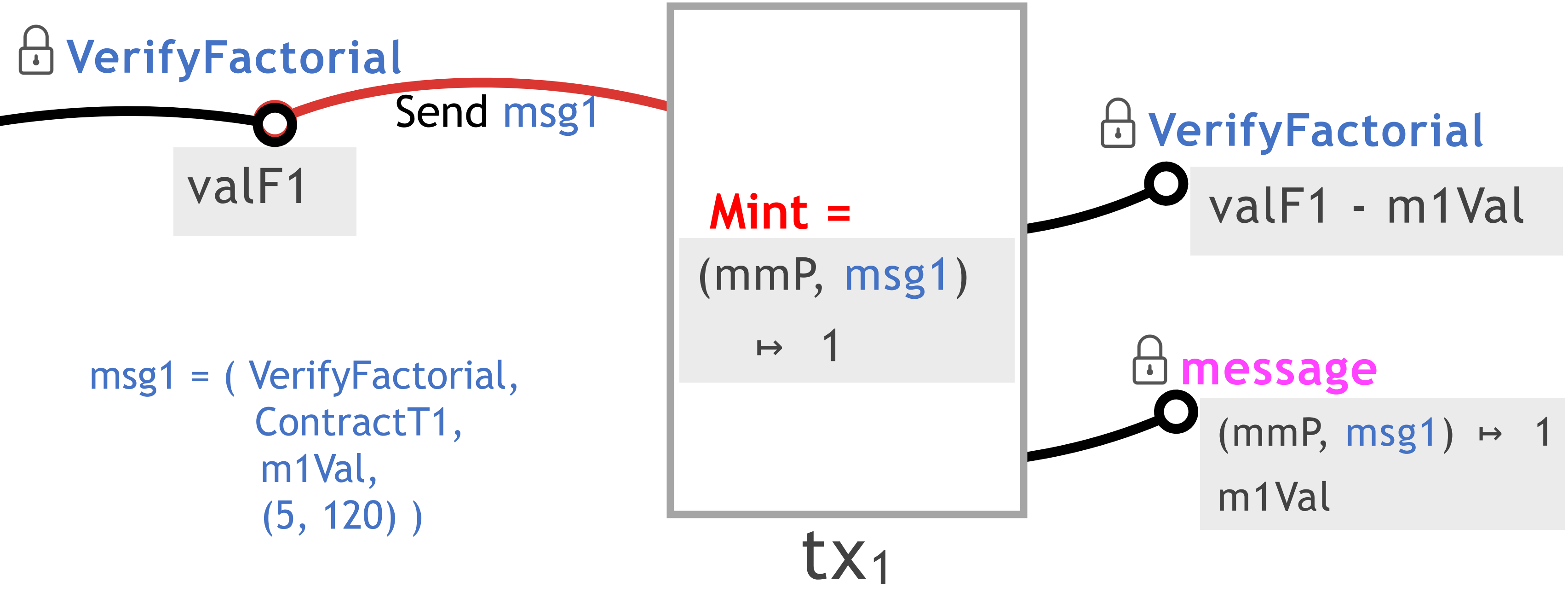
  - Same for … `Receive msg` … `mTo` …

- For **PK** sender/recipient, the association is 1-to-n
  - `pk` <-> msg's with `mFrom = pk`
  - Same for `mTo = pk`
  - This is fine because each such a message is produced/consumed by the owner of the PK
    - They can do whatever checks they want
    - Each send/receive is associated with explicit permission from key owner

What can messages do?

Given a function `f`, messages can represent verified input-output pairs `(x, f(x))` for `f`

Example : a message that provides a proof that `factorial 5 = 120` that ContractT1 may use



