From the Blockchain to Logic Programming and Back: Research Perspectives

Giovanni Ciatto*, Roberta Calegari*, Stefano Mariani†, Enrico Denti*, Andrea Omicini*
*Department of Computer Science and Engineering (DISI)
ALMA MATER STUDIORUM–Università di Bologna, Italy
Email: roberta.calegari@unibo.it, giovanni.ciatto@unibo.it, enrico.denti@unibo.it, andrea.omicini@unibo.it
†Department of Sciences and Methods for Engineering (DISMI)
Università degli Studi di Modena e Reggio Emilia, Italy
Email: stefano.mariani@unimore.it

***Abstract*—The blockchain is a novel approach to support distributed systems enabling a common, consistent view of a shared state among distributed nodes. There, smart contracts are computer programs that allow users to deploy arbitrary computations, in charge of automatically regulate state transitions and enforce properties. In this paper we speculate on how the blockchain and smart contracts could take advantage of a logic programming approach, and, complementarily, on how logic programming can benefit from the blockchain infrastructure. Accordingly, we discuss some possible research directions and open questions for future research.**

***Index Terms*—blockchain, logic programming, smart contracts.**

I. INTRODUCTION

The blockchain technology (BCT henceforth) gained attention as the backbone of cryptocurrencies such as Bitcoin [1]. Yet, its capabilities extend far beyond that, enabling on the one hand existing applications to be improved – such as the Domain Name System [1], identity or copyright management [2] –, on the other hand novel ones to be conceived and deployed—such as DAOs [3] or crytocurrencies [4] themselves. A lot of expectations lie around such a novel approach to distributed systems [2], mostly concerning the way financial transactions, identity management, or asset property tracking are performed.

A popular categorisation of blockchain models orders them in three generations [5]: the first limited to cryptocurrencies, the second generalising to any sort of asset tracking, and the third introducing *smart contracts* [6]. Smart contracts are computer programs deployed "on the blockchain", meant to reify agreements between mutually distrusting participants automatically enforced by the blockchain consensus mechanism—without resorting to a trusted centralised authority.

The implementation of smart contracts offered by nowadays BCT constrains users to express their business logic by means of object-oriented programming (OOP) abstractions and an imperative programming style—both strictly bound to the underlying BCT. In the following we take Ethereum[2] as our reference as the most well-known framework for smart contracts currently available [7].

We suggest that the choice of OOP may have hindered the full realisation of the expectations related to third generation BCTs. In fact, smart contracts were originally conceived as a means to *automatically enforce* the conditions defined in a contract, and to *guarantee invariants* and properties despite the ever-changing state of the blockchain [8], [9], [10].[3] Not exactly what OOP is best for.

So, there seems to be a *mismatch* between the computational model chosen for Ethereum's smart contracts and the one actually fitting their original requirements and intended purpose. Among the many programming languages and computational models available – as Solidity[4] for Ethereum, Go for the Hyperledger Fabric[5], or Java for Corda[6] – a declarative language backed by a *logic programming* computational model could be best suited for both *(i)* expressing high level policies in an unambiguous and *human-readable* way, and *(ii)* supporting *inference* and sound demonstration of properties of interest. In turn, as a distributed ledger technology, the blockchain could bring to distributed logic programming more than a few desirable features, such as consistency, accountability, and fault tolerance.

Accordingly, in this paper we identify the research opportunities arising from re-interpreting the blockchain from a logic-based perspective: on the one hand, how logic programming can improve the capabilities of the blockchain and, especially, of smart contracts; on the other hand, how logic programming may benefit from the blockchain, either as a distributed computing model or as an infrastructure.

The reminder of the paper is therefore organised as follows: Section II provides an overview about the main elements of BCTs and the smart contract abstraction, Section III introduces our vision of logic-based smart contracts, describing how logic programming would provide some useful properties, in Section IV we discuss how the BCT introduces novel ways to face well-known issues of the distributed logic programming research area, and, finally, Section V concludes the paper.

