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 Introduction

Consider the UC setting, with an environment $\mathcal{E}$, an adversary $\mathcal{A}$ and a set of ITIs that follow a given protocol Π.

Definition 1 (Player). *A player is an ITM that follows Π. Let $\mathcal{P}$ be the set of all players.*

Intuitively, players spontaneously feel different desires of varying intensities and seek to satisfy them, either on their own, consuming part of their input tokens in the process, or by delegating the process to other players and paying them for their help with part of their input tokens. The choice depends on the perceived difference in price and whether other players are trustworthy enough to satisfy the desire as promised. Each player plays rationally, always attempting to maximize her utility.

2 Mechanics

More precisely, let $\mathcal{D}$ be a set containing all possible desires. At arbitrary moments during execution, $\mathcal{E}$ can provide input to any player $Alice \in \mathcal{P}$ in the form (idx, d), where $idx \in \mathbb{N}, d \in \mathcal{D}, u \in \mathbb{R}^{+}$. idx represents an index number that is unique for each input generated by $\mathcal{E}$ and d represents the desire. d is satisfied when $Alice$ learns the string $s\left(idx, d, Alice\right)$, either by directly calculating it or by receiving it as a message from another player. Some of the players, given as input the tuple $(idx, d, Alice)$, can calculate $s\left(idx, d, Alice\right)$ more efficiently than $Alice$, which means that they need to consume less input tokens than $Alice$ for this calculation. $Alice$ can choose to delegate this calculation to a more efficient player Bob and provide the necessary input tokens for his computation with a surplus to compensate *Bob* for his effort. Both players are better off, because *Alice* spent less tokens than she would if she had calculated $s\left(idx, d, Alice\right)$ herself, whilst *Bob* obtained some tokens which can in turn be used to satisfy some of his future desires.

Since *Bob* can commit fraud and get the tokens without satisfying *Alice*'s desire in exchange, *Alice* must be well informed on the trustworthiness of potential collaborators before handing in the tokens. *Alice* can use any out-of-band means to this end, which we model as an oracle $\mathcal{O}_{Trust}$ which is input a desire and a list of potential players and returns the player that increases the utility the most (on expectation), along with the expected utility increase.

Definition 2 (Cost of desire). *The cost of Alice's indexed desire, say* $(idx, d) \in (\mathbb{N}, \mathcal{D})$*, when satisfied by Bob is equal to the input tokens that Alice is required by Bob to give to him in order for him to calculate* $s\left(idx, d, Alice\right)$ *and is represented by* $c\left(idx, d, Alice, Bob\right)$*. The cost of satisfying this desire herself is represented by* $c\left(idx, d, Alice\right)$ *and is equal to the number of input tokens Alice must consume in order to calculate* $s\left(idx, d, Alice\right)$ *herself. Let* $c\left(idx, d, Alice, Alice\right) = c\left(idx, d, Alice\right)$*.*

It is reasonable to assume that there exists an absolute minimum of tokens that must be spent for the satisfaction of a desire, no matter how efficient the calculating party is.

Definition 3 (Minimum cost of desire). *The minimum cost of Alice's indexed desire, say* $(idx, d) \in (\mathbb{N}, \mathcal{D})$*, is equal to the theoretical minimum of input tokens required for the calculation of* $s\left(idx, d, Alice\right)$ *and is represented by* $c_{min}\left(idx, d, Alice\right)$*.*

Note that $1 \leq c_{min}\left(idx, d, Alice\right) \leq \min\limits_{v \in \mathcal{P}} c\left(idx, d, Alice, v\right)$.

Definition 4 (Player of desire function).
The function $Player : \mathbb{N} \times \mathcal{D} \to \mathcal{P}$ *takes as input an indexed desire and returns the player to which this desire was input by the environment.*

Definition 5 (Is the desire satisfied? property).
The property $isSatisfied : \mathbb{N} \times \mathcal{D} \to \{True, False\}$ *takes as input an indexed desire and returns whether it has been satisfied.*

Definition 6 (Desires of Player function).
The function $Desires : \mathcal{P} \to 2^{(\mathbb{N} \times \mathcal{D})}$ *takes as input a player and returns the set of desires that the player was assinged throughout the game.*

$$Desires\left(v\right) = \bigcup_{(idx,d) \in \mathbb{N} \times \mathcal{D}} \left\{(idx, d) : Player\left(idx, d\right) = v\right\}$$

We define $SatDesires : \mathcal{P} \to 2^{(\mathbb{N} \times \mathcal{D})}$ *and* $UnsatDesires : \mathcal{P} \to 2^{(\mathbb{N} \times \mathcal{D})}$ *as well:*

$$\begin{aligned} SatDesires\left(v\right) = \bigcup_{(idx,d) \in \mathbb{N} \times \mathcal{D}} \{(idx, d) : Player\left(idx, d\right) = v \land \\ \land isSatisfied\left(idx, d\right) = True\} \\ UnsatDesires\left(v\right) = \bigcup_{(idx,d) \in \mathbb{N} \times \mathcal{D}} \{(idx, d) : Player\left(idx, d\right) = v \land \\ \land isSatisfied\left(idx, d\right) = False\} \end{aligned}$$

It is straightforward to see that

$$\forall v \in \mathcal{P}, SatDesires\left(v\right) \cap UnsatDesires\left(v\right) = \emptyset \enspace .$$

If additionally we suppose that $isSatisfied\left(\right)$ is always a computable function, then

$$\forall v \in \mathcal{P}, Desires\left(v\right) = SatDesires\left(v\right) \cup UnsatDesires\left(v\right) \enspace .$$

$\mathcal{E}$ can calculate the functions and the property defined above for all inputs at any moment in time.

