applied sciences

Article

From Agents to Blockchain: Stairway to Integration

Giovanni Ciatto [1], Stefano Mariani [2,*], Andrea Omicini [1] and Franco Zambonelli [2]

1 Dipartimento di Informatica—Scienza e Ingegneria—ALMA MATER STUDIORUM–Università di Bologna, 47522 Cesena (FC), Italy; giovanni.ciatto@unibo.it (G.C.); andrea.omicini@unibo.it (A.O.)
2 Dipartimento di Scienze e Metodi dell'Ingegneria—Università di Modena e Reggio Emilia, 42122 Reggio Emilia (RE), Italy; franco.zambonelli@unimore.it
* Correspondence: stefano.mariani@unimore.it

Received: 14 September 2020; Accepted: 19 October 2020; Published: 23 October 2020

Abstract: The blockchain concept and technology are impacting many different research and application fields; hence, many are looking at the blockchain as a chance to solve long-standing problems or gain novel benefits. In the agent community several authors are proposing their own combination of agent-oriented technology and blockchain to address both old and new challenges. In this paper we aim at clarifying which are the opportunities, the dimensions to consider, and the alternative approaches available for integrating agents and blockchain, by proposing a roadmap and illustrating the issues yet to be addressed. Then, as both validation of our roadmap and grounds for future development, we discuss the case of Tenderfone, a custom blockchain integrating concepts borrowed from agent-oriented programming.

Keywords: blockchain; smart contracts; agents; autonomy; multi-agent systems; Tenderfone

1. Introduction

The blockchain is having a growing impact on heterogeneous research and application domains, ranging from distributed storage and computing [1–4] to supply chain management and accountability in healthcare [5,6]. The promise of delivering a secure and fault-tolerant ledger that mutually untrusted parties can use to track computations and interactions in a totally distributed and decentralised way—that is, without the need to resort to a central authority—is indeed appealing for many different use cases. The agent community is no different [7], as it started showing interest in the opportunities provided by the blockchain to either solve (or mitigate, at least) long-standing issues (e.g., trust management in open systems, accountability of actions for liability, amongst others [7–10]) or exploit its expected benefits to endow a system with new properties (e.g., novel infrastructures for multi-agent systems (MAS) [11], trustworthy coordination [12]).

The central role of smart contracts (SC) in boosting such a wide impact is widely recognised, as they enable the blockchain to work as a general-purpose distributed computing engine while still preserving its main properties regarding security, trust, and fault-tolerance [13] and actually extend their reach to cover computations beyond data storage and management. In current practice, smart contracts are mostly applied in three use cases: automating procedures, verifying properties of transactions, and mediating interactions amongst untrusted distributed parties.

In this context, smart contracts are exploited to perform either pure "algorithmic" computations (e.g., automating fees or shares, enforcing access rights, etc.) or interactive computations aimed at governing when, how, and to what extent off-chain entities can communicate. Both in the distributed computing and MAS communities it is well understood how interactions (among distributed or concurrent active entities) increase the expressiveness and the effectiveness of computational systems [14]. Therefore, the integration of the agent and blockchain worlds is promising: on the one side, agents are distributed autonomous entities whose interactions should be governed and mediated in reliable ways; on the other side, blockchains and smart contracts are trust-sensitive tools for mediating and governing interactions [15].

Before the advent of smart contracts, computations in a blockchain were assigned to off-chain entities, such as dedicated software processes working as blockchain "clients", whereas with smart contracts such processes can be effectively transferred to the blockchain nodes themselves—that is, to on-chain entities. This observation could be transferred to the integration process: should agents be put "into the blockchain", perhaps as a new computational model for smart contracts, or should they stay out of it, placed side by side? Most approaches place MAS and blockchain side by side, with opportunistic interactions when the former requires services to the latter [7]. Instead, what about placing the agents on the blockchain? A principled approach to integration should at least consider these opportunities, carefully analyze them for potential benefits and limitations, then decide what to attempt to pursue in practice, at the technology level, and what not. Enumerating the possibilities and discussing their potential impact is one of the contributions of this paper.

1.1. Motivation

A principled approach to integrate agents and the blockchain is needed from both conceptual and technological perspectives.

At the conceptual level, current efforts are mostly tailored to a particular application domain or even a specific goal, and they lack a more systematic analysis of all the opportunities and aspects involved in a principled integration of blockchain and agent-oriented models and technologies—such as the on-chain vs. off-chain aspect, the relationship with smart contracts, and the computation vs. interaction dimension already discussed. Indeed, as already mentioned, no effort seems to consider the possibility of adopting agent-orientation as the computational model for smart contracts [7].

At the technological level, a careful analysis of expressiveness of current blockchain technologies (there including smart contracts) with respect to their ability of fully supporting the agent abstraction is currently lacking in the literature, apart from our preliminary attempts [15], as well as an analysis of the benefits that transferring blockchain concepts to the agent realm may bring along. As a paradigmatic example, pro-activeness and execution of time-triggered activities, which are common traits of agents defining their autonomy to take action spontaneously, are technical capabilities difficult or impossible to obtain with state-of-the-art blockchain implementations [16].

Autonomy-related issues can easily be seen when mainstream blockchains—say Ethereum (https://ethereum.org) or HyperLedger Fabric (HLF) (https://www.hyperledger.org/projects/fabric)—are exploited to support the supply chain of products, a common scenario for blockchain-based IoT applications [17]. In fact, one of the most natural applications of BCT is to track the movements of goods amongst the many actors involved in transportation from the manufacturing facility to retail stores.

Consider for instance the case of a manufacturing company willing to sell its products overseas. To make this possible, an articulated supply chain is in place involving, e.g., a shipping company, a distribution company, and a retail store. In such a scenario, the manufacturing company may leverage smart contracts to *(i)* track the movements of its products along the supply chain and *(ii)* receive automatic refunding in case some of its partner companies fail in dispatching the products on time. In particular, one smart contract is instantiated for each step of the supply chain.

Focusing on the case of the shipping company, a smart contract may be programmed to *(i)* keep track of the shipping status of the products, and *(ii)* automatically refund the manufacturing company if the products reach the distribution company later than an agreed-upon amount of time since shipment. This is exactly what is represented in Figure 1, where the interaction of a *Shipment SC* with off-chain entities is described through UML. Essentially, the diagrams represent one possible design of this scenario on top of an Ethereum or HLF smart contract. Upon shipment, an off-chain entity representing the *Manufacturer* invokes a method `ship(...)` on the *Shipment SC*, notifying the handover of the product. Accordingly, the SC registers the beginning time of the shipment. Whenever the product actually reaches the *Distributor*, the latter—which is again an off-chain entity—certifies the successful dispatch of the product by invoking the method dispatched(...) on the *Shipment SC*. By reacting to such method invocation, the SC decides if a refund is to be performed or not, depending on whether the total time required for the shipment is greater than a given threshold (ΔT) or not.

This scenario highlights, for instance, the expressiveness limitations of mainstream SC w.r.t. pro-activeness and, hence, autonomy, a definitory trait of agenthood. If the *Shipment SC* were endowed with computational autonomy, it would have been capable of performing an automatic refunding as soon as the shipment duration overcame the ΔT threshold, without requiring any external intervention. Conversely, given the current nature of mainstream SC, which are inherently reactive to off-chain entities, the *Shipment SC* can perform its refunding only after a third party, namely the *Distributor*, triggers it. This is troublesome in practice since mainstream SC have no actual capability to enforce the rules they have been instantiated for, if no external entity triggers them. On the one side, according to the recent literature about smart contracts [18], such issues can be tackled through oracles, which are, essentially, trusted off-chain parties in charge of providing fresh and timely inputs to smart contracts in order for them to realize their business logic. On the other side, we argue that the oracle-based approach is just a workaround. In fact, the aforementioned limitations reveal that smart contracts need a paradigmatic shift towards a higher degree of computational autonomy. Agent orientation may be an answer to this need, as the agent-oriented paradigm explicitly embraces the autonomy of computational entities.

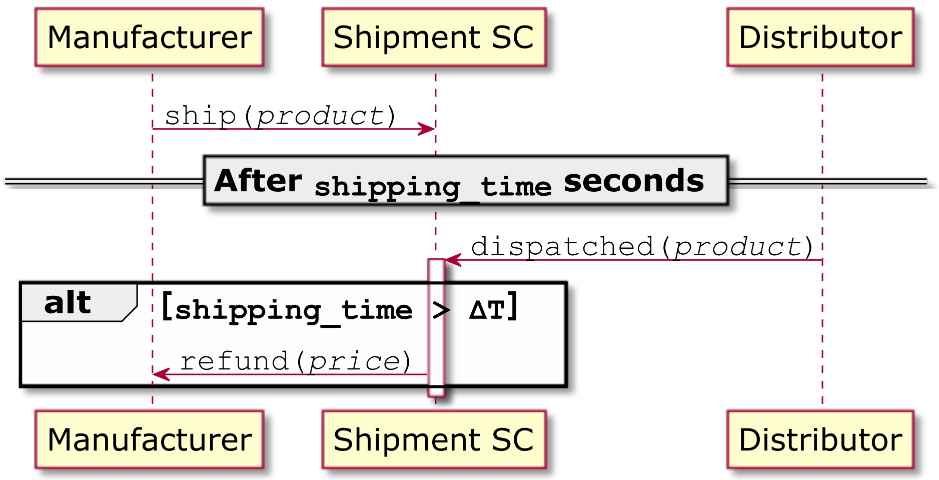

Figure 1. Schematic representation of a simple scenario where *Shipment SC* tracks the shipping status of a product produced by a *Manufacturer*, and to be dispatched by a *Distributor*. In case of a late dispatch, *Shipment SC* performs a full refunding of the shipping price to the *Manufacturer*.

