Abstract

Almost two centuries ago Pierre-Joseph Proudhon proposed social contracts — voluntary agreements among free people — as a foundation from which an egalitarian and just society can emerge. A digital social contract is the novel incarnation of this concept for the digital age: a voluntary agreement between people that is specified, undertaken, and fulfilled in the digital realm. It embodies the notion of “code-is-law” in its purest form, in that a digital social contract is in fact a program — code in a social contracts programming language, which specifies the digital actions parties to the social contract may take; and the parties to the contract are entrusted, equally, with the task of ensuring that each party abides by the contract. Parties to a social contract are identified via their public keys, and the one and only type of action a party to a digital social contract may take is a “crypto-speech act” — signing an utterance with her private key and sending it to the other parties to the contract.

Here, we present a formal definition of a digital social contract as agents that communicate asynchronously via crypto-speech acts, where the output of each agent is the input of all the other agents. We outline an abstract design for a social contracts programming language and show, via programming examples, that key application areas, including social community; simple sharing-economy applications; egalitarian currency networks; and democratic community governance, can all be expressed elegantly and efficiently as digital social contracts. Possible extensions, described in companion papers, include autonomous deterministic agents akin to “smart contracts”, and joint agents executed jointly by several parties to the contract; a definition of a distributed implementation of digital social contracts in the presence of faulty agents; and egalitarian and just currency networks, suitable for realization via a network of digital social contracts.

1 Introduction

Digital technology, as exemplified by social networks and “sharing economy” platforms, has been the great unequalizer: Making a few digital barons rich and powerful, while keeping the multitudes performing digital labour (e.g., producing content and rating products and services) for the benefit of the barons, without compensation. Like feudal lords of yore, the digital barons set their digital community’s rules of conduct (legislative), their interpretation (judicial) and implementation (executive), practicing surveillance capitalism [17] and violating the dictum: “no taxation without representation”. (We note that users must have ticked a box saying they agree to
all this.) Digital social contracts are a novel incarnation of the old but viable concept of social contracts; we hope it would provide a foundation for an alternative to today’s array of feudal digital communities – an egalitarian and just digital society.

Digital social contracts can be investigated using multiple intellectual disciplines and can be described at multiple levels of abstraction:

1. **Philosophy, Political Theory**: A digital social contract is the digital counterpart of social contracts, as envisioned almost two centuries ago (1851) by Pierre-Joseph Proudhon: “What really is the Social Contract? An agreement of the citizen with the government? No, that would mean but the continuation of [Rousseau’s] idea. The social contract is an agreement of man with man; an agreement which must result in what we call society.” [10]. A digital social contract is a voluntary agreement between people, which is specified, undertaken, and fulfilled in the digital realm. Like Proudhon, we aim for digital social contracts that are equal (in sharing power) and just (in allocating resources) among the people participating in the contract. In the digital realm, equal sharing of power, namely democratic governance, requires mitigating sybils (fake and duplicate identities) [14]. Our team’s work on sybil-resilient social choice [14] and sybil-resilient community formation [9] provide a foundation for drafting social contracts for forming, growing, and democratically governing egalitarian digital communities. Our companion work on egalitarian and just currency networks [15] provides a foundation for launching novel cryptocurrencies, in which distributed and egalitarian coin minting provides a form of Universal Basic Income to the participants. Our companion paper on distributed fault-tolerant implementation of digital social contracts [6] provides a foundation for realizing digital social contracts on mobile phones. Combined, digital social contracts may provide a foundation from which an egalitarian and just digital society may emerge.

2. **Legal Theory**: Digital social contracts embody the notion of “code-is-law” [5] in its purest form, in that a digital social contract is in fact a program – code in a social contracts programming language, which specifies the digital actions parties to the social contract may take. But, here this code is not inflicted on the user by a monopolistic digital baron to its financial benefit; rather, people that know and trust each other enter into this code-regulated relationship voluntarily, using a social contract tried by others and/or written by a trusted expert. Furthermore, there is no outside jurisdiction to such a contract – the parties to the contract are entrusted, equally, with the task of ensuring that each party abides by the contract. Naturally, one of the actions that a party to a digital social contract may take is the transmittal of a duly-signed digital document that is legally binding in a particular jurisdiction (e.g. confirmation for a room reservation). The recipient of such a document may later choose to commence legal proceedings in that jurisdiction against the signatory of the document, outside the realm of digital social contracts.

3. **Linguistics, Cryptography**: A digital social contract specifies the speech acts [11] parties to the contract may take, possibly in response to speech acts by the other parties, digitally. A digital speech act by a party, defined herein, is a sequentially-indexed and cryptographically-signed utterance, sent to the other parties to the contract. Cryptographic signatures of indexed actions ensure, on the one hand, that their recipient can identify the acting party and determine the order of her actions and, on the other hand, prevent the acting party from later repudiating her actions or their order.

4. **Mathematics of Computation**: A digital social contract is an abstract model of computation, defined herein – a transition system that defines a set of interacting automata, one for each party to the contract. Each automaton is a nondeterministic, potentially infinite, state transducer; its output is a string of digital speech
acts, and its input is the output strings of the other automata. As a mathematical model of computation, a transition system may be equivalently defined by a program in suitable high-level programming language, with which digital social contracts may be easier to express, comprehend and debug, as discussed next.

5. **Programming Languages**: A digital social contract is a distributed nondeterministic program in a Social Contracts Programming Language, a conceptual design of which is presented herein. A program in the language specifies the roles of the parties to the contract and the rules that govern their behavior. A program rule specifies an output action a party in a role may take, given its state and an input action by another party to the contract. Nondeterministic choices in the program are presented as a query to the human operator of the digital party: Which room to book? What payment to make and to whom? How to vote? And resolved according to her answer. Digital social contracts can be programmed for a variety of applications, including social networks, political communities, cryptocurrencies, and sharing economy, all potentially endowed with democratic governance. We present basic examples of social contracts in these domains, expressed in the Social Contracts Programming Language, illustrating their underlying concepts as well as the expressive power of the language.

6. **Distributed Systems**: A digital social contract defines the possible interactions of a distributed group of agents equipped with a public-key infrastructure and connected via a reliable asynchronous communication network, each broadcasting digital speech acts and receiving such from others, with a bounded fraction of these agents possibly being faulty (Byzantine). In a companion paper \[6\] we present a fault-tolerant distributed transition system that implements the abstract model of digital social contract presented herein, given a bound on the fraction of faulty agents; it provides the mathematical foundation for a novel blockchain technology, discussed next.