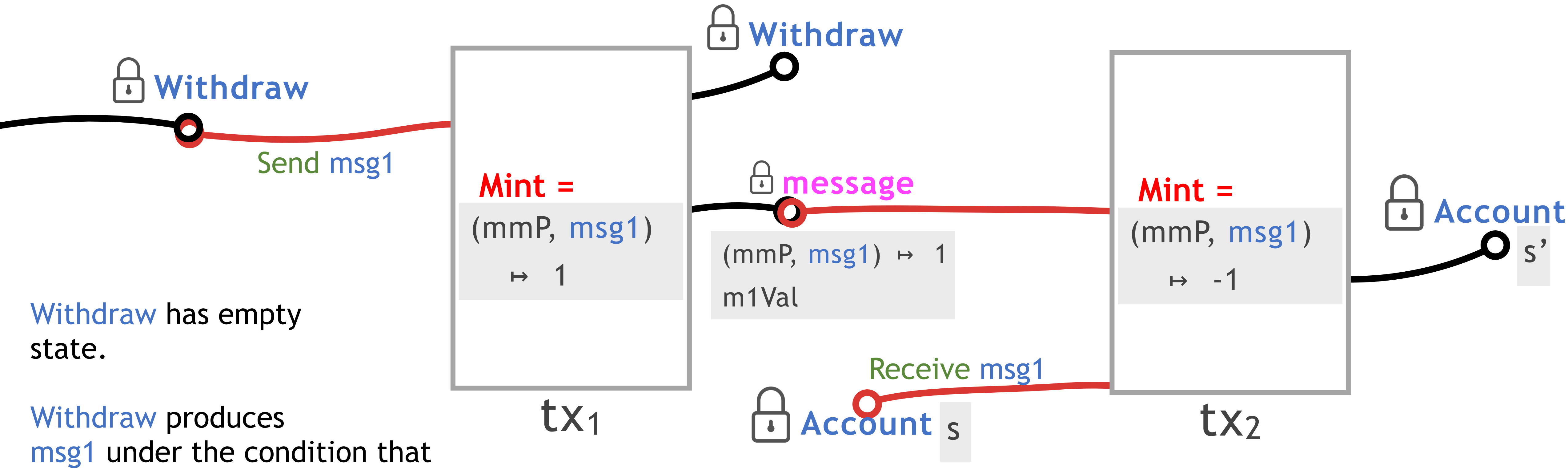
VerifyFactorial has empty state.

It produces msg1 under condition that

```
snd (mData msg1) ==
factorial (fst (mData msg1))
```

What can messages do?

Given a contract `Account` with input type `Withdraw WArgs | … | Transfer TArgs`



Withdraw has empty state.

Withdraw produces msg1 under the condition that

```
acctTrans s (Withdraw args) = s'
```

msg1 = (Withdraw, Account, 0 , (s, s'))

What can messages do?

We can use these proof artefact messages to distribute computation across multiple contracts

For a state machine `Account`, the contract `Withdraw` **precomputes** the state to which Account must transition to perform the `Withdraw` function

- This precomputation is **recorded as an artefact on-chain**, with **provenance** guaranteed by the token

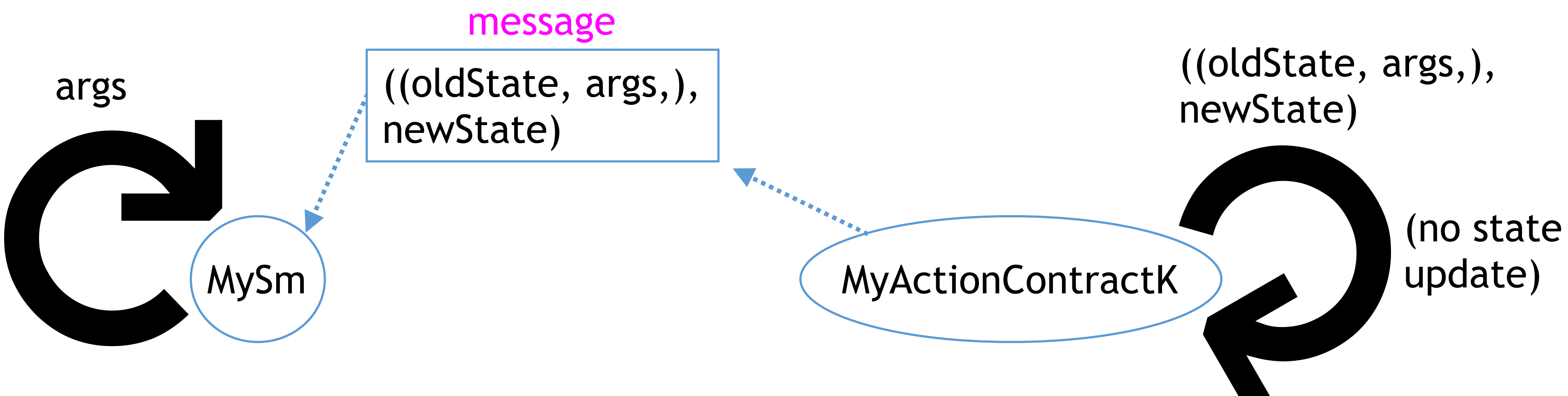
We can generalize this to contracts where the state computation for each transition input constructor is less trivial

What can messages do?