1.2. Contribution

The contribution of this paper is twofold:

- We present and discuss the dimensions of agent–blockchain integration (computation vs. interaction) and the impact on agent-oriented practice and blockchain, then we enumerate and analyze viable approaches to integration by considering what we call "agent-vs-blockchain" and "agent-to-blockchain" approaches, such as letting agents interact with the blockchain vs. modelling agents as smart contracts, or injecting blockchain and smart contract concepts into agents (e.g., programming agents through smart contracts). Accordingly, we propose two roadmaps for the exploration of the aforementioned approaches
- We validate the first steps of our roadmaps through Tenderfone [16,19], which is a prototype of agent-oriented blockchain we designed and implemented, where smart contracts are empowered

with autonomy through mechanisms endowing them control flow and asynchronous interaction means, showing the advantages of pushing smart contracts closer to agent abstraction.

A preliminary version of this work appeared at the 17th International Conference on Practical Applications of Agents and Multi-Agent Systems [19]. Here, we present another roadmap covering the interaction dimension of MAS engineering, sensibly expand discussion of the context, motivations, and related works demanding this contribution, and validate a few steps along each roadmap through Tenderfone, a custom blockchain proposed in [16]. On top of this, we discuss the implementation of two case studies emphasizing the increased expressiveness with respect to state-of-the-art blockchains.

1.3. Structure

The remainder of this paper is organized as follows: Section 2 recalls the essential traits of blockchain, smart contracts, and MAS, then describes the proposed roadmaps for their integration; Section 3 presents Tenderfone, our custom blockchain infrastructure that sets a first foot on the stairway to integration, by endowing smart contracts with encapsulation of control flow and asynchronous interaction means; Section 4 discusses an exemplary scenario conceptually validating the increase in expressiveness of Tenderfone w.r.t. state-of-the-art blockchains; finally, Section 5 provides final remarks along with an outlook to promising future research efforts.

2. Stairway to Integration

In this section we first recall the basic concepts defining blockchain, smart contracts, and agency; then, we provide for a brief account of the known attempts at integrating agents and MAS with the blockchain and smart contracts. Finally, we describe the proposed roadmaps for "agent-to-blockchain" integration, one for the computational dimension and one for the interaction dimension.

2.1. Smart Contracts and Agents at a Glance

The term "smart contract" was introduced in [20] to define a "computerized transaction protocol that executes the terms of a contract", but without further formal specification. According to the authors' vision, smart contracts would replace real-world contracts by eliminating the need for a middleman, hence the cost of most transactions. They would in fact *(i)* automatically apply the terms of a contract while mediating the interactions between the involved parties, *(ii)* formalize (in a machine-readable and computable way) the implications of the contract itself, and of the admissible interactions between the parties, *(iii)* minimize the chance of malicious or accidental misconduct, and *(iv)* minimize or even eliminate the need for trusted intermediaries.

Based on this general definition, every second- or third-generation blockchain technology (simply "BCT" henceforth) proposed its own implementation of the concept. Despite their differences, it is possible to detect a common interpretation of what actually a smart contract should be implemented as: a reactive computational process executing a program on top of a blockchain. Such a reactive nature as a huge impact on the expressive reach of what can and cannot be expressed by a smart contract, and although not explicitly emphasized, is apparent as soon as one looks at technical documentation about smart contract programming languages or execution platforms (e.g., https://solidity.readthedocs.io/en/v0.5.3/introduction-to-smart-contracts.html or https://hyperledger-fabric.readthedocs.io/en/release-1.3/chaincode4ade.html). For instance, a request/response interaction protocol always initiated by an end user is almost always assumed.

A brief overview of how a generic blockchain works is necessary to fully understand also how smart contracts work. Any blockchain is essentially a peer-to-peer network of nodes storing data, adopting a consensus protocol aimed at making them work together as a single, consistent, and fault-tolerant virtual machine, and executing smart contracts (if any). Data stored on the blockchain are cryptographically secured by a combination of hashing and digital signature techniques, so as to be hard to tamper with, also thanks to consensus.

From an architectural point of view, the blockchain is a three-layer system:

1. In the bottom layer, a number of validator nodes take care of achieving consensus about ordering of transactions so as to actually create the chain of blocks constituting the shared ledger, representing the agreement about who made what happen, and when.
2. The middle layer is where the machinery executing smart contracts resides. This machinery is an interpreter/virtual machine replicated on every node (as validators), which ensures consistency amongst distributed executions. Here is where the operational semantics of a specific programming language for smart contracts is implemented.
3. The top layer is the application level, whose state and behavior are defined by the collective of smart contracts deployed on the blockchain at any given time.

The users of a blockchain interact with the top layer to affect the shared ledger, either by issuing transactions modifying data, or by deploying and invoking smart contracts. When a smart contract is deployed, replicas of its source code are propagated through consensus across the network, to be executed when due—again, through consensus. Invoking a smart contract is a transaction (TX), too, which activates the target smart contract behavior, which in turn may further trigger other smart contracts.

Given the above description, and based on the analysis of the most well-established BCT such as Ethereum [21] and HyperLedger Fabric [22], it is possible to define smart contracts more rigorously, via their peculiar properties:

user-centric —created and deployed by end users, for end users
transparent —their source code can be inspected by any participant
immutable —their source code cannot be altered after deployment
reactive —they must always be triggered from end users in the first place
stateful —each smart contract encapsulates state and behaviour
deterministic —given the same initial state and inputs, their output is always the same, consistently across nodes
synchronous —they are invoked through a synchronous request-response protocol

This more rigorous description serves the purpose of emphasizing a discrepancy between the state-of-the-art interpretation of smart contracts and what their full realisation could be, as already observed by others [23]. They lack encapsulation of control flow, which hinders the most basic form of computational autonomy, that is, pro-activeness—in the sense of making things happen, acting without the need for external stimuli. This clearly defines smart contracts as objects in the sense of software engineering, as depicted in Table 1.

Table 1. OOP vs. agent programming according to Odell [24].

Programming $\rightarrow$ $\downarrow$ Unit	**Monolitic**	**Modular**	**Object-Oriented**	**Agent**
Behavior	Non-modular	*Modular*	*Modular*	*Modular*
State	External	External	*Internal*	*Internal*
Invocation	External	External (calls)	External (messages)	*Internal (rules, goals)*

Adding encapsulation of control flow, hence computational autonomy, would qualify smart contracts as software agents [25], extending their expressive reach—as exemplified by the case studies described in Section 4. Conversely to objects, agents are not "invoked": they are asked to do something through message passing, so as to leave them free of examining incoming messages when they like to, and serving or discarding them. Such message passing is necessarily asynchronous, otherwise their proactive capabilities would be hindered.

Other distinctive features of software agents, which will be recalled in Section 2.3 when outlining a possible path to integration between agents and the blockchain, are

situated [26,27]—Agent actions are bound to the (computational) environment where they occur, there including the space-time fabric.
social —Agents may overcome their own individual limitations by interacting with others, through both explicit communication mechanisms, such as message passing, and implicit ones, such as stigmergy [28].
goal-oriented [29]—Agents act towards achievement of a goal, either embedded in their design (goal-driven) or explicitly represented and amenable of manipulation (goal-governed).

2.2. State of the Art

Most of approaches trying to bridge MAS and blockchain technologies (BCT) are of the "agent-vs-blockchain" kind—that is, they put a MAS and a blockchain side by side, letting agents exploit blockchain services upon need, in an opportunistic way [7]. We argue that these approaches do not truly integrate agents and blockchain on the technological level, as simply the MAS exploits the blockchain as it would use any other software library. In fact, there the agents are essentially treated like any other off-chain entities, such as blockchain clients, whose interaction with the blockchain is limited to issuing transactions and deploying/invoking smart contracts—as they would do with any other service.

A more interesting family of approaches is what we call "agent-to-blockchain"—that is, efforts striving to put elements of agent-oriented models and technologies directly into the blockchain. For instance, adopting agent programming languages for implementing smart contracts would be an integration effort belonging to this category, as well as empowering smart contracts' computational model (usually, object-oriented) with features defining agent-orientation (e.g., pro-activeness) [19].

Then, another family of approaches is possible, "blockchain-to-agent", which would include integration approaches trying to inject into agents concepts and techniques belonging to the BCT realm. For instance, an agent-programming platform can be proposed, where agents are programmed with a smart contract language and their interactions are enabled and regulated by the transactions mechanism of a BCT.

Amongst the mentioned approaches, some naturally see agents as off-chain entities, while others try to fit them on-chain. In agent-vs-blockchain approaches, agents are always off-chain entities exploiting blockchain services upon need. In agent-to-blockchain approaches, agents are always on-chain entities that become part of the blockchain, either as smart contracts, or nodes. In blockchain-to-agent approaches, the on-/off-chain distinction loses meaning.

Orthogonal to the aforementioned dimensions, instead, is the dimension of computation that the integration targets: pure, "algorithmic" computation, or the interaction space. Existing literature targets both dimensions. Many approaches develop along the computational dimension, as they aim at using the blockchain to store data in a secure and traceable way, so as to make agents manipulating such data accountable for their actions [3,4,6]; others target the interaction dimension, as they exploit the blockchain transaction mechanism, often together with smart contracts, to validate and regulate interactions amongst agents [30]. Nevertheless, we emphasize that the whole spectrum of the interaction dimension is not fully considered. In existing approaches, interaction is always intended as occurring between off-chain entities (be them agents or not), whereas the possibility of using agent-oriented abstractions to mediate on-chain entity interactions (such as smart contract reciprocal invocations or node communication protocols) is left unexplored.

In next section we focus on "agent-to-blockchain" approaches, while leaving "blockchain-to-agents" as a future work.