7. **Blockchain Technology**: Digital social contracts can be viewed as programs for a novel, distributed, locally-replicated, asynchronous blockchain architecture \[6\]. Agents are expected to employ genuine identifiers \[13\] and function as both actors and validators. An agent may participate simultaneously in multiple social contracts; similarly, each contract may have a different set of agents as parties; the result is a hypergraph with agents as vertices and contracts as hyperedges. Signed indexed transactions (digital speech acts), carried out according to a contract, are replicated only among parties to a contract, who ratify them, and are synchronized independently by each agent. A transaction carried out according to a contract is finalized when ratified by an appropriate supermajority of the parties to the contract. A party to a contract is typically an agent who embodies a natural person via her genuine identifier, but, if deterministic, could also be an autonomous agent, which functions algorithmically, similarly to a “smart contract” of standard blockchains. Digital social contracts realize the blockchain paradigm of “code is law” \[5\] in its purest form, in that a digital social contract is in fact a program – code in a social contracts programming language, which specifies the digital actions parties to the social contract may take; and there is no outside jurisdiction to such a contract – the parties to the contract are entrusted, equally, with the task of ensuring that each party abides by the contract. Parties to a social contract are identified via their public keys, and the one and only type of action a party to a digital social contract may take is a “crypto-speech act” – signing an utterance with her private key and sending it to the other parties to the contract. A faulty party to a social contract may deviate from the contract’s program, e.g. if controlled by a malicious program; but, if the number of faulty parties are below the designated threshold, then faulty actions will never get finalized and the faulty party
will be identified and suspended.

8. **Social Networks**: Digital social contracts can be a platform for sovereign, egalitarian and just social networks. Sovereign, in that the social network is operated by and under the full control of its members. Egalitarian, in that control is shared equally among the members. Just, in that revenues, particularly from the consumption of advertisements and commercial content, are disbursed directly to the members that “labour” in producing the content that attracts views and/or to the members consuming said advertisements and content. We present below simple example of such digital social contracts.

9. **Cryptocurrencies**: Digital social contracts could be a platform for an egalitarian, just, grassroots, and environmentally-friendly network of cryptocurrencies [15], based on the notion of genuine identifiers [13] and employing a distributed, asynchronous and programmable blockchain technology [6]. Such a cryptocurrency network will realize distributive justice since coin minting is carried out equally by all members, providing a form of Universal Basic Income to each party to a currency contract.

10. **Sharing Economy**: Digital social contracts can be a platform for an egalitarian and just sharing economy. Using autonomous agents (aka smart contracts) as aggregators and intermediators, communities of buyers and sellers, or of service providers and consumers, can cooperate and share the costs and benefits of aggregation and intermediation. In existing “sharing economy” platforms, the digital baron takes the rent for aggregation and intermediation. Sharing-economy digital cooperatives offer an alternative in which rent is justly shared by the members of the cooperative, who also govern its operation, equally.

| Role   | Actions                                      |
|--------|----------------------------------------------|
| None   | 1. Initiate a community as a manager         |
|        | 2. Join a community as an member             |
| Manager| 1. Invite/remove a member                    |
|        | 2. Do anything a member can do               |
| Member | 1. See the manager and members               |
|        | 2. See messages sent by others               |
|        | 3. Send a message                            |
|        | 4. Leave the community                       |

Table 1: The **Community** Social Contract.

A fundamental tenet of our design is that social community communication rely on servers controlled by the service provider, which undermine the sovereignty of the community, let alone its privacy. The **community** digital social contract has roles and actions as detailed in Table[1]. It specifies a private social community, of which the manager is the sovereign; anyone can accept, reject or ignore an invitation to become a member in a community. Turning this autocratic digital social contract into an egalitarian one is discussed herein.

1.2 **Related Work**

The concepts and design presented here are reminiscent of the notions of blockchains [16], smart contracts [1], and their programming languages [2]. Hand-in-hand with these we are working on egalitarian currency networks [15], an egalitarian and just alternative to existing plutocratic cryptocurrencies such as Bitcoin [7] and Ethereum [2].

A fundamental tenet of our design is that social
contracts are made between people who know and trust each other, directly or indirectly via other people that they know and trust. This is in stark contrast to the design of cryptocurrencies and their associated smart contracts, which are made between anonymous and trustless accounts. A challenge cryptocurrencies address is how to achieve consensus in the absence of trust, and their solution is based on proof-of-work [3] or, more recently, proof-of-stake [8] protocols. In contrast, social contracts are between known and trustworthy individuals, each expected to possess a genuine (unique and singular) identifier [13] (see therein discussion on how this can be ensured). Hence, a different approach can be taken. In our approach, the integrity of the ledger of actions taken by the parties to the social contract is preserved internally, among the parties to the agreement, not between external anonymous “miners”, as in cryptocurrencies. This gives rise to a much simpler approach to fault tolerance.

In particular, and as discussed in the companion paper [6] our approach does not suffer from forks/delayed finality as for example the Bitcoin protocol [7], and does not need to reach Byzantine Agreement [4]. Instead, agents ratify actions of each other; an action is final when a supermajority of the agents ratify it; and agents take an action that depends on actions of others only once they are final. A consequence of our approach is that the handling of “double spend”, the key challenge for cryptocurrencies, is simpler: At most one, but possibly none, of the double actions is finalized, and the agent taking the double action is eventually declared faulty, resulting in further non-final actions of that agent being ignored.

1.3 Paper Structure

Next we describe a formal, descriptive model for digital social contracts, as well as a possible design for a programming language to program social contracts in Section 2. We then outline several examples of social contracts in Section 3 specifically, we describe a general design using the programming language for three broad settings: social networks, shared economy and sovereign currencies. We then discuss the realization of egalitarian social contracts and democratic governance in Section 4 and conclude with intriguing questions for further research.

2 Digital Social Contracts

Here we describe a formal model for digital social contracts. Note that we assume that all agents are non-faulty. A companion paper [6] addresses faulty agents.

2.1 Preliminaries

We assume a given finite set of agents $V$, each associated with a genuine (unique and singular) [13] identifier, which is also a public key of a key-pair. We expect agents to be realized by computer programs operating on personal devices (e.g. smartphones) of people. Hence, we refer to agents as “it” rather than as he or she.