The game begins with all players being created by $\mathcal{E}$, each allocated a number of tokens determined by the $\mathcal{F}_{Ledger}$ functionality (more on that later). The game ends at a moment specified by the $\mathcal{E}$, which is unknown to the players. At that moment $\mathcal{E}$ assigns a utility to each player depending on which desires were satisfied throughout the game.

Definition 7 (Utility).

$$\forall v \in \mathcal{P}, u_v = f\left(SatDesires\left(v\right), UnsatDesires\left(v\right)\right),$$
$$U = \bigcup_{v \in \mathcal{P}} \{u_v\}$$

3 High-level idea

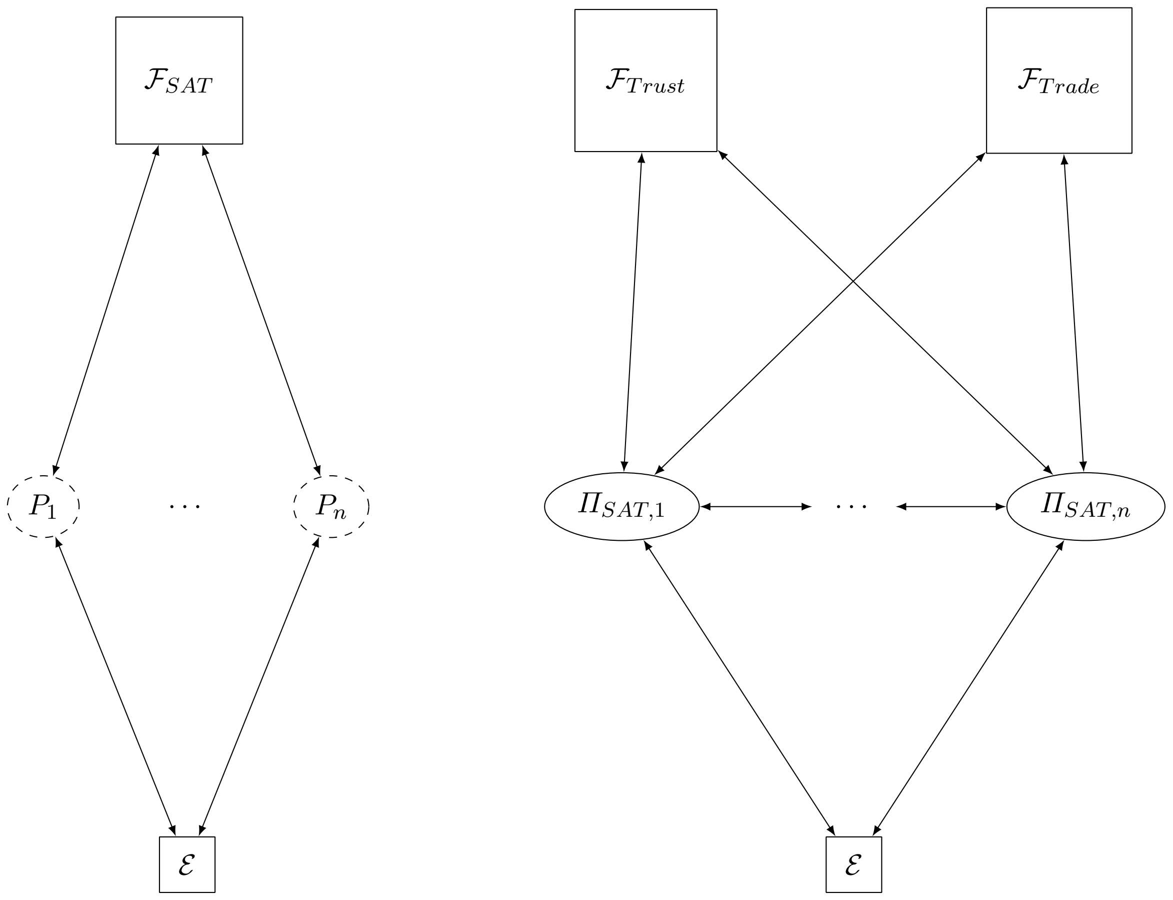