[1] http://www.namecoin.org/
[2] http://www.ethereum.org

[3] As detailed in the next section despite the similar name, our notion of logic-based smart contracts differs from the one proposed in [8].
[4] http://solidity.readthedocs.io
[5] http://www.hyperledger.org/projects/fabric
[6] http://www.corda.net/

II. Blockchain and Smart Contracts: Overview

The blockchain is essentially a means for distributed nodes to consistently perceive a common shared state of the system: essentially, it provides a novel way to address the problem of state machine replication [11], focusing on making some distributed processes expose the very same dynamics, given that they share the same program and initial state. As such, the blockchain provides a way to solve (or, at least, mitigate) some well known problems of distributed systems, such as the CAP [12] and FLP [13] theorems, the Sybil attack [14], Lamport's Byzantine Generals Problem [15].

The key point is that the blockchain endows applications with highly desirable properties such as immutability and untamperability of the past, consistency among nodes of the system state, fault tolerance of both data and computations. In the following, we describe the elements actually enabling these features, which are:

- the system (or world) state
- the transactions describing state transitions
- the blocks forming an ordered ledger containing all the events that occurred within the system
- the consensus mechanism (or protocol)
- smart contracts (3rd generation blockchain only)

A. States

States represent what is true for the system at a given instant in time. All nodes composing the system must *eventually* perceive the same system state. Generally speaking, it is a mapping between the entities' identifiers – i.e. users – and some arbitrary data associated to and controlled by that entity, there including smart contracts data and code, or user balances.

B. Transactions

Transactions encode the state variations yet to be performed. A transaction can apply to a state producing a new state. Nodes may issue (publish) transactions in order to produce an effect on the system state. Transactions may represent money transfer from an entity to another, the deployment of a smart contract, or the invocation of an already-deployed smart contract.

When a transaction is published, it is *replicated* over the whole system, thus making every node aware of it. However, a transaction can be applied only if it is *valid*, that is, well formed and correctly signed by the issuer. Otherwise, it is simply dropped, thus preventing unauthorised actions to carry on. Furthermore, to produce an effect on the system state, transactions must be executed without errors, otherwise their modifications to the system are simply reverted. As an example, suppose an entity is trying to transfer more money than it currently owns, by means of a transaction. Such a transaction would eventually throw an error provoking the money transfer and any other side effect to be reverted.

C. Blocks

Blocks are timestamped lists of transactions. As such, they certify *(i)* the *ordering* of edits applied to the system state, and *(ii)* that each transaction occurred at a specific instant in time, and therefore the resulting state is valid from that moment on. Blocks implement the timestamping schema described in [16], providing *untamperability* of the history about the past.

In fact, locks are replicated on all the nodes thanks to the blockchain protocol, which is in charge of ensuring that the last block – containing the most recent transactions – is the same for each node.

D. Consensus

The distributed consensus protocol makes sure that the distributed nodes – the *miners* – achieve a common view of the sequence of blocks, and, consequently, of the ordering of transactions: in short, its task is to *guarantee consistency* in spite of system state replication. While several consensus mechanisms exist, their detail is outside the scope of this paper: we forward the interested reader to [11].

For what concerns the following discussion it is sufficient to understand that BCTs supporting cryptocurrencies (such as Ethereum) usually employ the Proof-of-Work consensus approach, which *(i)* endows the cryptocurrency with its *economical* value, *(ii)* provides probabilistic and eventual consistency of the system state, and *(iii)* makes the blockchain resistant against Byzantine nodes and Sybil attacks.

E. Smart contracts

Smart contracts consist of *stateful* processes that can react to the publication of a particular kind of transaction: *invocation transactions*. For an invocation transaction to be valid, the triggered computation must terminate *without errors*: otherwise, its effects are reverted, leaving the system state unaltered.

Smart contracts can be conceived (and usually are) as objects in the OOP sense, thus their invocations as method calls. Nevertheless, deployment is quite peculiar: their programs are written and published by end users by means of *deployment* transactions, publishing the compiled source code (or, bytecode) of a smart contract. In Ethereum, smart contracts are usually programmed with Solidity, which is indeed an object-oriented language. The Solidity code is compiled into the Ethereum Virtual Machine (EVM) bytecode before deployment and the latter is deployed on the blockchain network. Thanks to transaction signatures and blocks hash links, the code of smart contracts is *immutable* after publication. This is what makes them amenable of trust in, e.g., handling financial transactions: indeed, since the source code is publicly inspectable and immutable, anyone can check the bytecode obtained (i.e. via a disassembler).