We identify an agent $v \in V$ with its genuine public identifier, and denote by $v(s)$ the result of agent $v$ signing the string $s \in S$ with the private key corresponding to $v$. We assume computational hardness of the public-key system, namely that signatures of an agent with a given identifier cannot be produced without possessing the private key corresponding to this identifier. To avoid notational clutter we do not distinguish between finite mathematical entities and their string representation. Identifying agents with their genuine identifiers makes $V$ totally ordered (by the numeric value of the identifier, namely the public key) and hence allows defining tuples and Cartesian products indexed by $V$. If $t$ is a tuple indexed by $V$, then we use $t[v]$ to refer to the $v^{th}$ element of $t$. We say that $t' = t$, except for $t'[v] := x$, to mean that the tuple $t'$ is obtained from $t$ by replacing its $v^{th}$ element by $x$. In particular, if $t[v]$ is a sequence, then we say that $t' = t$, except for $t'[v] := t[v] \cdot x$, to mean that $t'$ is obtained from $t$ by appending $x$ to the sequence $t[v]$.

---

1 We identify the set of agents $V$ with the set of parties to the agreement. Extensions will allow an agent to be a party to multiple agreements, and different agreements to have different sets of agents as parties.
All agents are assumed to be connected via a reliable asynchronous communication network (without assuming a known time limit on message arrival), and require all messages to be authenticated. Informally, the only things an agent can do as a party to a digital social contract are (i) perform a crypto-speech act [13], defined next; (ii) observe crypto-speech acts performed by others; and (iii) change internal state.

Definition 1 (Crypto-Speech Act, v-act). Given a set of agents \( V \), a crypto-speech act of agent \( v \in V \) consists of (i) signing an utterance (text string) \( s \), resulting in \( m = v(s) \); and (ii) broadcasting the message \( m \) to \( V \). We refer to a crypto-speech act by \( v \) resulting in the signed action \( m \) as the \( v \)-act \( m \), and let \( \mathcal{M} \) be the set of all \( v \)-acts for all \( v \in V \).

We employ a standard notion of a transition system.

Definition 2 (Transition System). A transition system \( TS = (S, s_0, T) \) consists of a set of states \( S \), an initial state \( s_0 \in S \), and a set of transitions \( T \subseteq S \times S \), with \((s, s') \in T\) written as \( s \to s' \). The set \( s \to * = \{ \hat{s} \mid s \to \hat{s} \in T \} \) is the outgoing transitions of \( s \). A run of \( TS \) is a sequence of transitions \( r = s_0 \to s_1 \to \ldots \) from the initial state.

2.2 Digital Social Contracts

Here we define digital social contracts in the abstract, in the sense that they do not address distributed realization nor agent faults. These issues are addressed in a companion paper [6].

Remark 1 (Parameterized Social Contracts). Agents should be able to participate simultaneously in multiple social contracts. Furthermore, a contract may have multiple parties with the exact same role. To address this properly, we should describe digital social contracts as a set of roles, each specified by a parameterized procedure, and bind the formal parameters to actual agent identifiers upon execution of the contract. This is akin to the standard legal practice of using a textual contract template, with role names as parameters (e.g. Landlord, Tenant), and filling in the identities of the parties assuming these roles in an instance of this contract template upon its signature. We defer this distinction between formal and actual parameters in a social contract to avoid notational clutter, and name the parties to the social contract by their genuine identifiers, i.e. their public keys. A practical social contracts programming language, which we expect to design and implement, will naturally make this distinction.

We assume a given set of actions \( A \). In the following, actions performed by an agent are first indexed and then signed. Signing an action by an agent \( v \) makes the action non-repudiated by \( v \). Signing indexed actions also makes the order of the actions of \( v \) non-repudiated by \( v \). All that a party to a digital social contract does, then, is perform a sequence of indexed crypto-speech acts, resulting in a non-repudiated history of the acts, defined next. In the automata view of a social contract, the history of a agent is its output tape, which is also the input tape of all the other agents.

Definition 3 (Agent History). An agent history of \( v \in V \) is a finite sequence of \( v \)-acts \( m_0, m_1, \ldots, m_n \), \( n \in \mathbb{N} \), of signed indexed actions \( m_i = v((i, a_i)) \), \( i \in [n] \), \( a_i \in A \). The set of all agent histories of \( v \) is denoted by \( H_v \).

The following definition formalizes the notion of a ledger of a digital social contract, which is but a set of the histories of the parties to the contract.

Given a sequence \( s = x_1, x_2, \ldots, x_n \), a sequence \( s' \) is a prefix of \( s \) if \( s' \subseteq s \), i.e. \( s' = x_1, x_2, \ldots, x_k \), for some \( k \leq n \).

Definition 4 (Ledger, Prefix, Agent View). A ledger \( l \in \mathcal{L} \) is a tuple of agent histories indexed by the agents \( V \), \( \mathcal{L} := \prod_{v \in V} H_v \). In such a ledger \( l \), \( l[v] \in H_v \) is \( v \)'s history in \( l \). Given a ledger \( l \), a \( v \)-view \( l_v \) is a ledger indexed by \( v \). Given a ledger \( l \), \( l_v \) is a \( v \)-view of \( l \) if \( l_v \leq l \) and \( l_v[v] = l[v] \). Two agent views \( l_u, l_v \) are consistent if \( l_u[v] \leq l_v[v], l_u[u] \leq l_v[u] \), and \( l_u[w] \leq l_v[w] \) or vice versa for every \( w \neq u, v \in V \).
Note that each agent independently indexes its own messages, so a ledger is not a linear data structure, like a blockchain, but a tuple of independent agent histories, each of them being a linear sequence of acts. Also note that a \( v \)-view of a ledger \( l \) may be behind on the histories of other agents but is up-to-date regarding its own history. And two views are consistent if they agree on their shared prefixes of agent histories, and each is most up-to-date about its own agent history. Relating this model to finite state transducers, think of \( l[v] \), the history of \( v \in V \), as a tape that is both the output tape of \( v \) and an input tape of all other agents/automata \( u \neq v \in V \). These automata are communicating in the sense that the output of one is the input of all others, and asynchronous in that each automaton reads its input tapes, namely the output tapes of all the others, independently of the others – at a different pace and possibly at a different relative order. In particular, \( v \)-view of \( l \) consists of its own output as well as all the outputs of the other agents it has already read.

Remark 2. Note that in the present asynchronous model, neither agent histories nor ledgers have a built-in notion of time. We would like, however, social contracts to be able to have a notion of time and refer to it. Adding time to digital social contracts is an anticipated future extension.

In automata-theoretic terms, the ledger \( l \) represents the portion of the tapes that were already read by \( v \) (\( l[u] \), \( u \neq v \in V \)) and written by \( v \) (\( l[v] \)).