Fig. 1: (Almost) all functionalities

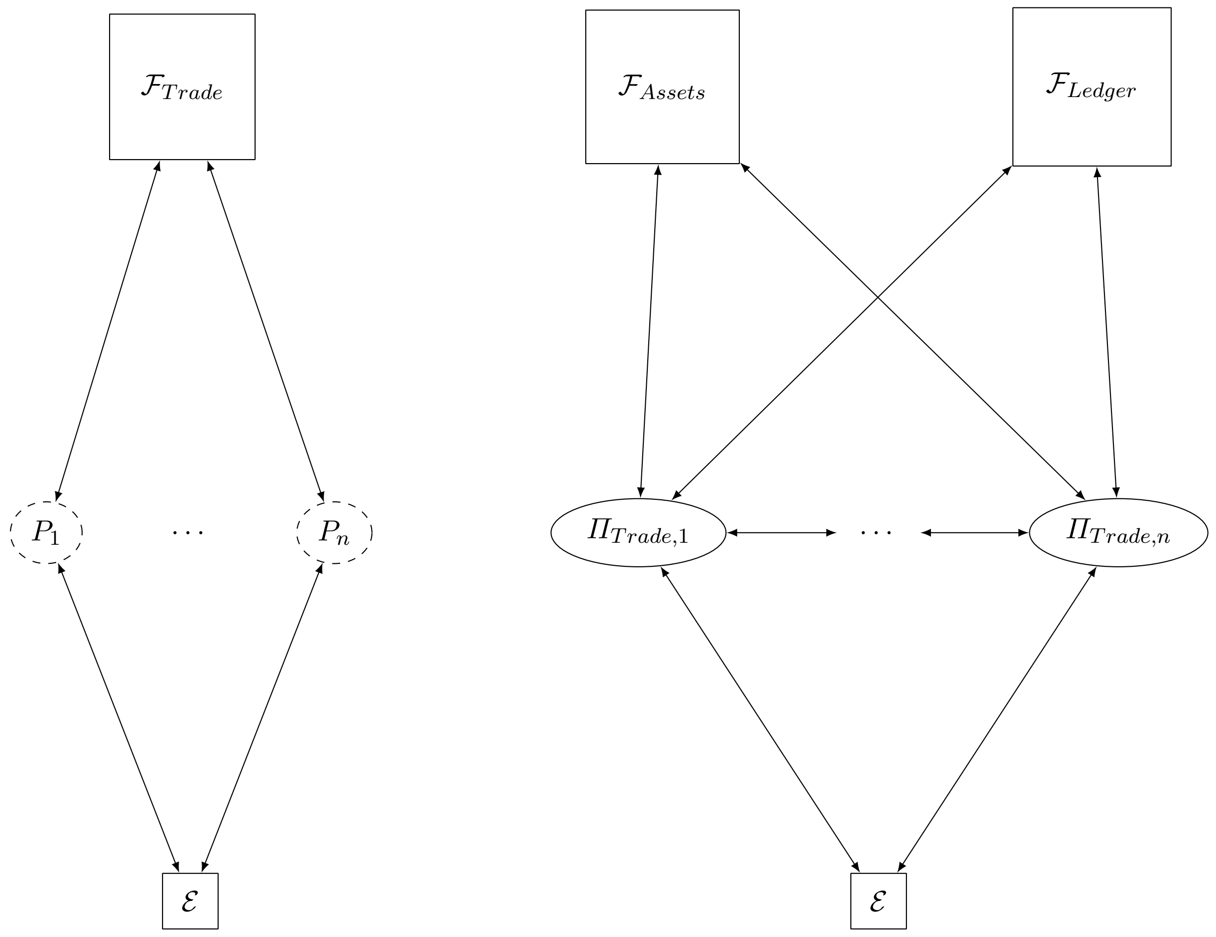

Fig. 2: Trade functionality and protocol

4 Utility Function Properties

Let $v \in \mathcal{P}$. The only property that any utility function should satisfy is that having more desires satisfied and less desires unsatisfied leads to a higher utility. More formally, consider two executions of the game (1) and (2):

$$\left.\begin{array}{r} SatDesires_1(v) \subseteq SatDesires_2(v) \\ UnsatDesires_2(v) \subseteq UnsatDesires_1(v) \end{array}\right\} \Rightarrow u_{v,1} \leq u_{v,2} \enspace .$$

If one of the two subset is strict, then the inequality becomes strict as well.