F. Miners

Miners are the peer nodes composing the backbone of the blockchain network by participating into the consensus mechanism. They locally store a copy of the whole blockchain – there included a copy of each smart contract – and execute all transactions. Since the system state must be kept consistent among all the miners, each of them locally executes smart contracts upon reception of invocation transactions. Therefore, the execution of non-terminating computations must be prevented,

since it would imply a deadlock of the whole blockchain network. The most notable solution is the one introduced by the Ethereum platform, where each computational step is *paid* by the initiator of the transaction in terms of *gas*—converted from the initiator balance. Executions running out of gas invalidate their invoking transactions but still consume the initiator's money.

These money can be redeemed by the first miner producing the next block of the blockchain, as a compensation for the computational effort. So, long/infinite computations become economically unsustainable. This mechanism highlights the aspect of the economical cost model of computations, which is currently instruction-based [17].

III. Logic Programming for Smart Contracts

We now introduce the idea of *logic-based* smart contracts as an intriguing research perspective, highlighting its challenges and possible benefits. It is worth remarking that we do not mean to produce "yet another compiler" for a logic language to EVM, but to replace the EVM itself with a logic engine and related computational model, so that smart contracts can be expressed in a logic language—e.g., Prolog.

A. Motivations

We see three major motivations for doing so: *(i)* the *declarative* nature of the source code, *(ii) observability* of the business logic, and *(iii) controlled mutability* of behaviour.

Adoption of a declarative language with a goal-oriented semantics could ease both developers' work and readers' understanding in general: the former would be able to explicitly state the business goals the smart contracts should pursue – which could in turn be able to share them or reason about how to reach them –, while for the latter would be easier to understand the functioning of already-deployed smart contracts. Furthermore, since Prolog is an interpreted language, it could also help strengthening trust via improved *inspectability*: in fact the source code may be deployed with no need of compilation, and it would not require any disassembler for being inspected later on.

By endowing logic-based smart contracts with a *static* knowledge base (KB) containing immutable rules and terms, alongside a *dynamic* KB containing the mutable ones, smart contracts could be structured in such a way to partially modify their behaviour in a controllable way. A practical example of such an interesting feature is extensively described in Subsection III-C. This helps mitigating the problems caused by buggy smart contracts that are deployed and, being immutable, cannot be fixed—anyway, a practice considered harmful [18]. Again, the fact that both the static and dynamic KBs can be easily inspected would be an added value, possibly paving the way towards cryptographically secured, while human-readable, contracts.

B. A possible roadmap

As a first step towards our vision we need to re-interpret the blockchain elements summarised in Section II according to a logic-based perspective. To this end, let us suppose that

- the system state maps smart contract identifiers into some data structure, including at least a balance to enable smart contracts to handle money, a static KB made of an immutable list of logic facts and rules, to be set upon deployment, and an initially empty dynamic KB which may be modified via `assert/1` and `retract/1`
- money transfer transactions retain their semantics, whereas smart contract invocations become goal-oriented computations as described in the following
- blocks and consensus retain their semantics

As a second step, we need to define the behaviour of logic based smart contracts, that is, how they react to transactions. While money transfer transactions retain their semantics, invocation transactions can be supposed to specify a logic term, say *Message*, that triggers a goal-oriented computation *Goal* if the KB of the invoked smart contract contains a rule such as `receive(+Message, +Guard):- Goal.` provided that *Guard* is true. If no such a rule is found, the transaction is considered invalid and it produces no effect.

The computation (*Goal*) is performed according to the usual SLD resolution process [19]: the side effects (assertion or retraction of terms) possibly performed affect only the dynamic KB of the smart contract. However, for the transaction to be considered valid and, consequently, its side effects to be persisted on the blockchain, the resolution process should terminate producing a proof for *Goal*: this is persisted along with the invocation transaction. The issuer of the transaction (or possibly any other interested party) may later inspect its local copy of the blockchain seeking for such a proof, if interested. In essence, the possibly many `receive/2` rules are the only access point to the smart contract code – i.e., its API –, while other clauses are considered an implementation detail of the smart contract and cannot be accessed by users. Such a mechanism would let the programmers be free to allow (or deny) the injection of some new behaviour into the smart contract by means of, e.g., the assertion of some new rule. This is ultimately what we mean by "controlled mutability".