2.3. The Roadmaps

The first roadmap depicts the steps involved in bringing the agent abstraction to the blockchain along the computational dimension—in particular, taking smart contracts as the reference abstraction to receive agent-oriented features (Figure 2).

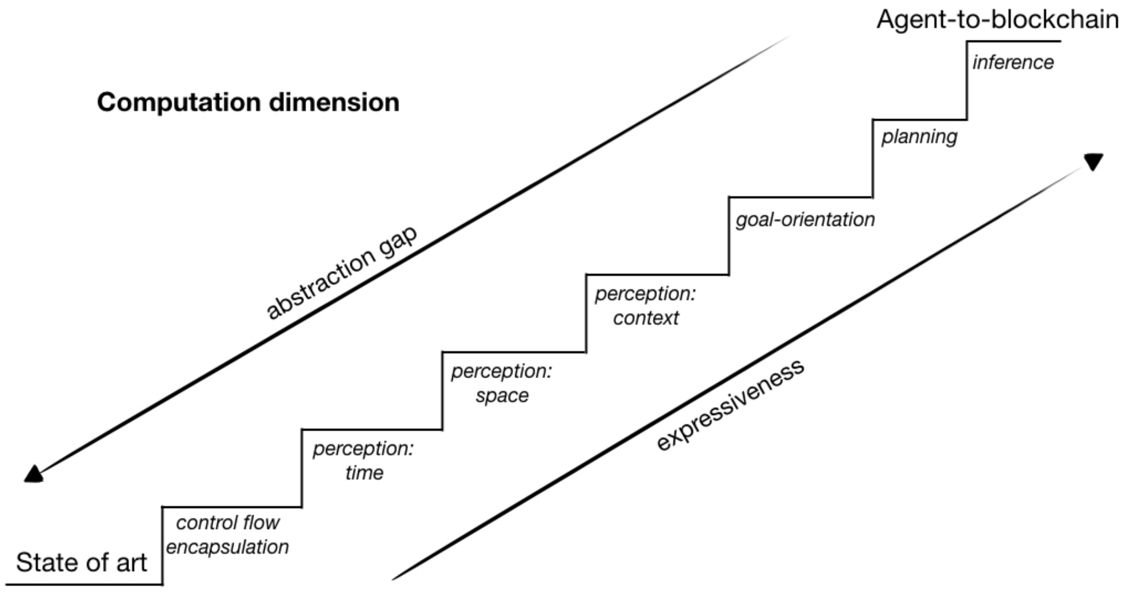

Figure 2. Roadmap to exploration of agent-to-blockchain integration focused on the computational dimension.

As discussed above, the first step is about endowing smart contracts with their own flow of control, so as to enable the pro-activeness that differentiates agents from, for instance, objects in object-oriented programming. In practice, encapsulation of control flow enables smart contracts to start computing in the absence of stimuli coming from off-chain entities—namely, without the need of some blockchain client to invoke them (it is worth clarifying here that even in the case of a smart contract being invoked by another one, in a sort of "call chain", the start of the chain always originates from an off-chain entity). Steps 2 to 4 are all about perception of events, hence reactiveness to events, and could have been put in any order. They stem from the characteristic of agents of being situated in an environment (either physical or computational) [31], which includes the space-time fabric as well as the context of action in terms of resources and properties affected. Step 5 deals with another peculiar trait of agency, often recognised as the most important one: goal-orientedness, which is the ability of directing an inner decision making process towards achievement of implicit (goal-driven) or explicit (goal-governed) goals [32]. Goals are the driving force that directs an agent's pro-activeness, as agents are expected to actively pursue the goals ascribed to them without the need of being continuously stimulated—as happens instead in the case of object-orientation with service/method/function invocation, where services/objects are purely reactive entities that provide functionalities. As an illustrative example, having goal-driven smart contracts would amount to having smart contracts capable of actively monitoring the blockchain to pro-actively take action as soon as some specific condition or event is detected. Step 6 directly follows: once agents are endowed with goals, they may be expected to carefully plan their course of actions towards achievement of such goals. In the blockchain realm, this would amount at letting smart contracts figure out by themselves what actions are needed to achieve some desired effect, such as validation of the state of the blockchain after a set of transactions, or ordering of off-chain entities interactions to avoid conflicts. The last step in the stairway requires inference, intended as the ability to autonomously synthesize novel information, plans, or even goals, depending on contextual information and learnt experience. Such an ability resembles human intelligence; hence, we regard it as the latter steps towards a full notion of agency. Transferring inference to the domain of blockchain and smart contracts requires a bit of imagination; nevertheless, we may envision smart contracts that upon inspection of the history of transactions recorded in the blockchain are able of summarizing such information to save storage space, or smart contracts that dynamically restrict the policy for transactions validation when they detect a surge in abnormal requests.

The expected practical benefits of actually walking the stairway are manifold: parallelism in on-chain processing (step 1); time-aware policies such as scheduling of periodic payments (step 2); reactiveness to off-chain events without resorting to oracles (step 4); ability to adapt actions and processes to current state of the blockchain (step 5). However, many open issues should also be addressed both conceptually and technically. For instance, inference may make the overall blockchain behavior less predictable and smart contracts non-deterministic, and perception of space may be meaningless for an infrastructure where every process is replicated across the whole network. Before designing a specific integration process for MAS and BCT, all of the aforementioned opportunities (steps) should be considered, along with expected benefits and potential issues. In Section 3, we set a first foot on the stairway, by endowing smart contracts of our custom Tenderfone BCT with both encapsulation of their own control flow, and reactiveness to time.

In order to complete discussion of agent-to-blockchain approaches, the second roadmap we discuss (Figure 3) is about the interaction dimension, still taking smart contracts as a reference.

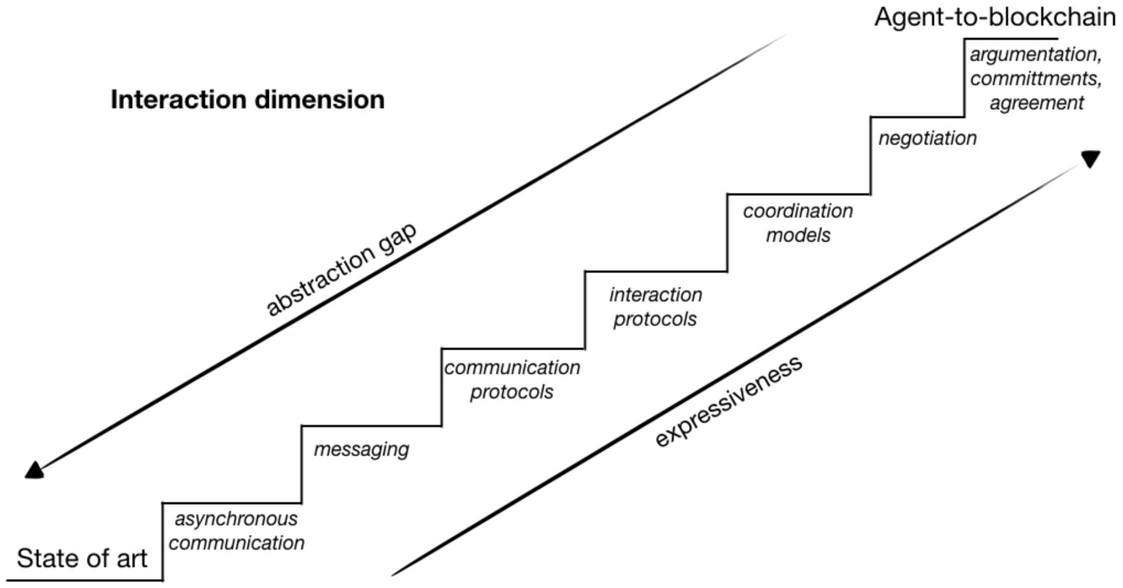

Figure 3. Roadmap to exploration of agent-to-blockchain integration focused on the interaction dimension.

The first step here is changing the communication means to asynchronous communication instead of synchronous method call/service request. In fact, one staple trait of agents is that they cannot be invoked or commanded, but may be only asked to perform an activity, letting them decide whether to comply or not. For smart contracts, this means shifting from a purely object-oriented style of interaction, where smart contracts demand (call) each other functionalities (methods) waiting for a reply (return of control flow), to a style akin to promise/futures/callbacks mechanisms. It is worth emphasizing that this first step along the interaction dimension is required by the first step on the computational dimension; without asynchronous communication, the flow of control of each smart contract would be interrupted by a synchronous method call. This is unacceptable under the agent abstraction, as it would hinder its quintessential feature: autonomy. The second step fosters messaging amongst smart contracts, likewise agents exchange messages to communicate information and request action. It is worth emphasizing that the difference with simple asynchronous communication is that messages may be stored, filtered, and later retrieved (from a mailbox) for processing, whereas asynchronous function calls are merely executed as soon as possible. Steps 3, 4, and 5 naturally follow: messaging enables agents to structure their communications according to predefined protocols, such as FIPA ones [33], whose state and admissible messages for each state can be tracked and checked against the protocol specification to detect deviations; then (step 4), it has to be recognized that not only communicative acts can be regulated by protocols, but more generally interactions, there including dependencies amongst activities that the blockchain may regulate even in the absence of specific communicative acts by participants; finally (step 5), full-fledged coordination models may be put to work to enable and govern interactions, with the aim of endowing further properties in the system, such as liveness and safety. In the realm of smart contracts, this may amount at endowing smart contracts with rich interaction protocols dictating when each smart contract has to take action in a given workflow so as to guarantee convergence to a desired functionality. Finally, step 6 goes beyond fixed coordination models and let agents exploit their goal-orientedness, planning, and inference capabilities to negotiate their joint activities so as to achieve their own individual or shared goals, while step 7 represents for us the most socially rich behavior agents can exhibit: the ability to argue about the state of affairs and joint actions, by reasoning about the motivations and debating on the shared course of action to carry out in terms of commitments, obligations, and expectations, so as to reach an agreement about who does what, when. The latter step is perhaps difficult to imagine applied to blockchain and smart contracts: it would amount at letting smart contracts figure out by themselves the best way to interact with each other so as to realize their functionalities and pursue their goals.