We can use these proof artefact messages to distribute computation across multiple contracts

- For each of `MyActionK`, `MySM` only emits a constraint that a message must be consumed from contract, with `mData = ((MyState, ArgsK), ???)`
  - Currently, constraints don't let us under-specify like this a currency being minted

If, eg. only of the MyActionK computations is a **large, but rarely used piece of code**, this reduce memory use



Can we specify messages in a more general way?

Messages as stateful contracts

Recall the ledger spec…

This is the **small-step-semantics specification** of how to **apply a transaction** to the ledger state

Ledger transition type :

$$\_ \vdash \_ \xrightarrow[\text{LEDGER}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{LEnv} \times \mathsf{LState} \times \mathsf{Tx} \times \mathsf{LState})$$

`LEDGER` transition (`isValid = True` rule) :

$$\text{ledger-V}\;\frac{\begin{array}{c}\mathsf{isValid}\ tx = \mathsf{True} \\[1em] \begin{array}{c} slot \\ txIx \\ pp \\ tx \\ acnt \end{array} \vdash dpstate \xrightarrow[\text{DELEGS}]{\mathsf{txcerts}\ (\mathsf{txbody}\ tx)} dpstate' \\[1em] (dstate, pstate) := dpstate \\ (\_, \_, \_, \_, \_, genDelegs, \_) := dstate \\ (poolParams, \_, \_) := pstate \\[1em] \begin{array}{c} slot \\ pp \\ poolParams \\ genDelegs \end{array} \vdash utxoSt \xrightarrow[\text{UTXOW}]{tx} utxoSt' \end{array}}{\begin{array}{c} slot \\ txIx \\ pp \\ acnt \end{array} \vdash \begin{pmatrix} utxoSt \\ dpstate \end{pmatrix} \xrightarrow[\text{LEDGER}]{tx} \begin{pmatrix} utxoSt' \\ dpstate' \end{pmatrix}}$$

Instance of a structured contract

An EUTxO structured contract is given by specifying the following data :

(i) Surjective projection $\pi_{\mathsf{State}} \in \mathsf{UTxOState} \rightarrow \mathsf{State}$

(ii) Surjective projection $\pi_{\mathsf{Input}} \in \mathsf{TxInfo} \rightarrow \mathsf{Input}^{?}$

(iii) Some set of inference rules that specify the transition of type `SMUP` $\_ \vdash \_ \xrightarrow[\mathrm{SMUP}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{TxInfo} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State})$

(iv) a proof that the **`StatefulStep`** and **`StatefulNoStep`** property is satisfied by the data in (i), (ii), and (iii)

Structured Contracts Simulation Relation

SMUP is simulated by the ledger when **StatefulNoStep** and **StatefulNoStep** are satisfied

$$\text{StatefulStep}\ \frac{\begin{array}{c} \mathit{txInfo} := \mathsf{txInfo}\ \mathsf{El}\ \mathsf{SysSt}\ \mathsf{Lang}\ \mathit{pp}\ (\mathsf{getUTxO}\ \mathit{utxoSt})\ \mathit{tx} \\ \mathsf{isValid}\ \mathit{tx} = \mathsf{True} \qquad \pi_{\mathit{Input}}\ \mathit{txInfo} \neq \diamond \\ \\ \begin{array}{c} \mathit{slot} \\ \mathit{txIx} \\ \mathit{pp} \\ \mathit{account} \end{array} \vdash \begin{pmatrix} \mathit{utxoSt} \\ \mathit{dpstate} \end{pmatrix} \xrightarrow[\mathrm{LEDGER}]{\mathit{tx}} \begin{pmatrix} \mathit{utxoSt}' \\ \mathit{dpstate}' \end{pmatrix} \end{array}}{\mathit{txInfo} \vdash (\pi_{\mathit{State}}\ \mathit{utxoSt}) \xrightarrow[\mathrm{SMUP}]{\pi_{\mathit{Input}}\ \mathit{txInfo}} (\pi_{\mathit{State}}\ \mathit{utxoSt}')}$$

Structured Contracts Simulation Relation

`SMUP` is simulated by the ledger when **`StatefulNoStep`** and **`StatefulNoStep`** are satisfied

