What is Trust

Orfeas Stefanos Thyfronitis Litos

University of Edinburgh
o.thyfronitis@ed.ac.uk

Abstract. We will try to define all the abstract properties that we would like "Trust" to have.

1 High-level idea

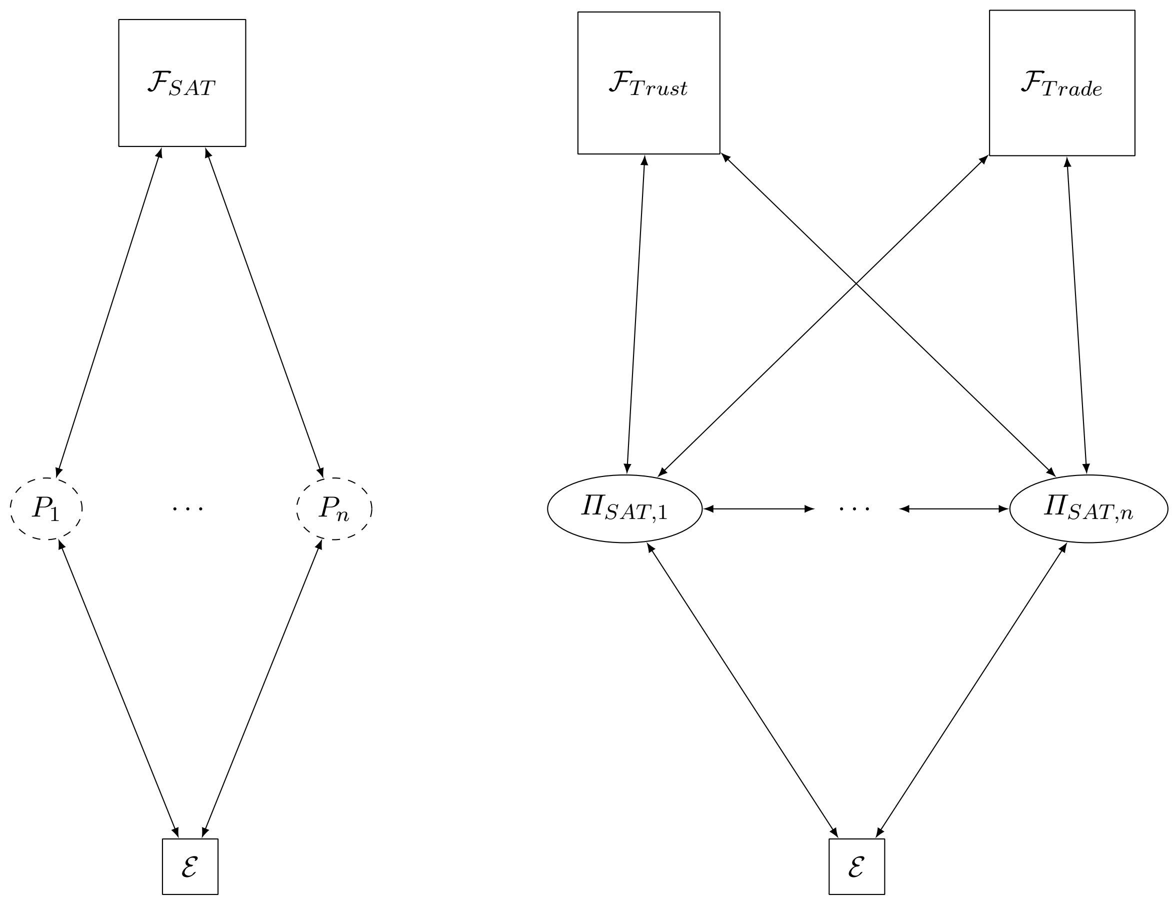

Fig. 1: (Almost) all functionalities

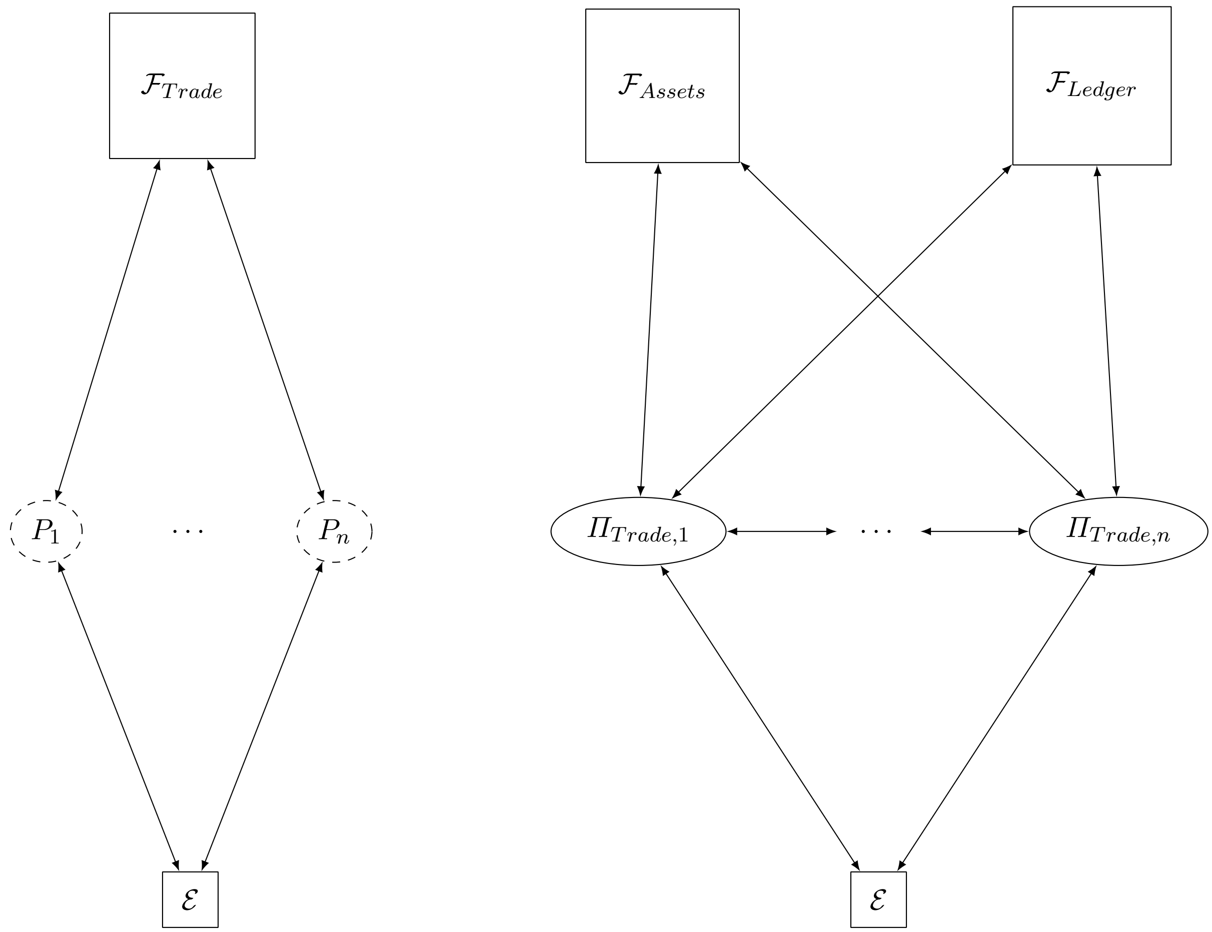

Fig. 2: Trade functionality and protocol

2 Utility Function Properties

Before *Alice* (an ITM that executes Π_{SAT}) can take any action, she must be assigned a utility function, U_{Alice}, and a discount function, λ_{Alice}, by $\mathcal{E}$.

Definition 1 (Utility Function). $U_{Alice} : Money \times multiset\,(Asset) \times Rep \rightarrow Utility$, *where* $Money, Rep \in \mathbb{N}, Utility \in \mathbb{R}$.

The utility function is strictly increasing with respect to the integer $Money$ and may do anything with respect to the quantity of any single

Asset or combination of *Asset*s. It is also strictly increasing with respect to the integer *Rep*.