In this case, we also expect several practical benefits: asynchronous interaction would let smart contracts invoke each other for performing long-running computations without blocking the whole control flow, messaging would enable storage, filtering, and later processing of incoming requests in the form of messages queued in a mailbox, and full-fledged coordination models put to work for governing interactions amongst smart contracts may enable easy setup and enforcement of arbitrary workflows. However, as in the case of the roadmap along the computational dimension, the expected benefits bring along open issues to consider as well: asynchronous interaction may hinder guarantees of termination and make smart contracts wait indefinitely for replies to requests, whereas message-based communication leaves to smart contracts the freedom to decide when to process incoming messages, whose replies may thus get delayed.

2.4. Discussion

We are aware that both the roadmaps we propose may be incomplete, or they may be arranged in a different order, yet we aim at emphasizing the ultimate goal of such an effort. Integration of blockchain and smart contracts technology with agent technology is a promising research direction that has already shown meaningful results, but for a well-founded assessment of the opportunities and issues of such an integration, a conceptually sound systematic exploration of the available alternatives has to be made. We are also aware that some of the steps we try to motivate may actually not have a clear practical development, and some steps that appear clear in one domain (agent-oriented literature) may have no clear mapping in the other (blockchain technology).

Nevertheless, we argue that the vision of endowing blockchain with agent-like smart contracts, hence with the full spectrum of features defining the very notion of agency, is an intriguing scenario worth exploring in a more systematic way than what is currently happening in the literature, where ad-hoc and goal-specific designs and approaches reign supreme.

The next section describes our proposal aimed at taking the first steps along both stairways: we endow smart contracts implemented in a custom blockchain with their own flow of control, reactiveness to time, and asynchronous communication means.

3. Towards Autonomous SC with Tenderfone

Generally speaking, BCT exposing some notion of smart contract can be described as layered distributed systems, whose main layers are depicted in Figure 4: *(i)* a ledger-based state machine replication (SMR) layer ($\mathbf{L}_1$), letting a number of distributed processes share a common state and update it in a consistent way, via consensus protocols, and *(ii)* a replicated interpretation layer for SC ($\mathbf{L}_2$), built on top of the SMR layer, dictating smart contract semantics and exposing two main interfaces to either SC developers or end users.

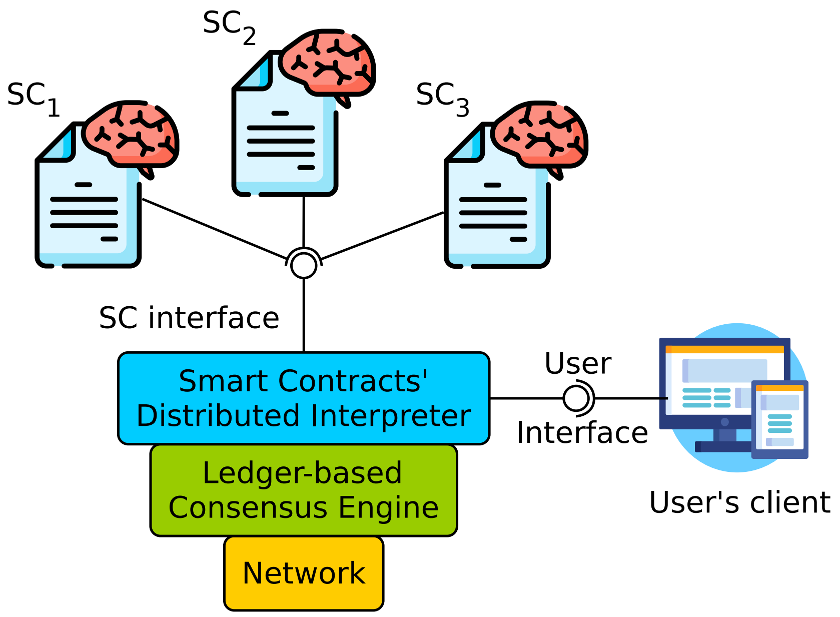
SC1
SC2
SC3
SC interface
Smart Contracts' Distributed Interpreter
User Interface
Ledger-based Consensus Engine
Network
User's client

Figure 4. General layered architecture of smart contract (SC)-enabled blockchain technology (BCT).

The end-user interface ($\mathbf{I}_1$) is aimed at letting users deploy SC or send messages to them, by issuing transactions. Conversely, the SC interface ($\mathbf{I}_2$) aims at providing SC developers with built-in functionalities to write SC. These may include, for instance, message passing (or money exchange) primitives, common computational and storage constructs, and some means to read relevant information from the underlying ledger (e.g., time).

With the twofold aim of validating our envisioned roadmaps and providing a common ground for researchers to experiment with and build systems integrating the foundational concepts of agency into the blockchain, in this section we describe a novel design for $\mathbf{L}_2$, $\mathbf{I}_1$, and $\mathbf{I}_2$, aimed at supporting autonomous, asynchronous, time-aware, and logic-based smart contracts. In particular, we discuss the case of Tenderfone (source code publicly available at https://gitlab.com/pika-lab/blockchain/tenderfone/tenderfone-sc), an experimental platform we built to advance the state-of-the-art blockchains with autonomous, asynchronous, time-aware, and logic-based smart contracts. First, we describe the language for programming SC and manipulating them (i.e., $\mathbf{I}_1$ and $\mathbf{I}_2$), then we summarize the functioning of the Tenderfone Interpreter (i.e., $\mathbf{L}_2$), emphasizing the assumptions required for the underlying BCT to make SC autonomous, asynchronous, time-aware, and logic-based.

As a last note, it is worth to highlight that, while currently leveraging on Tendermint [34] as the underlying ledger-based SMR technology, the Tenderfone Interpreter specification is both technology and consensus protocol agnostic as it relies on very loose assumptions about the underlying SMR layer. Thus, even if we chose Tendermint as our implementation platform, any other BCT satisfying such assumptions could be used as well.

3.1. An Agent-Oriented Language for SC

Tenderfone comes with a declarative language based on first-order logic (FOL). In particular, it adopts the Prolog syntax; hence, its interpreter is a Prolog engine—tuProlog [35,36] is the chosen one. Despite the choice of a declarative and logic language for smart contract programming is not new [37], as it allows the design and implementation of SC at a higher-abstraction level, in Tenderfone we leverage logic programming as a means to endow smart contract with goal-oriented reasoning. Given this choice, a Tenderfone SC is actually a logic theory against which goals, expressed as Prolog terms, can be proven. Or, in other terms, it is a knowledge base describing the application state and enabling to check properties of such states (e.g., to verify that transactions respect some constraints). Such a KB is actually split in two parts: *(i)* a static part stores the the logic program to be executed and is immutable (it is what gets deployed to the blockchain with a deployment transaction), while *(ii)* a dynamic part tracks the mutable state of the smart contract.

Let us provide an overview of how SC code is structured and expected to work in Tenderfone. A Tenderfone SC program consists of many callbacks (i.e., Prolog rules) to be implemented by smart contracts developers. Within the bodies of such callbacks, developers can leverage on a number of built-in predicates ready to use. Callbacks come either in the form "`init(+Args) : -Body`" (we rely on Prolog standard notation for input/output arguments: `+` is an input, `-` is an output, `?` can be both, whereas `@` is a ground input) or in the form "`receive(+Msg, @Sender) :- Body`". There, the `init` callback is triggered (just once) right after a deployment transaction is published, and it is aimed at endowing the SC with its own flow of control, whereas the `receive` callback is executed whenever *(i)* a user publishes an invocation transaction, or *(ii)* the SC is the recipient of a message from a SC (there including itself), as depicted in Table 2. It is worth highlighting that messages sent through the `send(+Msg,+Recipient)` built-in are characterized by asynchronous semantics. This implies the `receive` callback is eventually triggered on the recipient SC if, and only if, execution of the sender's entry point (where `send` is called) terminates successfully. This implies that the recipient callback is always executed after the sender's one—even if sender and receiver coincide.

Within callback bodies, Tenderfone SC can autonomously affect its own control flow in several ways, thanks to some built-in predicates that have been conceived to serve this specific purpose. For instance, the Tenderfone language endows smart contracts with the "`deploy(+Interval, +Message, -GeneratedID)`" built-in predicate, which allows the calling SC to postpone a computation of some relative `Interval` of time, by scheduling the execution of some `receive` callback for a later moment in time. Similarly, the "`periodic(+Period, +Message)`" built-in predicate provokes the periodic execution of some `receive` callback on the calling SC, thus allowing smart contracts to perform cyclic, time-related activities.

Finally, off-chain users can interact with Tenderfone smart contracts by means of an API composed by three primitives, namely "`deploy(+Theory, +Args)`", "`invoke(+Message, +Receiver)`", or "`observe(+Query, +Receiver, -Answer)`", aimed at instantiating, triggering, or inspecting a smart contract, respectively. In particular, the `deploy` API primitive provokes a deployment transaction to be registered on the blockchain, whereas the `invoke` one provokes the registration of an invocation transaction. Conversely, as the `observe` primitive is aimed at letting users query the inspectable status of a particular SC in a read-only fashion, it does not provoke any transaction to be registered on the blockchain.