**Definition 5 (Agent Transition).** Given a ledger \( l \in L \), an agent \( v \in V \), and a \( v \)-view \( l_v \) of \( l \), a \( v \)-transition \( t = l_v \to l'_v \in L \times L \) satisfies that \( l_v \) is \( v \)-view of \( l \), and one of the following holds:

1. **Input** \((m)\): \( l'_v = l_v \) except for \( l'_v[u] := l_v[u] \cdot m \) for some \((|l_v[u]| + 1)\)-indexed \( u \)-act \( m \), \( u \neq v \in V \), provided that \( l'_v \preceq l \).

2. **Output** \((m)\): \( l'_v = l_v \) except for \( l'_v[v] := l_v[v] \cdot m \) for some \((|l_v[v]| + 1)\)-indexed \( v \)-act \( m \).

In which case we say that \( t \) is **enabled** by \( l \).

**Definition 6 (Monotonicity).** Let \( T_v \) be a set of \( v \)-transitions. We say that \( T_v \) is **monotonic** if for every

\[
\text{Input}(m): \quad l_v \to l'_v \in T_v
\]

\[
\text{Output}(m): \quad l_v \to l'_v \in T_v
\]

**program rule:**

\[
S, m \to v', S', \quad \text{if Conditions}
\]

Figure 1: Agent States, Transitions and Programs: A \( v \)-act \( m_v \), a history, a ledger \( l \), an input transition (blue) inputs the \( u \)-act \( m \), adding it to the history \( l[u] \) of \( u \) in \( l \); an output transition (red) that outputs the \( v \)-act \( m' \), adding it to the history \( l[v] \) of \( v \) in \( l \), resulting in \( l' \), changes the ledger of \( v \) from \( l \) (to left of grey line) to \( l' \) (to left of blue or red line, respectively). The **Input** transition followed by the **Output** transition can be abbreviated in a social contract program by a rule \( S, m \to m', S' \), in which \( S \) summarizes \( l \) and \( S' \) summarizes \( l' \) (blue and red combined).
transition $t \in T_v$ and every two ledgers $l \preceq l'$ for which $l[v] = l'[v]$, if $l$ enables $t$ then $l'$ enables $t$.

We require the set of agent transitions to be monotonic.

A $v$-view $l_v$ and an input and output $v$-transitions are depicted in Figure 1.

**Definition 7** (Ledgers Configuration, Consistent, Diagonal). A ledgers configuration $L$ is a $V$-indexed tuple of agent views, $L \in \mathcal{L}^{|V|}$, with $l_v := L[v]$. A ledgers configuration is consistent if its views are pairwise consistent, namely $l_v$ is consistent with $l_v$ for every $u, v \in V$. The diagonal of $L$, $\bar{L}$, is the ledger defined by $\bar{L}[v] := l_v[v]$ for every $v \in V$.

In the automata view of social contracts, a ledger $l_v$ in a ledgers configuration $L$ records the output tape $l_v[v]$ of each agent $v$, where the writing head of an agent is always next to its last output, as well as the location of the reading heads of all agents, where the reading head of agent $v$, reading the output tape of agent $u$, is after the last symbol of $l_u[u]$, which is always a prefix of the output tape of $u$, $l_u[u]$.

**Observation 1** (Consistency and the Diagonal). Given a ledgers configuration $L$, $L$ is consistent iff $l_v \preceq \bar{L}$ for every $v \in V$.

**Definition 8** (Digital Social Contract). A digital social contract $SC = (S, T)$ is a transition system with ledgers configurations as states $S \subseteq \mathcal{L}^{|V|}$, initial state $L_0 := \Lambda^{|V|}$, and transitions $T \subseteq \mathcal{L}^{|V|} \times \mathcal{L}^{|V|}$, where $T = \bigcup_{v \in V} T_v$, $T_v$ is monotonic, and for each $L, L' \in T_v$, $L$ and $L'$ are consistent and $L' = L$ except for $l_v'$, which is the result of a $v$-transition $l_v \rightarrow l_v'$ enabled by $\bar{L}$.

Note that consistency of $L'$ requires that an input $v$-transition does not "invent" an input $u$-act $m; v$ can only employ as input a $u$-act that has already been taken by $u$. Furthermore, it can only input actions in the order in which they were performed. If $v$ takes as input a $u$-act $m$ which $u$ has not taken yet, or is if $m$ out-of-order, and records $m$ in the history of $u$ in its view $l_u[u]$, this will result in an inconsistent ledgers configuration, as $l_u[u]$ will no longer be a prefix $l_u[u]$ and hence not a prefix of the diagonal $\bar{L}$.

This completes the definition of the computational model of Digital Social Contracts. Its distributed fault-tolerant implementation is discussed in a companion paper [6].

### 2.3 A Social Contracts Programming Language

Here we outline a design of a social contract programming language. Abstractly, a transition of an agent depends on its view of the ledger, namely a record of the crypto-speech acts of the parties to the contract the agent has input and output. To be practical, such a history is condensely summarized in the local state of the agent. The state can change as a result of an input and/or output transition.

Hence we view the program of an agent $v$ as consisting of a set of rules of the following form:

$$S, m \rightarrow m', S', \text{ if Conditions}.$$  

where $S$ is the internal state of $v$, $m$ is a $u$-act by some $u \in V$ received by an input transition of $v$, $m'$ is a $v$-act taken by an output transition of $v$, $S'$ is the updated internal state of $v$, and Conditions are any conditions on $S$, $m$, $m'$, and $S'$, that are required for the input and output transitions to take place. Note that degenerate rules, i.e., rules that do not involve input or output, are of course possible. The informal operational semantics of such a rule is given in Figure 2. We note that the local state $S$ of the agent $v$ is a digest of its view of the ledger $l$. In particular, if the rules of $v$ have finitely many states, this means that the $v$ has only limited memory and hence cannot "look back" arbitrary deeply into its ledger.

If we were to use a syntax closer to conventional procedural programming languages, then a smart contracts program might look like this:

**when in $S$:**

  - upon $m_1$, if Conditions $\rightarrow m', S'$;
  - upon $m_2$, if Conditions $\rightarrow m', S'$;

**when in $S_2$:**

  - upon ...

However, since we wish to retain the close relation between program rules and transitions, as well as the
modularity that comes with this notation, we will stick with the rule-based notation in this paper.