As an example, consider two executions of the game (1) and (2) in which v was input exactly the same desires and satisfied the same desires, except for (idx, d), which was satisfied only in game (2) and remained unsatisfied in game (1). Then $u_{v,1} < u_{v,2}$. Going in some detail:

$$\text{Let } \mathit{Desires}_1(v) = \mathit{Desires}_2(v) = \{(idx, d)\} \cup \mathit{Rest} \ .$$

$$\text{It is: } \begin{array}{r} \mathit{SatDesires}_1(v) = \mathit{Rest} \ , \\ \mathit{UnsatDesires}_1(v) = \mathit{Rest} \cup \{(idx, d)\} \ , \\ \mathit{SatDesires}_2(v) = \mathit{Rest} \cup \{(idx, d)\} \ \text{and} \\ \mathit{UnsatDesires}_2(v) = \mathit{Rest} \ , \end{array}$$

$$\text{thus: } \left. \begin{array}{r} \mathit{SatDesires}_1(v) \subset \mathit{SatDesires}_2(v) \\ \mathit{UnsatDesires}_2(v) \subset \mathit{UnsatDesires}_1(v) \end{array} \right\} \Rightarrow u_{v,1} < u_{v,2} \ .$$

5 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 $\Pi_1, \dots, \Pi_n$. $\mathcal{E}$ 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:

```
$\Pi\,(\Delta_1, \Delta_2, \Delta_3, \Delta_4)$
Declarations:
  set offeredSats

Initalization:
  offeredSats = $\emptyset$

Upon receiving message satisfy($d$, $L$) from $\mathcal{E}$:
  listen for message abort($d$, $L$) from $\mathcal{E}$
    Upon receiving such message:
      go to Idle State
  aux = $L$
  While aux is not empty
    $Bob$ = chooseBestSeller($d$, aux) # (e.g. with Trust is
        Risk)
    send message canYouSatisfy($d$) to $Bob$
    wait for response1 from $Bob$ for $\Delta_1$ steps
    If response1 $\in \{$timeout, IcannotSatisfy($d$), invalid$\}$
      aux = aux $\setminus \{Bob\}$
    Else If response1 == IcanSatisfy($d$, $x$)
      send message doIhaveBalance($x$) to $\mathcal{F}_{Ledger}$
      wait for responseFromLedger from $\mathcal{F}_{Ledger}$ for $\Delta_2$
      If responseFromLedger == youHaveBalance($x$)
        send message payment($d$, $x$) to $Bob$
        send message payment($Bob$, $x$) to $\mathcal{F}_{Ledger}$
        send message paymentSent($d$, $x$, $Bob$) to $\mathcal{E}$
        wait for response2 from $Bob$ for $\Delta_3$ steps
        If response2 $\in \{$timeout, invalid$\}$
          send message ignored($d$, $x$, $Bob$) to $\mathcal{E}$
        Else If response2 == satString($d$, $x$, $s$)
          If stringSatisfies($d$, $s$)
            send message satisfied($d$, $s$, $Bob$) to $\mathcal{E}$
          Else
            send message partiallySatisfied($d$, $s$, $Bob$) to $\mathcal{E}$
        Else
          (Unreachable)
        stop listening for message abort($d$, $L$) from $\mathcal{E}$
        go to Idle state
    Else
      (Unreachable)
  send message noOneSatisfies($d$, $L$) to $\mathcal{E}$
  stop listening for message abort($d$, $L$) from $\mathcal{E}$

Upon receiving message makeAvailable($d$, $L$, $x$) from $\mathcal{E}$
  offeredSats = offeredSats $\cup\,(d, L, x)$
  send message madeAvailable($d$, $L$, $x$) to $\mathcal{E}$

Upon receiving message makeUnavailable($d$, $L$, $x$) from $\mathcal{E}$
  offeredSats = offeredSats $\setminus\,(d, L, x)$
  send message madeUnavailable($d$, $L$, $x$) to $\mathcal{E}$

Upon receiving message increaseDirectTrust($Alice$, $x$) from $\mathcal{E}$
  $\mathcal{F}_{Ledger}$.increaseDirectTrust($Alice$, $x$)
  send message increasedDirectTrust($Alice$, $x$) to $\mathcal{E}$

Upon receiving message decreaseDirectTrust($Alice$, $x$) from $\mathcal{E}$
  $\mathcal{F}_{Ledger}$.decreaseDirectTrust($Alice$, $x$)
  send message decreasedDirectTrust($Alice$, $x$) to $\mathcal{E}$

Upon receiving message stealDirectTrust($Alice$, $x$) from $\mathcal{E}$
  $\mathcal{F}_{Ledger}$.stealDirectTrust($Alice$, $x$)
  send message stoleDirectTrust($Alice$, $x$) to $\mathcal{E}$

Upon receiving message canYouSatisfy($d$) from $Alice$
  If $\exists (a, b, c) \in$ offeredSats $: (d == a \wedge Alice \in b)$
    send message IcanSatisfy($d$, $c$) to $Alice$ # $c$ is the price
  Else
    send message IcannotSatisfy($d$) to $Alice$

Upon receiving message payment($d$, $x$) from $Alice$
  If $\exists (a, b, c) \in$ offeredSats $: (d == a \wedge Alice \in b)$
    send message correctPaymentReceived($d$, $x$, $Alice$) to $\mathcal{E}$
    listen for message fulfill($d$, $Alice$, $s$) or
        ignore($d$, $Alice$) from $\mathcal{E}$ for $\Delta_4$ steps
      Upon receiving message fulfill($d$, $Alice$, $s$) from $\mathcal{E}$:
        send message satString($d$, $x$, $s$) to $Alice$
      Upon receiving message ignore($d$, $Alice$) from $\mathcal{E}$:
        go to Idle state
  Else
    send message wrongPaymentReceived($Alice$, $d$, $x$) to $\mathcal{E}$
```