The many Tenderfone features described in this overview are summarized in Table 2. In the following paragraphs, we provide a more detailed description of the intended semantics of each construct.

Table 2. Tenderfone language at a glance (*message* is a new kind of transaction defined in Tenderfone).

Callback	Executor	Trigger	Carrier	Initiator
`init(A)`	a SC whose *KB* = `T`	API `deploy(T, A)`	deployment TX	a user
`receive(M, S)`	a SC `R`	API `invoke(M, R)`	invocation TX	a user `S`
		built-in `send(M, R)`	*message*	a SC `S`
`receive(M, R)`	SC `R`	built-in `when(T, M)`	*message*	a SC `R`
		built-in `delay(D, M)`	*message*	a SC `R`
		built-in `periodically(P, M)`	*message*	a SC `R`
`answer(Q, S, X)`	a SC `R`	API `observe(Q, R, X)`	*query*	a user `S`

3.1.1. Computational Autonomy

Computational autonomy, in the most basic acceptation of "encapsulating the flow of control", is actually enabled by the interplay between a few distinct mechanisms, among which the Tenderfone Interpreter plays a crucial role, as clarified in Section 3.2. Nevertheless, a first autonomy enabler is the `init(+Args)` callback, which the developer defines to specify what the smart contract should do as its very first action. In other words, such callback dictates what a SC should do right after its deployment transaction completes successfully.

Within the body of the `init/1` callback (for the sake of brevity, we also rely on the standard Prolog notation when referring to rules: `rule_name/N` indicates a rule named `rule_name` which accepts `N` arguments), the developer may decide to take advantage of time-awareness and, possibly, asynchronous message passing to specify a pro-active behavior in which *(i)* the smart contract periodically executes tasks, *(ii)* there including reception of messages and *(iii)* deliberation about whether and when to actually process them. This is technically enabled by a clever combination of the `receive/2` callback and the time-aware built-in predicate, both described in the following.

The semantics enforced by the Tenderfone Interpreter guarantees that deployment of a valid smart contract can only succeed if goal `init/1` can be proven using arguments `Args` provided by the transaction issued by end user U. In case of success, the new smart contract SC is actually deployed, and its KB is stored on the blockchain. In case of failure, no smart contract is deployed and an error description is returned to the deploying user U.

3.1.2. Asynchronous Interaction

Tenderfone smart contracts interact through asynchronous message passing, as follows.

- `receive(+Message, @SenderID)` is a callback that specifies what a smart contract should do every time it is triggered by an invocation transaction carrying a `Message`, and coming from either an end user (through `invoke/2`) or another smart contract (through `send/2`) identified by `SenderID`. The execution of each `receive/2` callback, on any smart contract, is atomic and must terminate like in any other SC-enabled blockchain technology. However, differently from other BCT, smart contracts in Tenderfone can trigger their own `receive/2` callbacks, autonomously, by means of some mechanisms described below. For instance, a smart contract may simply decide to store the message but process it later on, through the time-aware constructs described in next subsection, so as to preserve its autonomy without hindering the blockchain semantics.
- `send(+Message, @ReceiverID)` is a built-in predicate aimed at letting smart contracts send messages to each other. It works by simply adding `Message` to the outbox of the current smart contract, in order to let it be eventually dispatched to `ReceiverID`. However, the actual dispatching of `Message` does not occur immediately. Accordingly, we say that the `send/2` built-in predicate just "outboxes" a `Message`. As any other built-in predicate, the `send/2` one can only be used within the body of some callback. According to Tenderfone asynchronous semantics, the actual dispatching of the (possibly many) messages outboxed by the (possibly multiple) invocations

of the `send/2` predicate within a given callback is conditioned by the successful execution of such callback. If (and only if) the execution of the wrapping callback terminates successfully, then the messages in the outbox are actually dispatched to their recipients. This implies that the `receive/2` callback is then triggered on the receiver smart contracts (i.e., the ones identified by `ReceiverID`) by means of an invocation transaction, which is generated by the Tenderfone Interpreter whenever some callback successfully terminates.

- `answer(+Message, @SenderID, -Result)` is a callback that specifies how the smart contract should answer (with which `Result`) every time it is triggered through `observe/3` API primitive (last paragraph) by an end user whose identifier is `SenderID`. Any side effect possibly produced by the callback during its execution is simply dropped as soon as the callback terminates. In other words, the `answer/3` callback is aimed at serving users' queries in a read-only fashion. In fact, this callback is intended to promote encapsulation: by defining many `answer/3`, the developer chooses which parts of the SC status to make observable.

It is worth to be noted that any invocation transaction issued by entity E towards SC to transmit a message M is valid only if the goal "`receive(Message, SenderID)`"—subject to `Message` $= M$ and `SenderID` $= id(E)$—can be proven against the current KB of SC. In case of success, the current state of SC's dynamic KB is committed to the blockchain and, therefore, updated. In case of failure, any change to SC's dynamic KB, and possibly any outgoing message queued by `send/2`, is dropped.

3.1.3. Reactiveness to Time

Tenderfone assumes that the underlying BCT provides a global notion of (logical) time, T_g, maintained through consensus, hence consistently observable every time a new block is created—a hypothesis which is likely to hold given that virtually all practical consensus algorithms leverage on some shared notion of time-round to circumvent the FLP impossibility theorem [38]. The following built-in predicates are then available to Tenderfone developers, to let smart contracts observe time (time-awareness) and act depending on it (reactiveness to time).

- `now(-Timestamp)` is a built-in predicate aimed at letting SC inspect the current time. It works by retrieving the last value of T_g shared through consensus, that is, the value of T_g stored in the block whose commit has triggered execution of the current smart contract. Hence, multiple invocations of this predicate occurring in the same callback will always get the same value for `Timestamp`.
- `when(@Instant, +Goal)` is a built-in predicate aimed at letting SC postpone a computation for a future moment in time. It works by scheduling the execution of the `receive(Goal, Me)` callback, assuming `Me` is the executing smart contract ID, as soon as $T_g \geq$ `Instant`, as a way to delay a computation to an absolute point in time.
- `delay(@Interval, +Goal)` is a built-in predicate aimed at letting SC delay a computation for a relative amount of time. It works by scheduling the execution of the `receive(Goal, Me)` callback as soon as $T_g \geq T_g' +$ `Interval`, where T_g' is the value of the global time when `delay/2` was invoked.
- `periodically(@Period, +Goal)` is a built-in predicate aimed at letting SC execute a computation periodically. It works by *(i)* triggering the `receive(Goal, Me)` callback other than *(ii)* scheduling the future execution of the same callback for the first possible instant $T_g \geq t_0 +$ `Period`, where t_0 is the value of the global time when `receive/2` was first invoked. Such process is repeated again as soon as the `receive/2` callback is executed twice. The repetition of this process over and over again produces a periodic series of executions of the `receive/2` callback, which is unlimited in length. The temporal distance among any two consecutive executions of the `receive(Goal, Me)` is constant and is equal to `Period` units of time. The series is interrupted as soon as some execution of `receive(Goal, Me)` fails.

Notice that in the description above, we assumed the variables `Timestamp` and `Instant` represent an instant in time, such as `datetime(Y, M, D, HH, MM, SS)`, whereas variables `Interval`, Delay, and Period represent an interval of time, such as seconds(SS), ..., years(Y) or any combination thereof.

3.1.4. Miscellaneous

The following built-in constructs are also provided as general-purpose facilities.

- self(-Identifier) retrieves the Identifier of the current smart contract—which is automatically generated by the Tenderfone Interpreter and immutably stored into each SC's static KB upon creation.
- owner(-Identifier) retrieves the Identifier of the owner of the current smart contract—that is, by construction, the user creating it.
- set_data(@Key, ?Value) either stores the (Key, Value) pair in the dynamic KB of the current smart contract or replaces (Key, Whatever), if present.
- get_data(?Key, ?Value) checks if a pair matching (Key, Value) exists within the current smart contract's dynamic KB, and, if it does, lets the SC read it.

3.1.5. End User API

Users can interact with Tenderfone smart contracts pretty much as they do with state-of-the-art smart contracts in mainstream BCT. In particular, Tenderfone exposes three API primitives aimed at letting off-chain users manipulate smart contracts.

- deploy(@StaticKB, +Args, -GeneratedID) creates a smart contract whose static KB is set to StaticKB and triggers its callback init/1 with arguments Args. If execution of the callback is successful, the smart contract is actually deployed, and a fresh and unique identifier is automatically assigned to it by the Tenderfone Interpreter. Furthermore, the generated identifier is bound to variable GeneratedID and thus returned to the user.
- invoke(+Goal, @SmartContractID) triggers the receive(Goal, UserID) callback, where UserID is the identifier of the calling user, on the smart contract whose identifier is SmartContractID. Execution fails if receive(Goal, UserID) fails, and in that case, all side effects occurred are discarded; hence, invoke/2 has all-or-nothing semantics.
- observe(+Query, @SmartContractID, -Result) triggers the answer(Query, User, Result) callback, where variable User is bound to the identifier of the invoking user, on the smart contract whose identifier is SmartContractID. Execution fails if and only if the triggered answer/2 callback fails. In any case, all side effects possibly occurred are dropped, as observe/3 is intended with a read-only semantics to let users inspect the state of the blockchain and of SC

3.2. The Tenderfone Interpreter

The API described so far requires the underlying interpreter to support asynchronous communication and computation for smart contracts. In particular, such an interpreter should let SC react to both user- and SC-generated messages. This is usually not the case in mainstream BCT implementations, where SC can only react to user-generated messages carried by transactions, as transactions can only be issued by off-chain users.

To overcome the aforementioned limitations, we designed the Tenderfone interpreter, which essentially lets SC issue transactions autonomously, and also route messages directed towards other SC with the purpose of triggering them. To do so, Tenderfone relies on a modular design which is inspired to the Tendermint SMR technology.