This is not a rule, it is a **constraint** that may or may not be satisfied in general

- but has to be, by definition, if SMUP is a structured contract

$$\text{StatefulStep}\ \dfrac{\begin{array}{c} \mathit{txInfo} := \mathsf{txInfo\ El\ SysSt\ Lang}\ \mathit{pp}\ (\mathsf{getUTxO}\ \mathit{utxoSt})\ \mathit{tx} \\ \mathsf{isValid}\ \mathit{tx} = \mathsf{True} \qquad\qquad \pi_{Input}\ \mathit{txInfo} \neq \diamond \\ \begin{array}{c} \mathit{slot} \\ \mathit{txIx} \\ \mathit{pp} \\ \mathit{account} \end{array} \vdash \begin{pmatrix} \mathit{utxoSt} \\ \mathit{dpstate} \end{pmatrix} \xrightarrow[\text{LEDGER}]{\mathit{tx}} \begin{pmatrix} \mathit{utxoSt}' \\ \mathit{dpstate}' \end{pmatrix} \end{array}}{\mathit{txInfo} \vdash (\pi_{State}\ \mathit{utxoSt}) \xrightarrow[\text{SMUP}]{\pi_{Input}\ \mathit{txInfo}} (\boldsymbol{\pi_{State}\ \mathit{utxoSt}'})}$$

Message-passing implementation

Message-passing transition type : $\_ \vdash \_ \xrightarrow[\text{MSGS}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{TxInfo} \times \mathsf{Msgs} \times \mathsf{MsgIn} \times \mathsf{Msgs})$

Message-passing transition specification :

$$\begin{aligned} (produce, consume) &:= mint \\ mint &= \pi_{MsgIn}\; txInfo \end{aligned}$$

$$cstate' \;:=\; cstate \underset{\rightarrow}{\cup} \{(\mathsf{mFrom}\; msg) \mapsto (s', \;\mathsf{updateM}\; cstate\; msg\; msgs) \mid msg \;\in\; msgs\,\}$$

$$cstate'' \;:=\; cstate' \underset{\rightarrow}{\cup} \{(\mathsf{mTo}\; msg) \mapsto (s', \;\mathsf{updateB}\; cstate'\; msg\; msgs) \mid msg \;\in\; msgs\,\}$$

$$\text{Mint}\; \frac{\begin{array}{c} txInfo \vdash (\,cstate\,) \xrightarrow[\text{SMACC}]{\mathsf{Send}\; produce} (\,cstate'\,) \\ txInfo \vdash (\,cstate\,) \xrightarrow[\text{SMACC}]{\mathsf{Receive}\; consume} (\,cstate''\,) \end{array}}{txInfo \vdash (\,msgs\,) \xrightarrow[\text{MSGS}]{mint} (\,msgs \cup produce - consume\,)}$$

Implementation of message-passing

The implementation projection functions formally relates the **message-passing specification** MSGS transition type and rules to the **Plutus boolean verifier code that runs on-chain**

This gives us a concrete formulation of what it means to *implement a structured contract*

$$
\begin{aligned}
\pi_{\mathsf{Msgs}}\ \mathit{utxoSt} \quad & = \{\, \mathit{tn} \mid \_ \mapsto \mathit{out} \in (\mathsf{getUTxO}\ \mathit{utxo}),\ \mathsf{msgAddress} = \mathsf{addressCredential}\ (\mathsf{txOutAddress}\ \mathit{out}), \\
& \quad \boxed{(\mathsf{msgPolHash},}\ \mathit{tn}) \mapsto (1) \in \mathsf{txOutValue}\ \mathit{out} \,\} \\
\pi_{\mathsf{MsgIn}}\ \mathit{txinfo} \quad & = (\{\, \mathit{rdm} \mid \mathsf{msgPolHash} \mapsto \mathit{lsRdm} \in \mathsf{txInfoRedeemers}\ \mathit{txInfo}, \\
& \quad \mathit{rdm} \in \mathit{lsRdm},\ (\boxed{\mathsf{msgPolHash},}\ \mathit{rdm}) = \mathit{aid}, \\
& \quad \mathit{aid} \mapsto q \in \mathsf{txInfoMint}\ \mathit{txinfo},\ q \geq 1 \,\}, \\
& \quad \{\, \mathit{rdm} \mid \mathsf{msgPolHash} \mapsto \mathit{lsRdm} \in \mathsf{txInfoRedeemers}\ \mathit{txInfo}, \\
& \quad \mathit{rdm} \in \mathit{lsRdm},\ (\mathsf{msgPolHash},\ \mathit{rdm}) = \mathit{aid}, \\
& \quad \mathit{aid} \mapsto q \in \mathsf{txInfoMint}\ \mathit{txinfo},\ q \leq (-1) \,\})
\end{aligned}
$$