6 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 4, 5 and 6 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.

```
$\mathcal{F}_{SAT}(p, \Delta_1, \Delta_2)$
Declarations:
  boolean allSet
  set pendingDesires
  map isSet($\mathcal{P}$ => boolean)
  map price($\mathcal{P}$ => map ($\mathcal{D}$ => number))
  map tokens($\mathcal{P}$ => number)
  map fulfillment($\mathcal{P}$ => map ($\mathcal{D}$ => SAT))
  map utility($\mathcal{P}$ => function(set, set): number)
       # This map has hardcoded values
  map offeredSats($\mathcal{P}$ => set)

Initialization:
  allSet = False
  pendingDesires = $\emptyset$
  $\forall$ $Alice \in \mathcal{P}$
    isSet($Alice$) = False
    price($Alice$) = null
    tokens($Alice$) = $\mathcal{F}_{Ledger}$.getTokens($Alice$)
    $\forall$ desire $\in \mathcal{D}$
      fulfillment($Alice$)(desire) = null

Upon receiving message
      init(map prices($\mathcal{D}$ => number)) from $\mathcal{P}$ $Alice$:
  If isSet($Alice$) == False
    isSet($Alice$) = True
    price($Alice$) = prices
    If $\forall$ $Bob \in \mathcal{P}$, isSet($Bob$) == True
      allSet = True

Upon receiving message cost($\mathcal{P}$ $Bob$, $\mathcal{D}$ desire) from $\mathcal{P}$ $Alice$:
  If allSet == True
    send message price($Bob$)(desire) to $Alice$

Upon receiving message satisfy($d$, $L$) from $\mathcal{P}$ $Alice$:
  If allSet == True
    actPerfectly = flipCoin($p$) # probability $p$ to act
        perfectly
    If actPerfectly == True
      number minPrice = $\min_{Bob \in \mathcal{P}} \{$price($Bob$)($d$)$\}$
      $\mathcal{P}$ $Bob$ = $\operatorname{argmin}_{Charlie \in \mathcal{P}} \{$price($Charlie$)($d$)$\}$
      If tokens($Alice$) $\geq$ minPrice
        tokens($Alice$) = tokens($Alice$) - minPrice
        tokens($Bob$) = tokens($Bob$) + minPrice
        leak message payment($Bob$, $d$, minPrice) to $\mathcal{A}$ # OK?
        send message payment($Bob$, minPrice) to $\mathcal{F}_{Ledger}$
        satString($Alice$)($d$) = $s$($Alice$, $d$)
#     Else If utility($Alice$)(satDesires($Alice$) $\cup$ desire,
    unsatDesires($Alice$) $\setminus$ desire) -
    utility($Alice$)(satDesires($Alice$), unsatDesires($Alice$)) >
    (utility that can be gained for other players for
    minPrice)
#       support = choosePayers(minPrice)
#       $\forall$ ($Bob$, partialPrice) $\in$ support
#         tokens($Bob$) = tokens($Bob$) - partialPrice
#         leak message payment($Bob$, $d$, partialPrice) to $\mathcal{A}$
#         send message payment($Bob$, partialPrice) to $\mathcal{F}_{Ledger}$
#       satString($Alice$)($d$) = $s$($Alice$, $d$)
#     Else # desire remains unsatisfied
#       pendingDesires = pendingDesires $\cup$ ($Alice$, $d$)
#     Attempt to satisfy desires from pendingDesires
     # How? Also, could be $\mathcal{F}_{SAT}$ idle action
    Else # If actPerfectly == False
      listen for message abort($d$, $L$) from $Alice$
        Upon receiving such message:
          go to Idle State
      $L$ = reorderByBestSeller($d$, $L$)
      For $Bob$ in $L$
        If $\exists(a,b,x) \in$ offeredSats($Bob$) : $(d == a \wedge Alice \in b)$
          send message doIhaveBalance($x$) to $\mathcal{F}_{Ledger}$ as $Alice$
          wait for response from $\mathcal{F}_{Ledger}$ for $\Delta_1$ steps
          If response == youHaveBalance($x$)
            send message payment($Bob$, $x$) to $\mathcal{F}_{Ledger}$ as $Alice$
            send message paymentSent($d$, $x$, $Bob$) to $\mathcal{E}$ as $Alice$
            send message correctPaymentReceived($d$, $x$, $Alice$)
                to $\mathcal{E}$ as $Bob$
            listen for message fulfill($d$, $Alice$, $s$) or
                ignore($d$, $Alice$) from $Bob$ for $\Delta_2$ steps
            Upon receiving message fulfill($d$, $Alice$, $s$)
              satString($Alice$)($d$) = $s$
              send message satisfied($d$, $s$, $Bob$) to $\mathcal{E}$ as $Alice$
            Upon receiving message ignore($d$, $Alice$) from $Bob$
              send message ignored($d$, $x$, $Bob$) to $\mathcal{E}$ as $Alice$
            stop listening for message abort($d$, $L$) from $Alice$
            go to Idle state
        send message noOneSatisfies($d$, $L$) to $\mathcal{E}$ as $Alice$
        stop listening for message abort($d$, $L$) from $Alice$

 Upon receiving message makeAvailable($d$, $L$, $x$) from $Alice$
   offeredSats($Alice$) = offeredSats($Alice$) $\cup (d, L, x)$
   send message madeAvailable($d$, $L$, $x$) to $\mathcal{E}$ as $Alice$

 Upon receiving message makeUnavailable($d$, $L$, $x$) from $Alice$
   offeredSats($Alice$) = offeredSats($Alice$) $\setminus (d, L, x)$
   send message madeUnavailable($d$, $L$, $x$) to $\mathcal{E}$ as $Alice$

 Upon receiving message
      getFulfillment($\mathcal{D}$ desire) from $\mathcal{P}$ $Alice$:
   If allSet == True
     send message fulfillment($Alice$)(desire) to $Alice$

 choosePayers(number price) returns set of ($\mathcal{P}$, number)
 # Chooses to charge players that are most likely not to
     benefit by their tokens

 satDesires($\mathcal{P}$ $Alice$) returns set:
   ret = $\emptyset$
   $\forall$ desire $\in$ $\mathcal{D}$
     if fulfillment($Alice$)(desire) $\neq$ null
     ret = ret $\cup$ desire
   return ret

 unsatDesires($\mathcal{P}$ $Alice$) returns set:
   ret = $\emptyset$
   $\forall$ desire $\in$ $\mathcal{D}$
     if fulfillment($Alice$)(desire) == null
     ret = ret $\cup$ desire
   return ret

 # May be redundant
 transfer($\mathcal{P}$ $Alice$, $\mathcal{P}$ $Bob$, number tokens):
   If allSet == True
     If tokens($Alice$) $\geq$ tokens
     tokens($Alice$) = tokens($Alice$) - tokens
     tokens($Bob$) = tokens($Bob$) + tokens
```