Definition 2 (Reputation function). *Reputation* : $\mathcal{P} \times Time \times 2^{Asset} \to$ *Rep, where* $Time, Rep \in \mathbb{N}$

$$\begin{gathered} Rep\,(Alice, t, set) > Rep\,(Bob, t, set) \overset{def}{\Longleftrightarrow} \\ \Pr\left[Alice \text{ receives message } \left(\text{canYouSatisfy}, d\right) \text{ at time t}\right] > \\ \Pr\left[Bob \text{ receives message } \left(\text{canYouSatisfy}, d\right) \text{ at time t}\right] \\ \text{where } d \in set \end{gathered}$$

Rep is an integer that corresponds to *Alice*'s perceived rank in a partial ordering of all the players where *Alice* assumes that she has a higher rank than *Bob* if the average probability of third players choosing her before *Bob* when they want to buy something amongst a list of *Asset*s that *Alice* is interested in selling is higher than *Bob*'s corresponding probability for the same *Asset*s, as perceived by *Alice*.

Definition 3 (Discount function). $\lambda_{Alice} : Time \to [0, 1]$*, where* $Time \in \mathbb{N}$ *and* λ *is strictly decreasing.*

Definition 4 (State). *Let* $state_{Alice} : Time \to Money \times multiset\,(Asset) \times Rep$*.*

Alice's target is to maximise

$$\sum_{t=1}^{\infty} \lambda_{Alice}\,(t) \left(\sum_{\substack{(m,a,r) \in \\ Money \times mset(Asset) \times Rep}} U_{Alice}\,(m, a, r) \cdot \Pr\left[state_{Alice}\,(t) = (m, a, r)\right] \right) .$$

3 Protocol

In this section we will describe the real protocol executed by the players in the absence of the $\mathcal{F}_{SAT}$ ideal functionality. The description of this protocol does not need any utility function; all the "important" decisions are taken by the environment.

Consider an environment $\mathcal{E}$, and adversary $\mathcal{A}$ and n players executing copies of the same protocol $\Pi_1, \ldots, \Pi_n$. $\mathcal{E}$ can send the following messages to Π_i:

1. Manage player desires
   (a) Satisfy $d \in \mathcal{D}$ through a player in $L \subseteq [n]$
   (b) Abort attempt to satisfy $d \in \mathcal{D}$
2. Manage offered desires satisfaction
   (a) Gain the ability to satisfy $d \in \mathcal{D}$ for players in $L \subseteq [n]$ for a price $x \in \mathbb{N}$
   (b) Lose the ability to satisfy $d \in \mathcal{D}$ for players in $L \subseteq [n]$ for a price $x \in \mathbb{N}$
3. Satisfy another player's desire
   (a) Satisfy player's $i \in [n]$ desire $d \in \mathcal{D}$ with the corresponding satisfaction string s
   (b) Satisfy player's $i \in [n]$ desire $d \in \mathcal{D}$ with the satisfaction string s' (normally suitable for satisfying $d' \neq d, d' \in \mathcal{D}$)
   (c) Ignore player's $i \in [n]$ desire $d \in \mathcal{D}$
4. Manage direct trusts
   (a) Increase direct trust to player $i \in [n]$ by $x \in \mathbb{N}$
   (b) Decrease direct trust to player $i \in [n]$ by $x \in \mathbb{N}$
   (c) Steal direct trust $x \in \mathbb{N}$ from player $i \in [n]$

Some of these messages (e.g. 1b) are meaningful only when some other messages have been delivered previously (e.g. 1a). $\mathcal{E}$ may send such messages even when they are not meaningful; the protocol should take care to reject/ignore such messages.

Let $i \in [n]$. Π_i can send the following messages to $\mathcal{E}$:

1. No player in L can satisfy my desire $d \in \mathcal{D}$
2. Desire $d \in \mathcal{D}$ made available for satisfaction amongst $L \subseteq [n]$ for price $x \in \mathbb{N}$
3. Desire $d \in \mathcal{D}$ made unavailable for satisfaction amongst $L \subseteq [n]$ for price $x \in \mathbb{N}$
4. Payment $x \in \mathbb{N}$ has been sent to player $j \in [n]$ for the satisfaction of desire $d \in \mathcal{D}$
5. Correct Payment $x \in \mathbb{N}$ from player $j \in [n]$ for the satisfaction of desire $d \in \mathcal{D}$ has been received
6. Wrong Payment $x \in \mathbb{N}$ from player $j \in [n]$ for the satisfaction of desire $d \in \mathcal{D}$ has been received
7. Player $j \in [n]$ has satisfied my desire $d \in \mathcal{D}$ with satisfaction string s
8. Player $j \in [n]$ has partially satisfied my desire $d \in \mathcal{D}$ with satisfaction string s' (normally suitable for satisfying $d' \neq d, d' \in \mathcal{D}$)
9. Player $j \in [n]$ has ignored my desire $d \in \mathcal{D}$
10. Direct trust to player $i \in [n]$ increased by $x \in \mathbb{N}$

11. Direct trust to player $i \in [n]$ decreased by $x \in \mathbb{N}$
12. Stole $x \in \mathbb{N}$ from player's $i \in [n]$ direct trust

Π_i should only send these messages when $\mathcal{E}$ is expecting them.