The third step should focus on inter-smart contract interaction. As the reader may expect, a `send(+Recipient, ?Message, ?Money)` primitive has been conceived too, which can be used by a smart contract to send a *Message* to another one (namely *Recipient*), possibly including some *Money*, therefore triggering one of its `receive(Message, Guard)` entry points. An interesting aspect which may be worth of further investigation is whether the `send/2` primitive should have a *synchronous* or an *asynchronous* communication semantics. In the first case, the inter-smart contracts interaction semantics would redeem their OOP-like nature, along with its well known re-entrancy issues [20], [21]. We speculate that the second case may mitigate the aforementioned issues and open new opportunities related to the engineering of smart contracts interaction patterns, but further research is needed to prove this claim. To complete the picture, some convenience predicates could be built-in into all static KBs, mimicking

Solidity's "Globally Available Variables"[7] like:

- `now(?Time)` may enable a smart contract to inspect the current time, or activate a given `prove` rule if the current time is after/before a given instant
- `sender(?EntityID)` may enable a smart contract to inspect the identity of the issuers of the transaction
- `value(?Money)` may enable a smart contract to know how much money it received along with the last message

Such predicates essentially provide *context awareness* in the sense that they allow the smart contract to reason and act taking into account the informations contained *(i)* within the lastly received transaction – such as the sender identity –, *(ii)* within the lastly published block—e.g., the last commonly accepted global time, etc. Examples showcasing the utility of these predicates are described in Subsection III-C.

The semantics of blocks and of the consensus algorithm remain unchanged: this implies no interference while a resolution process is being executed, since transactions are totally ordered and sequentially executed. The miners' role, too, remains unchanged: simply, in this vision they are in charge of the SLD resolution process. However, removing the EVM bytecode from the picture opens up the possibility of defining a more coarse-grained computation cost model, taking into account the specific features of the logic-based computational model—with the usual aim of discouraging long computations. For instance, should users pay for the proofs to their queries? Should they pay for unification of clauses and facts? What about paying for backtracking? These questions have no trivial answer at the moment, thus deserve further research.

C. Case study

As an illustrative example, consider a scenario where the flight company ACME Inc. exploits smart contracts for regulating costumer management. The company handles purchase of tickets through its `ACMEPortal` smart contract, which provides a number of functionalities, among which:

1) looking up for flights towards a given destination (`To`) within a given timespan (`Day`):

```
receive(look_up(Day, From, To), (now(Now), after(Day,
    Now))) :-
  setof(
    flight(Id, Time, Duration, From, To, Price),
    (flight(Id, Time, Duration, From, To, Price),
     within(Time, Day)),
    Flights
  ),
  sender(Customer),
  send(Customer, Flights, _).
```

where `within(+TimeSpan, ?Instant)` simply checks whether `Instant` is within `TimeSpan`.

2) reserving a seat paying the corresponding `Price`:

```
receive(reserve(Id), (flight(Id, Time, Dur, From, To,
    Price), value(X), X >= Price)) :-
  sender(Customer),
  assert(reservation(Customer, flight(Id, Time, Dur,
    From, To, Price))).
```

3) cancelling a reservation while getting a refund, possibly with a fee if cancellation occurs later than 3 days before expected departure:

```
receive(cancel(Id), (sender(Customer), reservation(
    Customer, flight(Id, ExpDeparture, _, _, Price)))
    ) :-
  now(CurrentTime),
  retract(reservation(Customer, flight(Id, _, _, _, _)
    )),
  compute_refund(CurrentTime, ExpDeparture, Price,
    Money),
  send(Customer, _, Money).
```

where computation of the amount to refund is encapsulated by the `compute_refund/4` predicate, assumed to be in the *dynamic* KB of the smart contract:

```
compute_refund(CancellationTime, DepartureTime, Price,
    Price) :-
  DepartureTime - CancellationTime > days(3).

compute_refund(CancellationTime, DepartureTime, Price,
    ToBeRefunded) :-
  DepartureTime - CancellationTime =< days(3),
  ToBeRefunded is Price / 2.
```