Accordingly, as depicted in Figure 5, Tenderfone assumes the underlying BCT system to be composed by several distributed validator processes, aimed at enacting a consensus protocol to keep track of the many transactions issued by the end-users. Periodically, validators agree on the ordering of the last N transactions. Then, they pack these transactions into a time-stamped block and forward it to a local interpreter process, which is in charge of executing transactions according to the Tenderfone semantics described in Section 3.1.

Validators and interpreters are assumed to exist in equal quantities, and to be deployed on different nodes of the network, in such a way that each validator–interpreter couple runs on a different node. To support the Tenderfone semantics, whenever a new block of transactions is ready, validators and interpreters enact the request–response protocol, defined in [16] and depicted in Figure 6, which aims at executing the transactions therein contained orderly, thus letting smart contracts execution progress. As a by-product of this protocol, a number of transactions may be spontaneously generated by interpreters, for instance to schedule time-reactive computations or to support asynchronous message passing. We call these transactions spontaneous, and they are the core mechanism enabling smart contract's autonomy in Tenderfone.

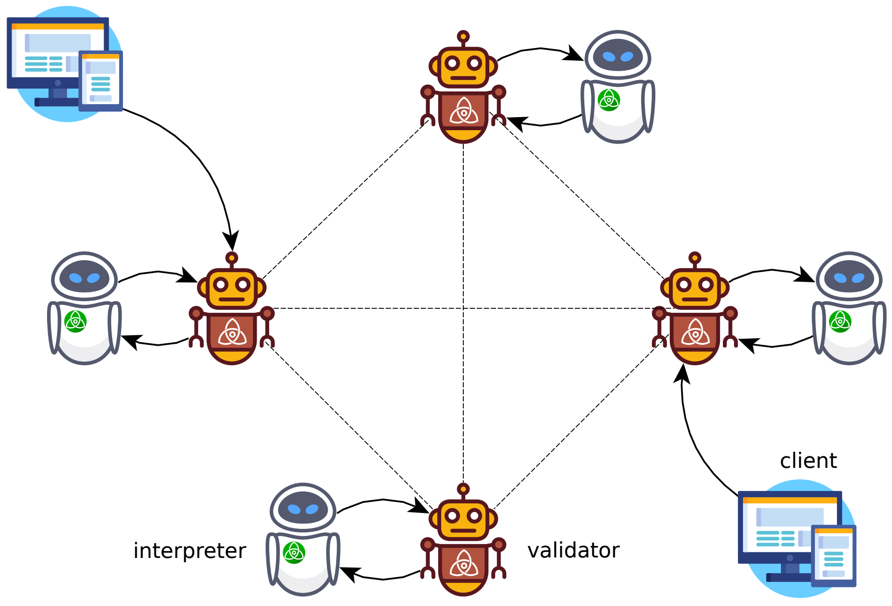

Figure 5. Representation of a Tenderfone system at run-time: a number of validator–interpreter couples are deployed on different nodes of the network and kept consistent via the Tendermint SMR engine. Off-chain entities (e.g., clients) and interpreters may issue transactions, which are ordered validators and executed by interpreters, in a replicated way.

More precisely, the Tenderfone block execution protocol works as follows:

1. via consensus, validators agree on the content of the i^{th} (i.e., transactions $t_{i,j} : j \in \{1, \ldots, N\}$) block and on its time-stamp τ;
2. each validator then notifies the beginning of the protocol to its local interpreter, informing it that the management of block i begins at time τ;
3. the local interpreter joins the protocol by returning a list of spontaneous transactions, one for each computation which has been previously postponed, delayed, or scheduled to an instant not later than τ;
4. then, the protocol makes each validator forward each transaction $t_{i,j}$ to the local interpreter, which in turn
    - executes $t_{i,j}$, possibly updating the current state of the system (e.g., by creating, destroying, or altering the internal state of some smart contract),
    - appends all the outgoing messages possibly sent by any smart contract triggered during the previous step into a temporary outbox queue;

5. once all transactions have been processed, validators notify the commitment of the block to their local interpreter;
6. each interpreter should then conclude the protocol by returning another list of spontaneous transactions, one for each outgoing message accumulated into the aforementioned outbox queue—which is finally emptied.

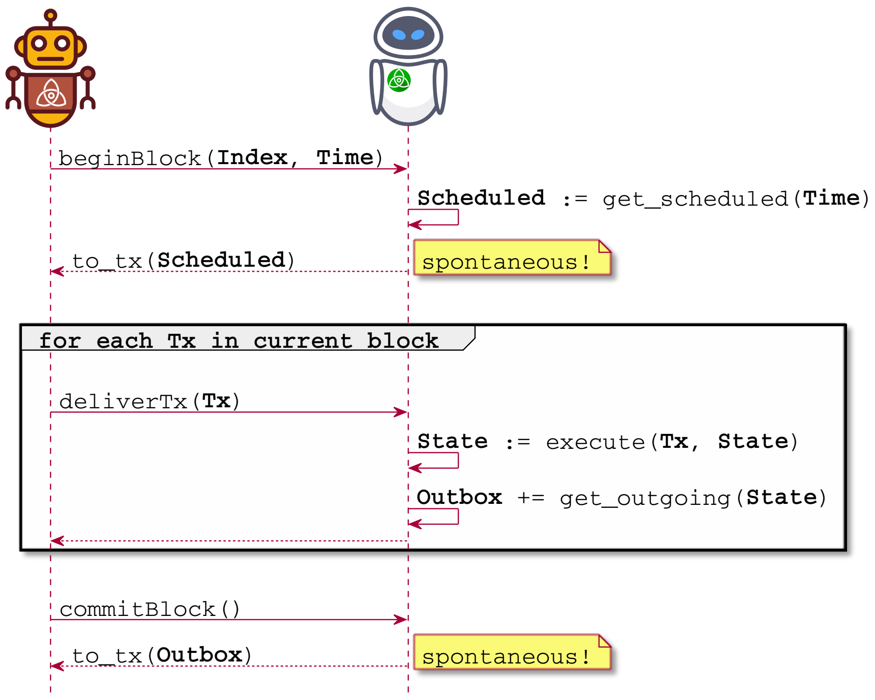
beginBlock(Index, Time)
Scheduled := get_scheduled(Time)
to_tx(Scheduled)
spontaneous!
for each Tx in current block
deliverTx(Tx)
State := execute(Tx, State)
Outbox += get_outgoing(State)
commitBlock()
to_tx(Outbox)
spontaneous!

Figure 6. Request–response protocol enacted by each validator–interpreter couple in Tenderfone to execute the transactions contained in a block. Notice that spontaneous transactions are generated by interpreters and returned to validators.

Spontaneous transactions generated during the block execution process are accumulated by validators as if they where ordinary transactions issued by end users. The only difference among spontaneous and ordinary transactions is that the former ones come in multiple copies. In fact, letting V be the number of validators in a Tenderfone system, any legitimate event provoking a spontaneous transaction (e.g., an outgoing message, as well as a postponed, delayed, or periodic computation) would be handled V times, by as many interpreters, provoking the generation of V copies of each spontaneous transaction. Despite being unavoidable, this is far from being an issue since, as discussed in Section 3.3, it lets the system distinguish among legitimate and forged events. However, this technicality leads to the need of executing just one copy out of V for each spontaneous transaction in order to adhere to the intended semantics of Tenderfone smart contracts. This is exactly what Tenderfone interpreters do, by only executing the V^{th} (i.e., the last) copy of each spontaneous transaction and ignoring the other $V - 1$ ones.

Summarizing, the Tenderfone Interpreter provides a notion of computationally autonomous smart contract by supporting both *(i)* the postponing, delaying, or periodic repetition of asynchronous computations, and *(ii)* asynchronous message passing via spontaneous transactions which are automatically generated during smart contracts execution.

3.3. Autonomy vs. Security

Trust issues may arise since byzantine interpreters or validators may technically forge spurious transactions out of thin air, with the purpose of triggering a smart contract in an illegitimate way. This is always possible, as spontaneous transactions are generated by interpreters after consensus has been achieved on a block, and nothing prevents interpreters from cheating after that. More importantly, this is a potential violation of smart contract computational autonomy, other than an undesirable security threat—for instance, vulnerable to DOS attacks.

To discriminate among legitimate and spurious, possibly forged, spontaneous transactions, Tenderfone exploits the existence of multiple copies for any legitimate spontaneous transaction. In particular, Tenderfone interpreters not misbehaving guarantee that only spontaneous transactions for which V copies have been observed are actually executed, and their effects persist on the blockchain, otherwise they are just registered on the blockchain unexecuted. In other words, from a smart contract perspective, its outgoing messages are actually considered "sent" only when the V^{th} copy of the corresponding transaction has been committed on the blockchain by the consensus algorithm. Faulty or malicious interpreters may of course still forge transactions, but unless other interpreters are faulty or malicious too, they will be simply ignored.

In practice, spontaneous transactions are cryptographically signed in Tenderfone by the interpreter generating them, even if their sender's identifier is a SC one. This is how spontaneous transactions are distinguished from ordinary ones. To prevent the forgery of spurious transactions, Tenderfone actually waits for V copies to appear on the blockchain, and requires such copies to be signed by all V interpreters.

As a last note, it is worth to be mentioned that Tenderfone is not just a theoretical model. Despite being a proof of concept, Tenderfone's ambition is to become a full-fledged smart contracts-enabled blockchain technology. To this end, it includes a number of practical features that are not discussed in this paper since they are not strictly concerned with autonomy. For instance, concerning security, the current implementation comes with a Role-Based Access Control (RBAC) sub-system based on public-key cryptography, which makes every entity composing the system (including end users) validators, and smart contracts to be individually accountable for their actions, thus easing the task of recognizing legitimate spontaneous transactions. Furthermore, concerning the termination of callbacks execution, Tenderfone leverages strict temporal bounds to prevent attackers to perform DoS attacks by deploying and triggering infinite recursion within some smart contract KB.