Let $i, j \in [n]$ The messages that can be sent between Π_i and Π_j are the following:

1. Can you satisfy $d \in \mathcal{D}$?
2. I can satisfy $d \in \mathcal{D}$ for a price $x \in \mathbb{N}$
3. I cannot satisfy $d \in \mathcal{D}$
4. Payment of $x \in \mathbb{N}$ for satisfaction of $d \in \mathcal{D}$ sent
5. Satisfaction string s', response to payment of $x \in \mathbb{N}$ for $d \in \mathcal{D}$

Π_i is supposed to send 4 when it has already paid through $\mathcal{F}_{Ledger}$. Similarly, it is supposed to send 5 when it has verified that the other party has sent the corresponding payment on $\mathcal{F}_{Ledger}$.

Going in more detail, the actual protocol is as follows:

Π_{SAT}

```
Initalization:
 util = ⊥

Upon receiving (type, t) from ℰ:
 util = t

Upon receiving message (satisfy, d, L) from ℰ:
 If util == ⊥:
   send message (utilityNotSet) to ℰ
   go to Idle state
 aux = L
 While (aux ≠ ∅):
   send message (chooseBestSeller, d, aux) to ℱ_Trust
   wait for response1 from ℱ_Trust
   If response1 == (bestSeller, d, L, Bob):
     send message (canYouSatisfy, d) to Bob
     wait for response2 from Bob
     If response2 == (IcanSatisfy, d, x, s):
       If util(state ∪ s − x) > util(state) and s ∈ d
         send message (trade, x, s, Bob) to ℱ_Trade
         wait for response3 from ℱ_Trade
           If response3 == (traded, x, s, Bob):
             send message (satisfied, d, x, s, Bob) to ℱ_Trust
```

```
          # maybe send only utility difference to $\mathcal{F}_{Trust}$
          send message (satisfied, $d, L$) to $\mathcal{E}$
          go to Idle state
        Else If response3 == (cheated, $x, s$, $Bob$):
          send message (cheated, $d, x, s, Bob$) to $\mathcal{F}_{Trust}$
          send message (cheated, $d, L$) to $\mathcal{E}$
          go to Idle state
    Else: # if response1 == $\perp$
      send message (unsatisfied, $d, L$) to $\mathcal{E}$
      go to Idle state
    aux = aux $\setminus \{Bob\}$ # only when response2 is not good
  send message (unsatisfied, $d, L$) to $\mathcal{E}$

Upon receiving message (obtain, $s$) from $\mathcal{E}$:
  send message (obtain, $s$) to $\mathcal{F}_{Trade}$
  wait for response from $\mathcal{F}_{Trade}$
  If response == (obtained, $s$):
    send message (obtained, $s$) to $\mathcal{E}$
  Else:
    send message (notObtained, $s$) to $\mathcal{E}$

Upon receiving message (lose, $s$) from $\mathcal{E}$:
  send message (lose, $s$) to $\mathcal{F}_{Trade}$
  wait for response from $\mathcal{F}_{Trade}$
  If response == (lost, $s$):
    send message (lost, $s$) to $\mathcal{E}$
  Else:
    send message (notLost, $s$) to $\mathcal{E}$

Upon receiving message (canYouSatisfy, $d$) from $Alice$:
  If util == $\perp$:
    ignore request, go to Idle state
  If (util($state + x + badRep$) > util($state$) or
      util($state \setminus s + x + goodRep$) > util($state$)) and $s \in d$:
    send message (IcanSatisfy, $d, x, s$) to $Alice$ # $x$ is the
        price
  Else:
    send message (IcannotSatisfy, $d$) to $Alice$

Upon receiving message (willWeCheat, $x, s, Alice$) from $\mathcal{F}_{Trade}$:
```

```
If util == ⊥: # (Unreachable since we've already engaged)
  ignore request, go to Idle state
If util(state \ s + x + goodRep) ≥ util(state + x + badRep):
  send message (doNotCheat, x, s, Alice) to F_Trade
Else:
  send message (cheat, x, s, Alice) to F_Trade
```