The program of an agent is **nondeterministic**, in the sense that, if several rules are enabled by a given state and message, then any one of them may be chosen. The intention of this design is that this nondeterminism be resolved by the person operating the computational agent: Abstractly, a nondeterministic choice faced by an agent would use the person operating the agent represents as its oracle in making the nondeterministic choice. Concretely, when faced with a nondeterministic choice, an agent would use a structured dialogue box to present the person operating it (the “user”) with a query, and the user would respond with her answer, the choice. This notion of nondeterminism as user interface and the user as the nondeterministic oracle will be made clearer via the examples below.

### 3 Examples of Digital Social Contracts

In this section we demonstrate the concepts and utility of digital social contracts via examples. These examples sketch highly simplified settings in which social contracts may allow agents to interact, and may be partitioned to three main thematic categories: social networks, shared economy, and sovereign currencies. The significance of social contracts stems from the fact that they allow the agents to retain the sovereignty, transparency, and privacy over the medium of interaction.

While we depict a direct interaction among the agents in each of these settings, a natural next step is the consideration of intermediators as market aggregators or brokers that may ease the interaction. Initially, these intermediators can be presented as natural agents, but later, ideally, they should be autonomous agents (aka smart contracts) that are operated by the community and for the benefit of the community. The use of autonomous agents operated jointly by the entire community, allows arbitrage profits to be shared among all members. More generally, we can have a buyers cooperative and a sellers cooperative, each as a separate social contract. Such cooperatives may decide to transact, so that there is a network of transacting buyer and seller cooperatives. We leave the exploration of autonomous agents to future work.

#### 3.1 An Online Lodging Marketplace

Here we show an Airnbn-like social contract. It is highly simplified, just to illustrate the concept:

1. There is a bunch of tourists and a bunch of hosts; initially all hosts are free.
2. A tourist may request a room from a host and wait; if the room is free, then the host grants the request and records the room as booked; if the room is booked, then the host denies the request.
3. If the request is granted, then the tourist checks in, and then checks out; when the tourist checks out, the host records that the room is free. If the request is denied, then the tourist may request a room again, from the same host or from another host.
4. No money exchange here.

Here is the Simple-Airbnb social contract. The actions are messages of the form \((\text{Addressee}, \text{Content})\). These messages are contract-level speech acts, visible to all parties, although in the Simple-Airbnb social contract program they are ignored by all parties except the addressee.

First, the host \(v\): The state of the host \(v\) can be either \(\text{Host}(v)\) – in which \(v\’s\) room is free; or \(\text{Booked}(v,u)\) – in which \(u\) currently lives in \(v\’s\) room. If an \(\text{Host}(v)\) received a \text{reserve} request, it grants it, and becomes \(\text{Booked}\). When the agent that reserved the room checks out, the the booked owner becomes \(\text{Host}(v)\) again. Here are the rules of the host \(v\):

**Program 1: Host**

\[
\begin{align*}
\text{Host}(v), (v,u(\text{reserve})) \rightarrow & (u,v(\text{granted})), \text{Booked}(v,u) \\
\text{Booked}(v,u), (v,u(\text{checkout})) & \rightarrow \text{Host}(v) \\
\text{Booked}(v,u), (v,w(\text{reserve})) & \rightarrow (w,v(\text{denied})), \text{Booked}(v,u).
\end{align*}
\]
Now, the tourist \( v \): The state of the tourist \( v \) can be either \( \text{Tourist}(v) \) – in which \( v \) currently does not live in any room; \( \text{Waiting}(v,u) \) – in which \( v \) is waiting for \( u \)'s answer to \( v \)'s request for \( u \)'s room; or \( \text{Renting}(v,u) \) – in which \( v \) currently lives in \( u \)'s room.

Here are the rules for a tourist \( v \): The first two rules are of the form \( S \rightarrow m', S' \), i.e., missing an incoming message \( m \), so this rule can apply whenever \( v \)'s state is \( S \). Note that a \( \text{Tourist} \) nondeterministically chooses a room owner to reserve a room from.

In our interpretation, this nondeterminism is realized as a query to the person operating this agent, which room would you like to reserve? If and when the person chooses which room \( u \) to reserve, the agent then requests to reserve the room, sending the message \( (u, v(\text{reserve})) \). The agent then waits for the response. If the reservation is granted it checks in at its leisure. If the request is denied, it may choose again a room owner to reserve from.

**Program 2: Tourist**

\[
\begin{align*}
\text{Tourist}(v) & \rightarrow (u, v(\text{reserve})), \text{Waiting}(v,u). \\
\text{Waiting}(v,u), (v, u(\text{denied})) & \rightarrow \text{Tourist}(v). \\
\text{Waiting}(v,u), (v, u(\text{granted})) & \rightarrow \text{Renting}(v,u). \\
\text{Renting}(v,u) & \rightarrow (u, v(\text{checkout})), \text{Tourist}(v).
\end{align*}
\]

This ends the program description. We now add a broker that accumulates a list of reservations and a list of free rooms, assume that all rooms are equivalent, and grants reservations on a first-come-first-served basis, under certain fairness assumptions (to be spelled out subsequently). As the broker has to maintain a list of reservations and a list of available hosts, we use the Prolog list notation, with [] (read “nil”) denoting the empty list, and [x—xs] denoting a list with the first element \( x \) and the rest of the list \( xs \).

**Program 3: Brokered Hosts/Tourists Social Contract**

\[
\begin{align*}
\text{Host}(v) & \rightarrow v(\text{free}), \text{Free}(v). \\
\text{Free}(v), \text{broker}(\text{granted}(v,u)) & \rightarrow \text{Booked}(v,u). \\
\text{Booked}(v,u), (v,u(\text{checkout})) & \rightarrow v(\text{free}), \text{Free}(v). \\
\text{Tourist}(v) & \rightarrow v(\text{reserve}), \text{Waiting}(v). \\
\text{Waiting}(v), \text{broker}(\text{granted}(u,v)) & \rightarrow \text{Renting}(v,u). \\
\text{Renting}(v,u) & \rightarrow (u, v(\text{checkout})), \text{Tourist}(v). \\
\text{Broker}(broker) & \rightarrow \text{Broker}(\text{broker},[],[]). \\
\text{Broker}(\text{broker},\text{Rooms},\text{Reservations}), v(\text{free}) & \rightarrow \text{Broker}(\text{broker},\text{Rooms}',\text{Reservations}) \\
& \quad \text{if Rooms'} \text{ is the result of appending } v \text{ to Rooms}. \\
\text{Broker}(\text{broker},\text{Rooms},\text{Reservations}), \text{v(\text{reserve})} & \rightarrow \text{Broker}(\text{broker},\text{Rooms}',\text{Reservations'}) \\
& \quad \text{if Reservations'} \text{ is the result of appending } v \text{ to Reservations}. \\
\text{Broker}(\text{broker},\text{Rooms},\text{Reservations}) & \rightarrow \text{broker}(\text{granted}(u,v)), \\
& \quad \text{Broker}(\text{broker},\text{Rooms}',\text{Reservations'}) \\
& \quad \text{if Rooms} = [u|\text{Rooms'}], \\
& \quad \text{Reservations} = [v|\text{Reservations'}].
\end{align*}
\]