4. Case Studies: Supply Chain and Public Tender

In this section we shortly describe two example use cases for Tenderfone smart contracts, which are out of reach for the state-of-the-art systems. In the first example, we show how Tenderfone addresses the supply chain scenario described in Section 1.1. The second example shows how Tenderfone may enhance public tenders through automation. In both cases, there is some temporal-related aspects in which mainstream BCT would be tackled through trusted, off-chain third parties such as oracles. Conversely, here we show how such aspects can be naturally modelled and implemented in Tenderfone.

4.1. Supply Chains

As discussed in Section 1.1, the supply chain scenario is of paramount importance within the scope of BCT-based IoT. In the described scenario, the off-chain entity in charge of acknowledging the dispatch of the products, namely the distributor company, represents a single point of trust. In fact, in the worst case it may arbitrarily delay its acknowledgement, thus preventing the manufacturing company from being refunded. As a countermeasure, the Tenderfone SC shown in Listing 1 can be exploited. It leverages on Tenderfone time-related features to make the smart contract autonomously refund the manufacturing company as soon as the shipment duration exceeds a given threshold. This removes the need for an off-chain entity in charge of triggering the refunding in case of a late dispatch. In other words, the SC from Listing 1 can be modelled as the timed finite state automaton depicted in Figure 7 where the transition into the "Late shipment" state can be provoked by the flow of time, automatically.

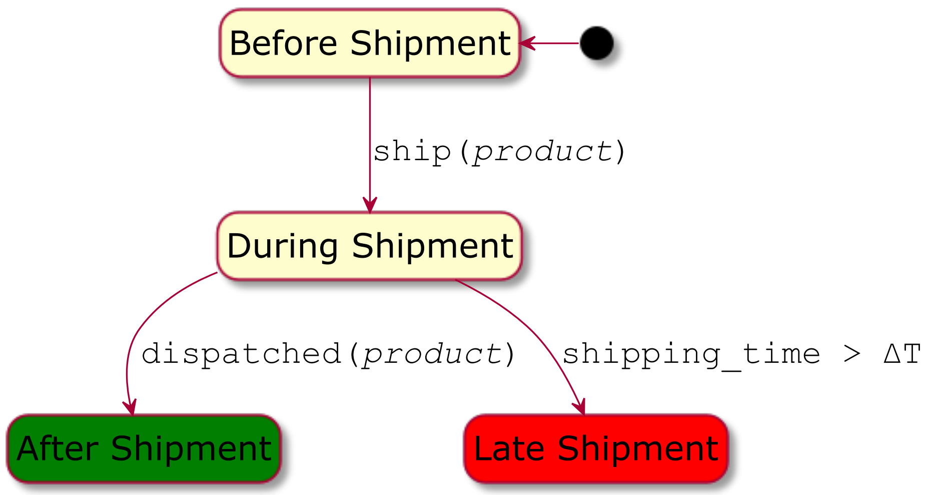

Figure 7. Time-reactive state diagram for the SC in Listing 1.

Listing 1. Automatic refund in case of late shipment.

```
init(Duration) :-
  set_data(state, before_shipment),
  set_data(expected_duration, Duration).

receive(ship(product), manufacturer) :- self(Me),
  get_data(state, before_shipment),
  get_data(expected_duration, DT),
  set_data(due(manufacturer, Me), euro(13))
  set_data(state, during_shipment),
  delay(DT, late).

receive(late, Me) :- self(Me),
  get_data(state, during_shipment),
  set_data(state, late_shipment),
  set_data(due(manufacturer, Me), 0),
  set_data(due(Me, manufacturer), euro(13)),
  send(refund(euro(13)), manufacturer).

receive(dispatched(product), distributor) :-
  get_data(state, during_shipment),
  set_data(state, after_shipment).
```

The functioning of the SC in Listing 1 is quite straightforward. By construction, it is assumed to know the identities of both the manufacturer and the distributor, other than the ID of the product being shipped. The expected duration of the shipment is provided to the SC upon deployment. After that, the SC *(i)* waits for the manufacturing company to initiate the shipping procedure; *(ii)* as soon as the shipping procedure is initiated, it schedules the automatic refunding for a later moment in time; *(iii)* then, it waits for the distributor company to acknowledge the product dispatch. The final outcome of the shipping procedure depends on which event occurs first: in case the distributor acknowledgement firstly occurs, then no refund is performed; otherwise, refund is automatically performed and the late acknowledgement is ignored.

4.2. Automatic Applications to Public Calls

In many countries, like for instance in Italy, career advancements in the public sector require a public tender to be won by a candidate employee. A public call is usually advertised on an official repository, instructing candidates on when and how they should submit their applications. Under such hypotheses, application submission management can easily become a trust-critical scenario where the adoption of BCT and smart contracts may introduce great advantages, by easily preventing situations where *(i)* one or more submitters lie about the timing of their applications, *(ii)* one or more corrupted employees tamper with one or more applications, or *(iii)* one or more applications are disclosed before the public submission period is over. The smart contract described below can be exploited as a countermeasure to the aforementioned issues.

Listing 2 shows the source code of an autonomous smart contract aimed at managing submissions. After deployment, and as long as the application period is open, it continuously collects submissions sent by other smart contracts or end users. More precisely, *(i)* it knows when to open and close applications, since this information is provided as input upon deployment; *(ii)* after deployment, there are only two possible situations: either applications are already open or they should be opened in a later moment; *(iii)* in the latter case, application opening is scheduled for the opening date provided as input upon SC deployment; *(iv)* in any case, after applications have been opened, the SC accepts submissions as long as the tender is open; finally *(v)* it exposes applications for public inspection only after the application period has been closed.

It is worth emphasizing what are the enablers of the greater expressiveness exhibited by Tenderfone smart contracts w.r.t. the state of the art: *(i)* computational autonomy, enabling smart contracts to set their own goals upon deployment and act as soon as applications need to be opened/closed without the need for external stimuli, and *(ii)* temporal situatedness, enabling them to perceive time and perform time-dependant computations.

Listing 2. A Tenderfone SC `autonomously managing an application submission` system.

```
% assume opening & closing date are provided upon SC deployment
init([OpeningDate, ClosingDate]) :-
  ClosingDate > OpeningDate, % closing date must be after opening date
  now(Now), Now < ClosingDate, % closing date should be in the future
  set_data(opening_date, OpeningDate), % store opening date for later usage
  set_data(closing_date, ClosingDate), % store closing date as well
  set_initial_status(Now, OpeningDate). % sets the initial state of this SC

set_initial_status(Now, OpeningDate) :- % when setting the initial state...
  Now >= OpeningDate, % ... if the opening date is in the past
  set_data(status, open). % it means submissions are already open
set_initial_status(Now, OpeningDate) :- % otherwise ...
  Now < OpeningDate, % ... if the opening date is in the future
  set_data(status, wait_opening),  % it means submissions are not open, yet
  when(OpeningDate, open_applications). % thus opening is scheduled

receive(open_applications, Sender) :- % when trying to open applications
  self(Sender), % only this smart contract can open applications
  get_data(status, will_open), % ensure not already open
  set_data(status, open), % applications are now open
  get_data(closing_date, ClosingDate), % retrieve closing date
  when(ClosingDate, close_applications). % schedule applications closing

receive(close_applications, Sender) :- % when trying to close applications
  self(Sender), % only this smart contract can close applications
  get_data(status, open), % ensure applications are open
  set_data(status, closed). % applications are now closed

receive(application(Details), Sender) :- % when submitting an application
  get_data(status, open), % ensure applications are open
  set_data(applicant(Sender), application(Details)). % store application

answer(application(Applicant), _, Details) :- % when quering an application
  get_data(status, closed), % ensure applications are closed
  get_data(applicant(Applicant), Details). % retrieve the application
```

5. Conclusions & Outlook

There are many motivations for seeking an integration of agent-oriented models and technologies with the blockchain, as well as the potential benefits such a marriage would bring to both worlds. In this paper, we tried to shed some light over the state-of-the-art integration attempts, mostly adopting a “agent-vs-blockchain” approach. We emphasize that an alternative approach is possible, which we denote as “agent-to-blockchain”. Then, we acknowledge the existence of two dimensions of integration, a computational and an interaction one, and we articulate the “agent-to-blockchain” approach along a roadmap enhancing smart contracts towards full agency, in both dimensions. Finally, we discuss Tenderfone as a custom blockchain providing pro-active smart contracts as a first step along the aforementioned roadmap, endowing smart contracts with control flow encapsulation, reactiveness to time, and asynchronous communication means.

Our future work will be devoted to further advance along the roadmap, and to complete conceptual analysis of the integration opportunities by reflecting on "blockchain-to-agent" approaches aimed at injecting blockchain and smart contracts concepts into the agent abstraction. In particular, the strict guarantees on execution time/effort provided by smart contracts (e.g., in the form of estimates on gas consumption in Ethereum) could be something to look at to be transferred to the agent domain. Having agents able to estimate time taken or computational effort to complete a task seems a desirable property for any MAS. Adopting consensus as the foundational layer for any distributed computation undertaken by smart contracts also seems promising in the agent realm as well. Agents could then rely on a shared understanding of the state of the world, continuously built and preserved by the underlying execution platform.

We hope that public availability of the Tenderfone platform will gather interest by researchers and programmers, so as to provide a common ground for experimenting the many steps left along the presented stairways.