4 Desire Satisfaction Ideal Functionality

Following the UC paradigm, in this section we define the ideal functionality for desire satisfaction, $\mathcal{F}_{SAT}$. In this setting, all the desires that are generated by the environment and are input to the players are immediately forwarded to $\mathcal{F}_{SAT}$; the functionality decides which desires to satisfy. Since the players are dummy and all desires are satisfied by the functionality, no trust semantics amongst the players are necessary.

Nevertheless, given that all desires have a minimum cost, the cost semantics are still necessary, as we show with the following example: Consider a set of desires D with more elements than the total number of tokens all players have. D could never be satisfied by the players because of the high total cost, but a $\mathcal{F}_{SAT}$ with no consideration for cost could in principle satisfy all desires in D.

The functionality can calculate the properties and functions defined in **??**, **??** and **??** for all inputs at any moment in time.

Without knowledge of the utilities the environment is going to give to each satisfied desire, the functionality may fail spectacularly. So knowledge of the utility of each desire, or at least some function of the utility given the desires is needed. We can assume that $\mathcal{F}_{SAT}$ knows U or an approximation of it.

Going into more detail, $\mathcal{F}_{SAT}$ is a stateful process that acts as a market and as a bank for the players. The market does not offer a particular product for the same price to all users; For some users it may be cheaper than for others, reflecting the fact that some players can realize some desires more efficiently than others.

$\mathcal{F}_{SAT}$ stores a number for each player that represents the amount of tokens this player has and a table with the price of each desire for each player. It provides the functions $cost\,(u, d)$ which returns the cost of the desire d for player u with no side effects, $sat\,(u, d)$ that returns the string that satisfies the desire d to u and reduces the amount of the tokens belonging to u by $cost\,(u, d)$. There exists also the function $transfer\,(u_1, u_2, t)$ which reduces the amount of tokens u_1 has by t and increases the tokens of u_2 by t, given that initially the tokens belonging to u_1 were equal or more than t. This function is private to the functionality, thus can be used only internally.

$\mathcal{F}_{SAT}$

```
Initialisation:
  $\forall Alice \in \mathcal{P}$,
    util($Alice$) = $\perp$
    assets($Alice$) = $\perp$

Upon receiving (type, $t$) from $Alice$:
  util($Alice$) = $t$

Upon receiving (satisfy, $d, L$) from $Alice$:
  If util($Alice$) == $\perp$:
    send message (utilityNotSet) to $Alice$
    go to Idle state
  Find list = $\{(Bob, x, s) \in L \times \mathbb{R} \times Assets$ :
    $s \in$ assets($Bob$) and $s \in d$ and $x \geq 0$ and
    $Alice$ has at least $x$ coins available and
    util($Alice$)($state_{Alice} \cup s - x$) > util($Alice$)($state_{Alice}$)
    and
    util($Bob$)($state_{Bob} \setminus s + x + goodRep$) > util($Bob$)($state_{Bob}$)
    and
    util($Bob$)($state_{Bob} \setminus s + x + goodRep$) >
        util($Bob$)($state_{Bob} + x + badRep$) $\}$
  send (chooseBestSeller, list, $d$, $Alice$) to $\mathcal{A}$
  wait for response from $\mathcal{A}$
  With response as (bestSeller, list, $Bob, x, s$), $Bob \in \mathcal{P}$:
    If $x > 0$:
      Pay $x$ from $Alice$ to $Bob$
    assets($Bob$) = assets($Bob$)$\setminus \{s\}$
    assets($Alice$) = assets($Alice$)$\cup \{s\}$
    send message (satisfied, $d, L$) to $Alice$
  Else If response $\neq$ (bestSeller, list, $Bob, x, s$): # e.g.
      $Bob = \perp$
    send message (unsatisfied, $d, L$) to $Alice$

Upon receiving (obtain, $s$) from $Alice$:
  assets($Alice$) = assets($Alice$)$\cup \{s\}$
  send message (obtained, $s$) to $Alice$
```

```
Upon receiving (lose, s) from Alice:
  assets(Alice) = assets(Alice) \ {s}
  send message (lost, s) to Alice
```