This is not far from how Airbnb operates. With a broker, one may have two separate social contracts. One among room owners and the broker (Airbnb), and one among the tourists and the broker (Airbnb again). The broker, then, is a party to both social contracts. The broker may receive a confirmation signed by the room owner via the social contract of the owners, and forward it to the tourist, via the tourists social contract. Such a signed confirmation is non-repudiated, even if obtained indirectly. Third, one may add payments, as in real life. Realizing a currency network using digital social contracts is described in a companion paper [15]. However, in our realization, the broker need not collect rent from transactions among the tourists and the room owners, as Airbnb does. It could be owned and operated cooperatively by the tourists, by the owners, or by
both. The issue of shared, autonomous agents (aka smart contracts) will be discussed in detail in a subsequent paper. Also, in this paper we have eschewed the programming language design issue of how to bind formal parameters to actual parameters in a social contract, parameter scope, and how to start a social contract. These are standard issues of programming language design, but fixing these details in this high-level paper is premature. The following muck-up code, without formal semantics yet, may illustrate a design option, assuming all names stand for public keys, for which the private keys are known by the respective agents.

Program 4: Initiating a Brokered Hosts/Tourists Social Contract

Start -->
     Broker(broker), Host(joe), Host(mary),
     Tourist(john), Tourist(sue).

3.2 A Simple Data Storage Service

A similar example of rental goods is a distributed data storage (cloud) service. Here is a simplified setting:

1. There are a bunch of users and a bunch of data storage owners; initially all storage spaces are free.

2. A user may request some storage space (to store his data) from an owner and wait; if some space is free, then the owner grants the request, possibly partially, and records the granted space as booked; if the space is booked, then the owner denies the request.

3. If the request is granted, then the user stores her data, and then checks out; when the user checks out, the owner records that the space is free. If the request is denied, then the user may request a space again, from the same owner or from another owner.

4. No money exchange here.

Program 5: Online Commerce

% Buyer can ask to buy from v
% Then, Buyer waits
Buyer(v) --> (u, v(buy)), Wait(v, u).

% If declined, cease to wait
Wait(v, u), u(decline) --> Buyer(v).

% If accepted, cease to wait
Wait(v, u), u(accept) --> Buyer(v).

% i is the number of items seller has
% Seller can add more goods
Seller(u, i) --> Seller(u, i').

% If no goods, decline
Seller(u, 0), (u, v(buy)) -->
     Seller(u, 0), (v, u(decline)).

3.3 An Online Commerce Example

Here we outline a simple Amazon-like social contract. This instance may be viewed as a market, operated by the agents, where buyers and sellers may interact. Consider the following:

1. There is a bunch of buyers and a bunch of room sellers; Each seller has some goods.

2. A buyer may request to buy a certain good from a seller; if seller has this good in his inventory, then he grants the request; Else, he denies the request.

3. If the request is granted, then the seller records the deal, updates his inventory and provides a receipt to the buyer. If the request is denied, then the buyer may request to buy the good again, from the same seller or from another seller.

4. No money exchange here.

Program 5: Online Commerce

% Buyer can ask to buy from v
% Then, Buyer waits
Buyer(v) --> (u, v(buy)), Wait(v, u).

% If declined, cease to wait
Wait(v, u), u(decline) --> Buyer(v).

% If accepted, cease to wait
Wait(v, u), u(accept) --> Buyer(v).

% Seller can add more goods
Seller(u, i) --> Seller(u, i').

% If no goods, decline
Seller(u, 0), (u, v(buy)) -->
     Seller(u, 0), (v, u(decline)).
Seller(u, i), (u, v(buy)) -->
Seller(u, i - 1), (v, u(accept)).

The contract can similarly incorporate an aggre-
gator (like Amazon), that receives order from buyers
and inventories from sellers, and matches one with
the other.

3.4 Citizens Band and Social Community Examples

Here we consider the situation that an agent wishes
to set a private social community among a subset of
the agents.

Remark 3. In the following, we place inside [square
brackets] text that hints at the privacy-related cryp-
tographic measures that we intend any implementa-
tion to employ. Such text can be skipped without
loss of continuity.

Since each agent, in the social community we wish
to program here, has its own key pair to begin with, it
is possible to set up a secret channel among any set
of agents using standard cryptographic techniques.
So here we assume that there is a community name
[which is a key-pair] that can be sent [securely] from
one agent to another, and that agents that know the
community name may communicate [privately] with
each other. The social contract focuses on the cre-
ation, management, and use of such communities.

The first community we describe is a simple broad-
cast (Citizens Band) channel. In it, every agent may
create a new channel. Every agent that knows the
name of a channel, may broadcast a message to it,
receive messages from it, and invite others to it. For
simplicity of presentation we restrict agents to have
at most one channel, and then generalize.

The states of agent $v$ are:

- $\text{Agent}(v)$: Agent $v$ is not (yet) a party to
  any channel.

- $\text{AgentCB}(v,\text{channel})$: Agent $v$ is a party to
  channel $\text{channel}$.

Here are the rules for an agent $v$ (some of these
rules depend on an incoming message $m$, while some
do not):

Program 6: Citizens Band

% create a channel for some
% new channel [key pair].
Agent(v) --> AgentCB(v,channel)

% invite u to a channel
AgentCB(v,channel) -->
(u,invite(channel)), AgentCB(v,channel)

% accept or ignore invite
Agent(v), (v,invite(channel)) -->
AgentCB(v,channel).

% send message with content
AgentCB(v, channel) -->
message(channel, v, content),
AgentCB(v, channel).

% receive message with content
AgentCB(v, channel),
message(channel, u, content) -->
AgentCB(v, channel).

Generalizing to many channels requires an agent
to maintain a list of channels, allowing an agent to
create a new channel and add it to its list, invite other
agents to any channel in its list, and send or receive
a message on any channel in its list.