$\mathcal{F}_{Trade}$

```
Upon receiving trade(ours, theirs, $Bob$) from $Alice$:
  AliceOK = checkAvailability(ours, $Alice$, $Alice$)
  BobOK   = checkAvailability(theirs, $Bob$, $Alice$)
  If AliceOK and BobOK:
    transfer(ours, $Alice$, $Bob$)
    transfer(theirs, $Bob$, $Alice$)

checkAvailability(object, player, invoker):
  If object is money:
    If player has enough money:
      return True
    Else:
      Send notEnoughMoney(object, player) to invoker
      return False
  Else: # object is asset
    If object is in the list of player's assets:
      return True
    Else:
      Send assetUnavailable(object, player) to invoker
      return False

transfer(object, sender, receiver):
  If object is money:
    send pay(object, receiver) to $\mathcal{F}_{Ledger}$ as sender
  Else: # object is asset
    send transfer(object, receiver) to $\mathcal{F}_{Assets}$ as sender
```

$\mathcal{F}_{Assets}$

```
Upon receiving add($asset$) from $Alice$:
  Add $asset$ to the list of assets belonging to $Alice$
  Send added($asset$) to $Alice$

Upon receiving remove($asset$) from $Alice$:
  If $asset$ exists in $Alice$'s list of assets:
    Remove $asset$ from the list of assets belonging to $Alice$
    Send removed($asset$) to $Alice$
  Else:
    Send unableToRemove($asset$) to $Alice$

Upon receiving doIhave($asset$) from $Alice$:
  If $asset$ exists in $Alice$'s list of assets:
    Send youHave($asset$) to $Alice$
  Else:
    Send youDoNotHave($asset$) to $Alice$

Upon receiving transfer($asset$, $Bob$) from $Alice$:
  If $asset$ exists in $Alice$'s list of assets:
    Remove $asset$ from the list of assets belonging to $Alice$
    Add $asset$ to the list of assets belonging to $Bob$
    Send transferred($asset$) to $Alice$
  Else:
    Send unableToTransfer($asset$, $Bob$) to $Alice$
```