$\mathcal{F}_{Trade}$

```
Initialisation:
  ∀Alice ∈ P,
    assets(Alice) = ⊥

Upon receiving (trade, ours, theirs, Bob) from Alice:
  If not isAvailable(ours, Alice):
    send message (youDontHave, ours) to Alice
    go to Idle state
  If transfer(ours, Alice, Bob) == True:
    send message (willWeCheat, ours, theirs, Alice) to Bob
    wait for response from Bob
    If (response == (complete, x, s, Alice) and
      not isAvailable(s, Bob)) or
      response ≠ (complete, x, s, Alice):
        send message (youDontHave, s) to Bob
        send message (cheated, ours, theirs, Bob) to Alice
        go to Idle state
    Else If (transfer, theirs, Bob, Alice) == True:
      send message (traded, ours, theirs, Bob) to Alice
  Else # failed to give (Unreachable for a good F_Ledger)
    send message (failed, ours, theirs, Bob) to Alice

isAvailable(object, player):
  If object is money:
    send (doIhaveBalance, object) to F_Ledger as player
    wait for response from F_Ledger
    return response
  Else: # object is asset
    If object ∈ assets(player):
      return True
    Else:
      return False

transfer(object, sender, receiver):
```

```
  If isAvailable(object):
    If object is money:
      send (pay, object, receiver) to F_Ledger as sender
      wait for response from F_Ledger
      Upon receiving (paymentDone, object, receiver):
        return True
    Else: # object is asset
      assets(sender) = assets(sender) \ {object}
      assets(receiver) = assets(receiver) ∪ {object}
      return True
  return False

Upon receiving (obtain, s) from Alice:
  assets(Alice) = assets(Alice) ∪ {s}
  send message (obtained, s) to Alice

Upon receiving (lose, s) from Alice:
  assets(Alice) = assets(Alice) \ {s}
  send message (lost, s) to Alice
```

$\mathcal{F}_{Assets}$

```
Initialisation:
  ∀Alice ∈ P,
    assets(Alice) = ⊥

Upon receiving (add, asset) from Alice:
  assets(Alice) = assets(Alice) ∪ {asset}
  send (added, asset) to Alice

Upon receiving (remove, asset) from Alice:
  If asset ∈ assets(Alice):
    assets(Alice) = assets(Alice) \ {asset}
    send (removed, asset) to Alice
  Else:
    send (unableToRemove, asset) to Alice

Upon receiving (howManyDoIhave, asset) from Alice:
  send (youHave, assets(Alice).count(asset)) to Alice

Upon receiving (transfer, asset, Bob) from Alice:
```

```
  If asset ∈ assets(Alice):
    assets(Alice) = assets(Alice)\{asset}
    assets(Alice) = assets(Bob)∪{asset}
    send (transferred, asset, Bob) to Alice
  Else:
    send (unableToTransfer, asset, Bob) to Alice
```

Π_{Trade}

```
Upon receiving (trade, ours, theirs, Bob) from ℰ:
  Send (letsTrade, ours, theirs) to Bob
  If transfer(ours, Bob) == True:
    Send (transferred, ours, Bob) to Bob and ℰ
    Wait for response from Bob
    If response == (transferred, theirs, Bob):
      send message (traded, ours, theirs, Bob) to ℰ
    Else:
      send message (cheated, ours, theirs, Bob) to ℰ

Upon receiving (letsTrade, theirs, ours) from Bob:
  Send (willWeCheat, theirs, ours, Bob) to ℰ
  Wait for response from ℰ
  If response is (doNotCheat, theirs, ours, Bob):
    If (transfer, ours, Bob) == True:
      Send (transferred, ours, Bob) to Bob and ℰ

transfer(object, receiver):
  If isAvailable(object):
    If object is money:
      send (pay, object, receiver) to F_Ledger
      wait for response from F_Ledger
      Upon receiving (paymentDone, object, receiver):
        return True
    Else: # object is asset
      send (transfer, object, receiver) to F_Assets
      wait for response from F_Assets
      Upon receiving (transferDone, object, receiver):
        return True
  return False

isAvailable(object):
```

```
  If object is money:
    send (doIhaveBalance, object) to $\mathcal{F}_{Ledger}$
    wait for response from $\mathcal{F}_{Ledger}$
    return response
  Else: # object is asset
    Send (doIHave, object) to $\mathcal{F}_{Assets}$
    wait for response from $\mathcal{F}_{Assets}$
    If response == (youHave, object):
      return True
    Else:
      return False

Upon receiving message (obtain, $s$) from $\mathcal{E}$:
  send message (add, $s$) to $\mathcal{F}_{Assets}$
  wait for response from $\mathcal{F}_{Assets}$
  If response == (added, $s$)
    send message (obtained, $s$) to $\mathcal{E}$
  Else
    send message (notObtained, $s$) to $\mathcal{E}$

Upon receiving message (lose, $s$) from $\mathcal{E}$:
  send message (remove, $s$) to $\mathcal{F}_{Assets}$
  wait for response from $\mathcal{F}_{Assets}$
  If response == (removed, $s$)
    send message (lost, $s$) to $\mathcal{E}$
  Else
    send message (notLost, $s$) to $\mathcal{E}$
```