We now show the social contract of a social pri-
vate community. For simplicity, we assume that the
community has one manager, its founder, that each
agent can be a member of at most one community,
and that an agent invited to become a member in a
community accepts the invitation. First we describe
community creation, joining, and use:

Program 7: Social Community

% create a new community [key pair]
Agent(v) --> Manager(v,community,[],[]) for some new community.
% join a community
Agent(v), (v,u(invite(community))) --> Member(v,community,u).

% invite u
Manager(v,community,invited,members) -->
  (u,v(invite(name))),
  Manager(v,name,[u|invited],members)
Manager(v,name,invited,members),
  (v,u(accept(name))) -->
  Manager(v,name,invited\',members')
% add u to the community
if add(u,invited, members,invited\',members').

% send a message with content
Member(v,community,u) -->
messagex(community,v,content),
Member(v,community,u).

% receive a message from w
Member(v,community,u),
messagex(community,w,content) -->
Member(v,community,u).

add(u,invited,members,invited\',members\') is a
simple list-processing routine that removes u from the
invited list to the members list, resulting in updated
lists invited\' and members\'.

Removing a member is a bit more involved. The
social contract may specify that if the community's
manager removes a member, then the member excludes itself from the community, for example as below:

Program 8: Removing a Member
Manager(v,community,invited,members) -->
  % remove u from the community
  (u,v(uninvite(name))),
  Manager(v,name,invited,members\')
  if remove(u,members,members\').

% leave the community.
Member(v,community,u),
  (v,u(uninvite(community))) -->
  Agent(v).

However, even if the member is non-faulty, it may
delay its response to the message, in the absence
of some further fairness (in the concurrency theory
sense) and responsiveness requirements on the execu-
tion of social contracts. Furthermore, even if the
agent removes itself, it may still keep, or disclose, the
community’s name [key pair], allowing itself or others
to eavesdrop on the community conversations. The
question of setting up and managing a community is
core to our approach, if we identify a community with
the parties to a contract. The question of setting up a
social contract, and of binding people’s personal iden-
tifiers to the computational agents of a social contract
is described in a companion paper [?].

3.5 Egalitarian Currency Example
Here we describe a social contract for a simple egal-
itarian currency. In this contract, there is a clock
agent, that sends clock ticks, that is, a sequence of
tick messages. The parties to the agreement start
with a zero balance of coins, and every clock tick,
each party may mint one coin. In addition, a party
can pay another party any amount not greater than
its balance.

Program 9: Single Egalitarian Currency
% the clock keeps ticking
clock --> tick, clock.

start --> Agent(x), Agent(z), clock.

% start with zero balance
Agent(v) --> Agent(v,0).

% mint one coin every clock tick
Agent(v,balance), tick -->
  Agent(v,balance+1).

% pay to another agent,
% less than the balance
Agent(v,balance) -->
  pay(v,u,x), Agent(v,balance-x)
  if balance >= x.

% receive a payment from another agent
Agent(v,balance), pay(u,v,x) --> Agent(v,balance+x).

Here is the code for a member in an egalitarian currency network. It assumes that each agent v is initialized with a list of pairs of the form (c,0) where c is the name of a currency and the initial balance is zero.

Program 10: Egalitarian Currency Network

\[
S(v,cs), \text{tick} \rightarrow \text{mint}(v,c), S(v,cs'),
\]
where (c,b) in cs, cs' obtained from cs by replacing (c,b) by (c,b+1).
% mint a c coin

Within this network, we show an implementation of an atomic swap between two currencies. In a \text{swap}(v,c_1,c_2,x) agent v burns x c_1 coins and mints the same amount of c_2 coins.

Program 11: Atomic Swap: Burning coins of one currency and minting coins of another

% burn x c_1 coins and mint x c_2 coins
S(v,cs) --> swap(v,c_1,c_2,x), S(v,cs'),
\[
\text{if balance of c_1 in cs is greater or equal to x, and cs'} \text{ is obtained from cs by subtracting x from c_1 and adding x to c_2.}
\]

4 Egalitarian Governance of Social Contracts

The examples up to this point allow agents to realize sovereignty, transparency, and privacy over a shared medium of interaction. Here, we focus on equality, and discuss several aspects of egalitarian governance of social contracts.

4.1 Egalitarian Records and Execution

This aspect of equality relates to the authorization to record and execute a social contract. In that sense, equality among a given community means that all individuals obtain the original contract and have equal authority to execute the contract. A correct implementation of the abstract model of digital social contracts must ensure this, as realized, for example, in the implementation described in the companion paper on a distributed fault-tolerant transition system that implements digital social contracts.

4.2 Equality in Voting

Here we consider the standard principle of democracy, generally abbreviated as “one person, one vote”. That is, we show how a community may form democratically. This method of democratic community formation could be applied to any of the social contracts mentioned above - the Social Community, Hosts and Tourists, and Egalitarian Currency. Here decisions to add or remove a member are taken by a simple majority, but extending the contract to require a supermajority for taking a decision is not difficult.

Note that this is a very simple program, in particular as (1) there is no option to decline an invitation (2) agents must wait for other agents to vote. Indeed, we provide the program mainly as a proof of concept, but to implement a practical democratic decision process, more complex program is needed. We also note that votes are public to the community members and are not anonymous.

Program 12: Chaired Democratic Community

% make a proposal and vote for it
Chair(v,\text{members}) -->
\[
\text{ballot(proposal,\text{members},1), Chair_wait(v,\text{members}),}
\]
\[\text{if proposal = Add(u) and u is not in members, or proposal = Remove(u) and u is in members.}\]
% invite if proposal accepted
Chair_wait(v,\text{members}),
\[
\text{ballot(Add(u),[\],result) --> invite(u,\text{members}), Chair(v,[u|\text{members}]), if result > 0.}\]
% remove if proposal accepted
Chair_wait(v,\text{members}),
\[
\text{ballot(Remove(u),[\],result) -->}\]
Chair(v,members'),
  if result > 0 and
  removing u from members gives members'.

% proposal declined
Chair_wait(v,members),
ballot(proposal, [], result) -->
  Chair(v,members),
  if result <= 0.

% accept invitation
Agent(v), invite(v, members) -->
  Member(v, members).

% vote on proposal
Member(v, members),
ballot(proposal, [v | rest], vote) -->
  ballot(proposal, rest, vote + myvote),
  Member(v, members),
  where myvote is 1 (for),
  -1 (against),
  or 0.