Π_{Trade}

```
Upon receiving trade(ours, theirs, $Bob$) from $\mathcal{E}$:
  If isAvailable(ours, $Alice$):
    Send willYouExchange(ours, theirs) to $Bob$
    Wait for response IwillExchange(ours, theirs) or
        IwillNotExchange(ours, theirs)
    If $Bob$ responded IwillExchange(ours, theirs):
      Send willExchange($Bob$, ours, theirs) to $\mathcal{E}$
      Send letsTrade(ours, theirs) to $Bob$
      transfer(ours, $Bob$)
      Send transferred(ours, $Bob$) to $Bob$ and $\mathcal{E}$
      Wait for response transferred(theirs, $Bob$) from $Bob$

Upon receiving letsTrade(theirs, ours) from $Bob$:
  Send willWeCheat(theirs, ours, $Bob$) to $\mathcal{E}$
  Wait for message action(theirs, ours, $Bob$) from $\mathcal{E}$
  If action is doNotCheat:
    transfer(ours, $Bob$)
    Send transferred(ours, $Bob$) to $Bob$ and $\mathcal{E}$

Upon receiving willYouExchange(theirs, ours) from $Bob$:
  Send willWeExchange(theirs, ours, $Bob$) to $\mathcal{E}$
  Wait for message action(theirs, ours, $Bob$) from $\mathcal{E}$
  If action is exchange:
    Send IwillExchange(theirs, ours) to $Bob$
  Else:
    Send IwillNotExchange(theirs, ours) to $Bob$

isAvailable(object, player):
  If object is money:
    Send queryBalance(player) to $\mathcal{F}_{Ledger}$
    Wait for response balance($coins$)
    If $coins$ >= object:
      return True
  Else: # object is asset
    Send has(object, player) to $\mathcal{F}_{Assets}$
    Wait for response has(object, player)
    return has

transfer(object, sender, receiver):
  If object is money:
    send pay(object, receiver) to $\mathcal{F}_{Ledger}$ as sender
  Else: # object is asset
    send transfer(object, receiver) to $\mathcal{F}_{Assets}$ as sender
```

7 Trust definitions

We define two kinds of trust: direct and indirect. Direct trust from *Alice* to *Bob* is represented by input tokens (initially belonging to *Alice*) actively put by her in a common account from which *Bob* can also take them. As long as *Bob* does not take these tokens, *Alice* directly trusts him equally to the amount of tokens deposited in the common account.

This information can be used by another player *Charlie* that directly trusts *Alice* in order to derive information regarding *Bob*'s trustworthiness, even if *Charlie* does not directly trust *Bob*. Through a transitive property, *Dean*, who directly trusts *Charlie*, can in turn derive information from the direct trust from *Alice* to *Bob*. This reasoning can be extended to an arbitrary number of players, as long as they have at least one trust path to *Alice*. This is called indirect trust.

Definition 8 (Direct Trust). *The direct trust from Alice to Bob, represented by $DTr_{Alice \rightarrow Bob}$, is equal to the total tokens that Alice has given as input to $\mathcal{F}_{Trust}$ with* `entrust(`Bob`, 1`$^{\texttt{tokens}}$`)` *and also equal to the available tokens count sent by $\mathcal{F}_{Trust}$ as a response to a message* `query_direct_trust(`$Alice$`, `Bob`)`.

Definition 9 (Indirect Trust). *The indirect trust from Alice to Bob, $Tr_{Alice \rightarrow Bob}$, is measured in input tokens and can be calculated deterministically given the existing direct trusts between all pairs of players. It is equal to the number sent by $\mathcal{F}_{Trust}$ as a response to a message* `query_indirect_trust(`$Alice$`, `Bob`)`.

By convention $DTr_{Alice \rightarrow Alice} = Tr_{Alice \rightarrow Alice}$ and are both equal to the quantity of input tokens $Alice$ has.

We would like to provide players with an ideal functionality where they:

1. Directly trust tokens to another player
2. Steal tokens previously directly entrusted by another player
3. Retract tokens previously directly entrusted to another player
4. Query indirect trust towards another player

The following functionality provides such an interface:

$\mathcal{F}_{Trust}$

```
Has oracle access to the players' utility functions $U$

Upon receiving chooseBestSeller($d$, $L$) from $Alice$:
  choose $Bob \in L$ such that:
    1) according to his utility function, will complete the
       trade and
    2) offers the minimum price and maximum quality of
       product (i.e. maximises $Alice$'s additional utility)
       amongst the players that satisfy (1)
  send message bestSeller($d$, $L$, $Bob$) to $Alice$
```

Assumption: Trades are atomic. $\mathcal{F}_{Trust}$ can deterministically decide whether Bob will complete the trade.

8 Desired Properties for Indirect Trust

1. $Tr_{Alice \rightarrow Bob} \geq DTr_{Alice \rightarrow Bob}$
2. If universe (1) and (2) are identical except for $DTr_{Alice \rightarrow Bob}$, then

$$Tr^2_{Alice \rightarrow Bob} = Tr^1_{Alice \rightarrow Bob} - DTr^1_{Alice \rightarrow Bob} + DTr^2_{Alice \rightarrow Bob} \ .$$

3. Consider an indexed desire (idx, d) $Alice$ has. Let $p\left(idx, d, Alice\right)$ be a function that returns the player that $Alice$ should rationally delegate the calculation of $s\left(idx, d, Alice\right)$ to.