5 $\mathcal{F}_{SAT}$ and Π_{SAT} are potentially distinguishable

Consider the hybrid world of Fig. 1 (right) with n ITMs executing Π_{SAT}, where $\mathcal{F}_{Trust}$ is replaced by $\mathcal{F}'_{Trust}$:

$\mathcal{F}'_{Trust}$

```
Upon receiving (chooseBestSeller, $d$, $L$) from $Alice$:
  $Bob \xleftarrow{R} L \cup \{\perp\}$
  send message (bestSeller, $d$, $L$, $Bob$) to $Alice$
```

We will show here that $\mathcal{E}$ can distinguish between $\mathcal{F}_{SAT}$ and a Π_{SAT} that uses $\mathcal{F}'_{Trust}$.

Distinguishability. Consider the following adversary and environment:

$\mathcal{A}$

Upon receiving (chooseBestSeller, list, d, $Alice$):
  If $|\texttt{list}| \neq 1 \vee |d| \neq 1$:
    go to Idle State
  Bob = $p : p \in$ list
  s = $asset : asset \in d$
  return (bestSeller, list, Bob, 1, s)

$\mathcal{E}$ distinguisher

$Alice \xleftarrow{R} \mathcal{P}$
$Bob \xleftarrow{R} \mathcal{P} \setminus \{Alice\}$
util($Alice$)($\{assets\}, x, r$) = $2\,|\{assets\}| + x$
util(Bob)($\{assets\}, x, r$) = $|\{assets\}| + 2x$
$\lambda\,(Alice)\,(t)$ = $\lambda\,(Bob)\,(t)$ = $\frac{1}{t^2}$
$\forall p \in \{Alice, Bob\}$:
  send message (type, util(p), $\lambda\,(p)$) to p
$s \xleftarrow{R} Asset$
send message (obtain, s) to Bob
send message (satisfy, $\{s\}$, $\{Bob\}$) to $Alice$

Upon receiving message (x, $\{s\}$, $\{Bob\}$) from $Alice$:
  If x == satisfied:
    return functionality
  Else: # if x $\in$ {cheated, unsatisfied}
    return protocol

Because of the way $\mathcal{E}$ is built, there always exists a seller (Bob) that can satisfy the desire of the buyer ($Alice$).

In case $\mathcal{E}$ interacts with $\mathcal{F}_{SAT}$, then $\mathcal{A}$ will always match the buyer and the seller because of the way the former is built. Additionally, the price of 1 is cheap enough for $Alice$ to want to buy the asset, since she values acquiring one asset more than one coin.

Furthermore, $\mathcal{F}_{SAT}$ never cheats on a trade, thus the exchange will always complete correctly and $\mathcal{E}$ will receive `satisfied` as response. $\mathcal{E}$ will always correctly output `functionality`.

On the other hand, in case $\mathcal{E}$ interacts with Π_{SAT}, then we observe that $\mathcal{F}'_{Trust}$ does not choose players depending on their reputation. Thus, if $\mathcal{F}'_{Trust}$ does not respond with $\perp$, it is always in Bob's interest to cheat (since keeping the asset is preferrable to giving it) and $\mathcal{E}$ will always receive `cheated` as response. If $\mathcal{F}'_{Trust}$ responds with $\perp$, the trade will not go through and $\mathcal{E}$ will receive `unsatisfied` as a response. In all cases $\mathcal{E}$ will correctly output `protocol`. $\square$

References