Implementation of message-passing

- We still need to show that the MSGS transition satisfies the **`StatefulStep`** and **`StatefulNoStep` constraints**

- To say that a contract **participates in message-passing**, or **is a message-passing contract** with input type `SimplInput` and transition type **`SMACC`**, we require that
  - There is a projection `pi_MsgIn : SimplInput -> MsgIn`
  - `MSGS` transition relation does not violate the `Mint` rule

- **Multiple implementations** are possible
  - Eg. All messages are stored in a CE state machine (like we do with naive accounts)

Accounts and Message-passing

Accounts with message-passing have an input of type :

```
MsgIn | OArgs |
CArgs | DArgs |
WArgs
```

$$\text{Transfer-From}\;\frac{\begin{array}{c} accIn \in \mathsf{MsgIn} \\[1em] \mathsf{idFrom}\ accIn \mapsto oldAcctFrom \in accts \qquad pkFrom := \mathsf{pk}\,(idFrom,\ oldAcctFrom) \\[1em] \mathsf{val}\ oldAcctFrom \geq \mathsf{val}\ accIn \geq \mathsf{zero} \\ \mathsf{val}\ changedAcctFrom = \mathsf{val}\ oldAcctFrom - \mathsf{val}\ accIn \\[1em] \mathsf{pk}\,(\mathsf{idFrom}\ accIn,\ changedAcctFrom) = pkFrom \\ pkFrom \in \mathsf{txInfoSignatories}\ txInfo \\[1em] txInfo \vdash msgs \xrightarrow[\text{MSGS}]{accIn} msgs' \end{array}}{txInfo \vdash (accts) \xrightarrow[\text{ACCNT}]{accIn} \left(accts \underset{\rightarrow}{\cup} \{idFrom \mapsto changedAcctFrom\} \right)}$$

Wallets and Message-passing

If a wallet is identified by a PK, each wallet's account is the sum of assets locked by that PK

$$\text{UpdateWallets}\ \frac{\begin{array}{c} \mathit{ins} := [\ \mathsf{txInInfoResolved}\ \mathit{ri}\ \mid\ \mathit{ri} \leftarrow \mathsf{txInfoInputs}\ \mathit{txInfo}\ ] \\ \mathit{outs} := (\mathsf{txInfoOutputs}\ \mathit{txInfo}) \\ \\ \forall\ (\mathsf{msgPolHash} \mapsto (\mathsf{getPreim}\ \mathit{txInfo}\ \mathit{msg}) \mapsto 1) \in \mathsf{txInfoMint}\ \mathit{txInfo},\ \mathsf{mFrom}\ \mathit{msg} \in \mathsf{PubKey}, \\ \mathsf{mFrom}\ \mathit{msg} \in \mathsf{txInfoSignatories}\ \mathit{txInfo} \\ \\ \forall\ (\mathsf{msgPolHash} \mapsto (\mathsf{getPreim}\ \mathit{txInfo}\ \mathit{msg}) \mapsto -1) \in \mathsf{txInfoMint}\ \mathit{txInfo},\ \mathsf{mTo}\ \mathit{msg} \in \mathsf{PubKey}, \\ \mathsf{mTo}\ \mathit{msg} \in \mathsf{txInfoSignatories}\ \mathit{txInfo} \\ \\ \mathit{accts}' := \mathit{accts} \cup_{+} \{\ \mathit{pk} \mapsto \Sigma_{((\mathit{pk},\_),v,\_) \in \mathit{outs}} v\ \mid\ ((\mathit{pk},\_),\_,\_) \in \mathit{outs}\ \} \\ \cup_{-} \{\ \mathit{pk} \mapsto \Sigma_{((\mathit{pk},\_),v,\_) \in \mathit{ins}} v\ \mid\ ((\mathit{pk},\_),\_,\_) \in \mathit{ins}\ \} \end{array}}{\mathit{txInfo} \vdash (\mathit{accts}) \xrightarrow[\text{ACCNT}]{\mathit{accIn}} (\mathit{accts}')}$$