Author Contributions: Conceptualisation, S.M., F.Z., A.O. and G.C.; methodology, S.M.; software, G.C.; investigation, S.M.; writing—original draft preparation, S.M. and G.C.; writing—review and editing, SM, F.Z., A.O. and G.C.; supervision, A.O. and F.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been partially supported by the H2020 Project "AI4EU" (G.A. 825619) and by the MIUR PRIN 2017 Project "Fluidware".

Acknowledgments: The authors would like to thank Alfredo Maffi for the implementation of the first version of Tenderfone.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Herlihy, M. Blockchains from a Distributed Computing Perspective. *Commun. ACM* **2019**, *62*, 78–85, doi:10.1145/3209623.
2. Stanciu, A. Blockchain Based Distributed Control System for Edge Computing. In Proceedings of the 2017 21st International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania, 29–31 May 2017; pp. 667–671.
3. Shafagh, H.; Burkhalter, L.; Hithnawi, A.; Duquennoy, S. Towards Blockchain-based Auditable Storage and Sharing of IoT Data. In Proceedings of the 2017 on Cloud Computing Security Workshop, Dallas, TX, USA, 3–5 November 2017; pp. 45–50.
4. Xu, Q.; Aung, K.M.M.; Zhu, Y.; Yong, K.L. A Blockchain-Based Storage System for Data Analytics in the Internet of Things. In *New Advances in the Internet of Things*; Yager, R.R., Pascual Espada, J., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 119–138.
5. Casado-Vara, R.; Prieto, J.; la Prieta, F.D.; Corchado, J.M. How blockchain improves the supply chain: case study alimentary supply chain. *Procedia Comput. Sci.* **2018**, *134*, 393–398.
6. Mettler, M. Blockchain technology in healthcare: The revolution starts here. In Proceedings of the 2016 IEEE 18th International Conference on e-Health Networking, Applications and Services (Healthcom), Munich, Germany, 14–17 September 2016; pp. 1–3.
7. Calvaresi, D.; Dubovitskaya, A.; Calbimonte, J.P.; Taveter, K.; Schumacher, M. Multi-Agent Systems and Blockchain: Results from a Systematic Literature Review. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection*; Demazeau, Y., An, B., Bajo, J., Fernández-Caballero, A., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 110–126.
8. Calvaresi, D.; Dubovitskaya, A.; Retaggi, D.; F. Dragoni, A.; Schumacher, M. Trusted Registration, Negotiation, and Service Evaluation in Multi-Agent Systems throughout the Blockchain Technology. In Proceedings of the 2018 IEEE/WIC/ACM International Conference on Web Intelligence (WI), Santiago, Chile, 3–6 December 2018; pp. 56–63.

9. Amato, F.; Femia, P.; Moscato, F. Enabling Accountable Collaboration in Distributed, Autonomous Systems by Intelligent Agents. In *Complex, Intelligent, and Software Intensive Systems*; Barolli, L., Hussain, F.K., Ikeda, M., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 807–816.
10. Papi, F.G.; Hübner, J.F.; de Brito, M. Instrumenting Accountability in MAS with Blockchain. In Proceedings of the First Workshop on Computational Accountability and Responsibility in Multiagent Systems co-located with 20th International Conference on Principles and Practice of Multi-Agent Systems, Nice, France, 30 October–3 November 2017; pp. 20–34.
11. Castelló Ferrer, E. The Blockchain: A New Framework for Robotic Swarm Systems. In *Future Technologies Conference (FTC)*; Arai, K., Bhatia, R., Kapoor, S., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 1037–1058.
12. Ciatto, G.; Mariani, S.; Omicini, A. Blockchain for Trustworthy Coordination: A First Study with Linda and Ethereum. In Proceedings of the 2018 IEEE/WIC/ACM International Conference on Web Intelligence (WI), Santiago, Chile, 3–6 December 2018; pp. 696–703.
13. Macrinici, D.; Cartofeanu, C.; Gao, S. Smart contract applications within blockchain technology: A systematic mapping study. *Telemat. Inform.* **2018**, *35*, 2337–2354, doi:10.1016/j.tele.2018.10.004.
14. Wegner, P. Why Interaction is More Powerful Than Algorithms. *Commun. ACM* **1997**, *40*, 80–91, doi:10.1145/253769.253801.
15. Ciatto, G.; Mariani, S.; Maffi, A.; Omicini, A. Blockchain-Based Coordination: Assessing the Expressive Power of Smart Contracts. *Information* **2020**, *11*, 52, doi:10.3390/info11010052.
16. Ciatto, G.; Maffi, A.; Mariani, S.; Omicini, A. Smart Contracts are More than Objects: Pro-activeness on the Blockchain. In *Blockchain and Applications*; Prieto, J., Kumar, A.D., Ferretti, S., Pinto, A., Corchado, J.M., Eds.; Springer: New York, NY, USA, 2020; Volume 1010, pp. 45–53.
17. Christidis, K.; Devetsikiotis, M. Blockchains and Smart Contracts for the Internet of Things. *IEEE Access* **2016**, *4*, 2292–2303, doi:10.1109/ACCESS.2016.2566339.
18. Gatteschi, V.; Lamberti, F.; Demartini, C.; Pranteda, C.; Santamaría, V. Blockchain and Smart Contracts for Insurance: Is the Technology Mature Enough? *Future Internet* **2018**, *10*, 20, doi:10.3390/fi10020020.
19. Ciatto, G.; Maffi, A.; Mariani, S.; Omicini, A. Towards Agent-oriented Blockchains: Autonomous Smart Contracts. In *Advances in Practical Applications of Survivable Agents and Multi-Agent Systems: The PAAMS Collection*; Demazeau, Y., Matson, E., Corchado, J.M., De la Prieta, F., Eds.; Springer International Publishing: Cham, Switzerland, 2019; Volume 11523, pp. 29–41.
20. Szabo, N. Smart Contracts. 1994. Available online: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html (accessed on 22 October 2020).
21. Wood, G. Ethereum: A Secure Decentralised Generalised Transaction Ledger. 2014 Available online: https://ethereum.github.io/yellowpaper/paper.pdf (accessed on 22 October 2020).
22. Androulaki, E.; others. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In Proceedings of the 13th EuroSys Conference (EuroSys'18), London, UK, 18–21 April 2018.
23. Stark, J. *Making Sense of Blockchain Smart Contracts*; CoinDesk: New York, NY, USA, 2016.
24. Odell, J. Objects and Agents Compared. *J. Object Technol.* **2002**, *1*, 41–53, doi:10.5381/jot.2002.1.1.c4.
25. Omicini, A.; Ricci, A.; Viroli, M. Artifacts in the A&A Meta-Model for Multi-Agent Systems. *Auton. Agents-Multi-Agent Syst.* **2008**, *17*, 432–456, doi:10.1007/s10458-008-9053-x.
26. Suchman, L.A. *Plans and Situated Actions: The Problem of Human-Machine Communication*; Cambridge University Press: New York, NY, USA, 1987.
27. Omicini, A.; Ricci, A.; Viroli, M. Timed Environment for Web Agents. *Web Intell. Agent Syst.* **2007**, *5*, 161–175.
28. Ricci, A.; Omicini, A.; Viroli, M.; Gardelli, L.; Oliva, E. Cognitive Stigmergy: Towards a Framework Based on Agents and Artifacts. In *Environments for MultiAgent Systems III*; Springer: New York, NY, USA, 2007; Volume 4389, pp. 124–140.
29. Conte, R.; Castelfranchi, C. *Cognitive and Social Action*; UCL Press: London, UK, 1995; p. 215.
30. Xu, X.; Pautasso, C.; Zhu, L.; Gramoli, V.; Ponomarev, A.; Tran, A.B.; Chen, S. The Blockchain as a Software Connector. In Proceedings of the 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), Venice, Italy, 5–8 April 2016; pp. 182–191.

31. Ferber, J.; Müller, J.P. Influences and Reaction: A Model of Situated Multiagent Systems. In Proceedings of Second International Conference on Multi-Agent Systems (ICMAS-96), Kyoto, Japan, 9–13 December 1996; pp. 72–79.
32. Castelfranchi, C.; Dignum, F.; Jonker, C.M.; Treur, J. Deliberative Normative Agents: Principles and Architecture. In *Intelligent Agents VI. Agent Theories, Architectures, and Languages*; Jennings, N.R., Lespérance, Y., Eds.; Springer: New York, NY, USA, 2000; Volume 1757, pp. 364–378.
33. O'Brien, P.D.; Nicol, R.C. FIPA—Towards a standard for software agents. *BT Technol. J.* **1998**, *16*, 51–59.
34. Kwon, J. Tendermint: Consensus without Mining. 2014. Available online: https://tendermint.com/static/docs/tendermint.pdf (accessed on 22 October 2020).
35. Denti, E.; Omicini, A.; Ricci, A. tuProlog: A Light-weight Prolog for Internet Applications and Infrastructures. In *Practical Aspects of Declarative Languages*; Springer: New York, NY, USA, 2001; Volume 1990, pp. 184–198.
36. Kowalski, R.A. Predicate Logic as Programming Language. In Proceedings of the 1974 IFIP Congress, Stockholm, Sweden, 5–10 August 1974; pp. 569–574.
37. Ciatto, G.; Calegari, R.; Mariani, S.; Denti, E.; Omicini, A. From the Blockchain to Logic Programming and Back: Research Perspectives. In Proceedings of the 19th Workshop "From Objects to Agents", Palermo, Italy, 28–29 June 2018; pp. 69–74.
38. Fischer, M.J.; Lynch, N.A.; Paterson, M.S. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* **1985**, *32*, 374–382, doi:10.1145/3149.214121.