% proposal to add accepted
Member(v, members),
ballot(Add(u), [], result) -->
  Member(v, [u | members]),
  if result > 0.

% proposal to remove other accepted
Member(v, members),
ballot(Remove(u), [], result) -->
  Member(v, members'),
  if result > 0 and u /= v
  and removing u from members
gives members'.

% proposal to remove self accepted
Member(v, members),
ballot(Remove(v), [], result) -->
  Agent(v)
  if result > 0.

4.3 Equality in Determining the Proposals to Vote Upon

So far, only the chair can propose to add or remove a member. Now we add the ability of members to do so. If a member makes a proposal, and another member seconds it, then the proposal is put for a vote.

Typically, in a direct democracy like Switzerland, citizens can also use a petition mechanism to add a topic to the agenda, but not to remove one. So, in general, in a direct democracy citizens do not have control over the agenda. The computational foundation for the realization of this aspect of equality is proposed in our paper on vote and proposal aggregation in metric spaces [12].

Program 13: Democratic Community with Member’s Proposals

% if a proposal is seconded, the chair
% puts the proposal for a vote
% and votes on it.
Chair(v, members),
seconded(u, w, proposal) -->
  ballot(proposal, members, myvote),
  Chair_wait(v, members),
  where myvote is 1 (for),
  -1 (against),
  or 0.

% make a proposal
Member(v, members) -->
  propose(v, Add(u)),
  Member(v, members),
  if u is not in members.
Member(v, members) -->
  propose(v, Remove(u)),
  Member(v, members),
  if and u is in members.

% second a proposal if you feel like it
Member(v, members), propose(u, proposal) -->
  seconded(v, u, proposal),
  Member(v, members),
  if u /= v.
4.4 Equality in Chairmanship

With a simple addition, members can propose to replace the Chair:

Program 14: Democratically Replacing the Chair

% Chair steps down
Chair_wait(v,members),
ballot(New_Chair(u),[],result) -->
Member(v,members),
if result > 0.

% Elected Member becomes a Chair
Member(v,members),
ballot(New_Chair(v),[],result) -->
Chair(v,members),
if result > 0.

% Member proposes a new Chair
Member(v,members) -->
propose(v,New_Chair(u)),
Member(v,members),
if u is in members.

We note that there is a key issue with electing a chair as it undermines equality by granting authority to a single individual, as in a typical representative democracy. However, once agents determine the agenda, the role of the chair is purely ceremonial and it involves no discretion, and thus may potentially (and ideally) be automated. However, several forms of discretion remains, namely how to resolve ties among proposals made at the same time, as well as issues with faulty or malfunctioning chair (e.g., where the chair is asleep/dies/runs out of battery). These issues may potentially be resolved via solutions of fault-tolerant autonomous agents, discussed below.

4.5 Fault Tolerant Autonomous Agents

Implementing an autonomous community aggregator (chair, mediator, aggregator, delegate) requires a solution to a standard problem in asynchronous systems: merge or serialization. The entire Nakamoto protocol is dedicated to global fault-tolerant serialization of all actions by all agents (in all smart contracts). Since we work in a different framework, serialization is typically required only for some actions of some agents within a specific social contract. The issue, in general, is achieving fault-tolerant and fair merge. The issue is as follows. An autonomous agent such as an aggregator/chair can be realized similarly to a smart contract: A deterministic program that is run by all agents, so there is no single point of failure and all non-faulty agents can easily agree on its outcome/behavior. But, if the autonomous agent interacts with multiple agents, there is indeterminism of message arrival. The purpose of a fault-tolerant fair merge is to provide a ”black box” that is fault-tolerant and hides this indeterminism. Its specification and implementation will be discussed in a subsequent paper.

5 Outlook

We have introduced the concept of digital social contracts, provided a mathematical definition of them, outlined a design for a social contracts programming language, and demonstrated its social utility via program examples. Much remain to be done; some is discussed in companion papers [6, 15].

Acknowledgements

Ehud Shapiro is the Incumbent of The Harry Weinrebe Professorial Chair of Computer Science and Biology. We thank the generous support of the Braginsky Center for the Interface between Science and the Humanities. Nimrod Talmon was supported by the Israel Science Foundation (ISF; Grant No. 630/19).

References

[1] Lin William Cong and Zhiguo He. Blockchain disruption and smart contracts. The Review of Financial Studies, 32(5):1754–1797, 2019.
[2] Chris Dannen. Introducing ethereum and solidity.

[3] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pages 3–16, 2016.

[4] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem, 2019.

[5] Lawrence Lessig. Code is law. The Industry Standard, 18, 1999.

[6] Ehud Shapiro Luca Cardelli, Gal Shahaf and Nimrod Talmon. Fault-tolerant digital social contracts, 2020. Unpublished manuscript.

[7] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2019.

[8] Cong T. Nguyen, Dinh Thai Hoang, Diep N. Nguyen, Dusit Niyato, Huyunh Tuong Nguyen, and Eryk Dutkiewicz. Proof-of-stake consensus mechanisms for future blockchain networks: fundamentals, applications and opportunities. IEEE Access, 7:85727–85745, 2019.

[9] Ouri Poupko, Gal Shahaf, Ehud Shapiro, and Nimrod Talmon. Sybil-resistant conductance-based community growth. In Proceedings of CSR ’19, pages 359–371, 2019.

[10] Pierre-Joseph Proudhon and John Beverley Robinson. General idea of the revolution in the nineteenth century. Courier Corporation, 2004.

[11] John R Searle. Speech acts: An essay in the philosophy of language, volume 626. Cambridge university press, 1969.

[12] Gal Shahaf, Ehud Shapiro, and Nimrod Talmon. Aggregation over metric spaces: Proposing and voting in elections, budgeting, and legislation. In Proceedings of ADT ’19, 2019.

[13] Gal Shahaf, Ehud Shapiro, and Nimrod Talmon. Foundation for genuine global identities. arXiv preprint arXiv:1904.09630, 2019.

[14] Gal Shahaf, Ehud Shapiro, and Nimrod Talmon. Sybil-resistant reality-aware social choice. In Proceedings of IJCAI ’19, pages 572–579, 2019.

[15] Gal Shahaf, Ehud Shapiro, and Nimrod Talmon. Egalitarian currency networks, 2020. Unpublished manuscript.

[16] Melanie Swan. Blockchain: Blueprint for a new economy. O’Reilly Media, 2015.

[17] Shoshana Zuboff. The age of surveillance capitalism: The fight for a human future at the new frontier of power. Profile Books, 2019.