(a) If a player is cheaper and more trustworthy than all other players, delegate the calculation to him.

$$\begin{gathered}\exists Bob \in \mathcal{P} : \forall Charlie \in \mathcal{P} \setminus \{Bob\} \\ (c\,(idx, d, Alice, Bob) < c\,(idx, d, Alice, Charlie) \wedge \\ \wedge\; Tr_{Alice \to Bob} > Tr_{Alice \to Charlie}) \Rightarrow \\ \Rightarrow p\,(idx, d, Alice) = Bob \;\; .\end{gathered}$$

(b) If there exists a player *Bob* that is both cheaper and more trustworthy than *Charlie*, do not delegate the calculation to *Charlie*.

$$\begin{gathered}\exists Bob, Charlie \in \mathcal{P} : \\ (c\,(idx, d, Alice, Bob) < c\,(idx, d, Alice, Charlie) \wedge \\ \wedge\; Tr_{Alice \to Bob} > Tr_{Alice \to Charlie}) \Rightarrow \\ \Rightarrow p\,(idx, d, Alice) \neq Charlie \;\; .\end{gathered}$$

Note that the first property can be deduced from the second.

Let $x = (idx, d, Alice)$. Several ideas exist as to what rules $p\,(x)$ should satisfy:

1. Indirect trust towards *Bob* is required to be greater than the cost of the calculation requested by *Bob* in order for *Alice* to delegate $s\,(idx, d, Alice)$ to him. If there exist multiple players that *Alice* indirectly trusts more than their cost, then the cheapest one is chosen. If multiple trustworthy enough players have the same cost, the most trustworthy one is chosen.

$$\begin{aligned} &c\,(x, Charlie) > Tr_{Alice \to Charlie} && \Rightarrow p\,(x) \neq Charlie \\ &\text{For the following, } \forall v \in \{Bob, Charlie\}\; c\,(x, v) \leq Tr_{Alice \to v} \;\; . \\ &c\,(x, Bob) < c\,(x, Charlie) && \Rightarrow p\,(x) \neq Charlie \\ &\left.\begin{array}{l} c\,(x, Bob) = c\,(x, Charlie) \\ Tr_{Alice \to Bob} > Tr_{Alice \to Charlie} \end{array}\right\} && \Rightarrow p\,(x) \neq Charlie \end{aligned}$$

2. Similarly to the previous idea, the indirect trust must exceed the cost. In this case however, in case of multiple trustworthy and cheap players, the indirect trust is considered before the cost.

$$\begin{aligned} &c\,(x, Charlie) > Tr_{Alice \to Charlie} && \Rightarrow p\,(x) \neq Charlie \\ &\text{For the following, } \forall v \in \{Bob, Charlie\}\; c\,(x, v) \leq Tr_{Alice \to v} \;\; . \\ &Tr_{Alice \to Bob} > Tr_{Alice \to Charlie} && \Rightarrow p\,(x) \neq Charlie \\ &\left.\begin{array}{l} Tr_{Alice \to Bob} = Tr_{Alice \to Charlie} \\ c\,(x, Bob) < c\,(x, Charlie) \end{array}\right\} && \Rightarrow p\,(x) \neq Charlie \end{aligned}$$

3. The player with the highest difference between indirect trust and cost is chosen.
$$p(x) = \underset{v \in \mathcal{P}}{\operatorname{argmax}} \left(Tr_{Alice \to v} - c(x, v) \right)$$
This aims to maximize trust and minimize cost and, contrary to the previous two approaches, attaches equal importance to these two metrics.
4. The player with the lowest difference between indirect trust and cost is chosen.
$$p(x) = \underset{v \in \mathcal{P}}{\operatorname{argmin}} \left(Tr_{Alice \to v} - c(x, v) \right)$$
Note that this last approach constitutes another direction, which departs from choosing the cheapest and most trustworthy vendor, opting for the one whose price and trustworthiness match. It evidently does not follow the property (3).

9 Motivation for our Trust model

Nevertheless, one can say that at first sight it is in *Bob*'s best interest to trick *Alice* into believing that he can efficiently calculate $s(idx, d, Alice)$ and skip the computation entirely after obtaining *Alice*'s input, thus keeping all the tokens of the defrauded player. Evidently *Alice* would avoid further interaction with *Bob*, but without any way to signal other players of this unfortunate encounter, *Bob* can keep defrauding others until the pool of players is depleted; if the players are numerous or their number is increasing, *Bob* may keep this enterprise very profitable for an indefinite amount of time. This being a rational strategy, every player would eventually follow it, which through a "tragedy of the commons" effect invariably leads to a world where each player must satisfy all her desires by herself, entirely missing out on the prospect of the division of labor.

One answer to that undesirable turn of events is a method through which *Alice*, prior to interacting with an aspiring helper *Bob*, consults the collective knowledge of her neighborhood of the network regarding him. There are several methods to achieve this, such as star-based global ratings. This method however has several drawbacks:

- Very good ratings cost nothing, thus convey little valuable information.
- Different players may have different preferences, global ratings fail to capture this. Arrow's impossibility theorem is possibly relevant here.

- Susceptible to Sybil attacks; mitigation techniques are ad-hoc and require (partial) centralization and secrecy/obfuscation of methods to succeed, thus undermining the decentralized, transparent nature of the system, a property that we actively seek.

References