4) automatically getting a refund in case of delays longer than some given amount of time:

```
receive(landing(flight(Id, ExpDeparture, Dur, _),
    ActualTime), true) :-
  setof(
    refund(Customer, Price),
    (reservation(Customer, flight(Id, ExpDeparture,
    Dur, _, _, Price)), too_late(ExpDeparture, Dur,
    ActualTime)),
    CustomersToBeRefunded
  ),
  refund_all(CustomersToBeRefunded).
```

where `refund_all/1` is a simple broadcast:

```
refund_all([]).
refund_all([refund(Customer, Price) | Rs]) :-
  send(Customer, _, Price),
  refund_all(Rs).
```

To support functionality §4, airports are assumed to deploy a `AirportNotification` contract in charge of notifying `ACMEPortal` about the actual departure and landing times, by sending messages in the form:

```
departure(flight(Id, ExpDeparture, Duration, FromAirport),
    ActualDepatureTime)

landing(flight(Id, ExpDeparture, Duration, ToAirport),
    ActualLandingTime)
```

Functionality §1 collects all flights information matching a custom filter. For instance, users may ask for all the flights departing from a given place on a given day by providing an unbounded `To` variable as input. The operation also shows how contextual predicates may be useful to implement user- and time-aware operations.

Functionality §2 shows how to add new data to the dynamic KB of the smart contract. Notice that, since functionailities are wrapped within the `receive/2` primitive, no entity except the smart contract itself can add (or remove) information to its

[7] http://solidity.readthedocs.io/en/latest/units-and-global-variables.html

KB. This is how *encapsulation* of smart contracts KBs, i.e., their states, can be achieved.

Functionality §3 shows the dual operation, namely, how to remove data, while illustrating what we mean by *controlled mutability*. Suppose ACME Inc. wants to adopt a different policy for cancellations. Assuming the smart contracts exposes one more functionality:

```
receive(compute_refund(C, D, P, T) :- NewPolicy), (sender(S
    ), authorised(S))) :-
  retract_all(compute_refund(_, _, _, _) :- _),
  assert(compute_refund(C, D, P, T) :- NewPolicy).
```

then it would be easy for the IT department to dynamically inject the novel policy as a replacement of the previous one. In Ethereum, for instance, such a flexibility is not possible without carefully engineering some cumber-stone interaction pattern between two or more smart contracts.

Finally, functionality §4 illustrates how to compose smart contracts declaratively to give a refund to customers in case of delayed flights, *automatically*. This is an example of inter-smart contracts interaction through the `send` and `receive` primitives involving more than two smart contracts.

IV. The Blockchain for Logic Programming

We now switch perspective w.r.t. previous section, that is, we explore the idea of exploiting the blockchain technology to face some well-known issues of logic programming in distributed systems—namely, consistency of a distributed, possibly dynamic logic theory which evolves over time, and concurrent reasoning. The blockchain may support for instance *cooperative* exploration of the proof tree for a given goal, while smart contracts may be the key for letting different, *para-consistent* theories coexist within the same system. As a first intuition, the blockchain can act as the *mediator* giving distributed access to the logical theory representing the knowledge to share.

In the following we discuss two distinct scenarios: in the first, the ledger is re-interpreted as a single distributed logic theory where nodes may participate to the resolution process, while in the second, smart contracts are conceived as containers of immutable logic theories.

A. Single theory

Several approaches for distributing logic programming have been historically explored, in particular towards implicit parallel evaluation, leading to explore AND-parallelism and OR-parallelism [22], [23], [24]. The most relevant results [25] perform well but are not meant to face distributed programming. Yet, implicit parallelism lacks two important control mechanisms—synchronisation of logic processes, and control over the non-determinism of schedulers.

The blockchain model and technology could open up new perspectives in the distribution and parallelisation of the resolution process with respect to a single theory shared between different participants of the distributed system. In this vision, a first intriguing possibility is to distribute the goal to solve over the blockchain: the blockchain system is so re-interpreted as an inference engine coordinating a number of processors – the miners – aimed at *(i)* handling users' *assert*, *retract*, and *solve* requests, or *(ii)* carrying on goal proving processes by concurrently exploring different paths of some goal's proof tree. We therefore imagine transactions as:

- information creation/consumption, in the form of *logic clauses*, to the distributed ledger—e.g., `assert(?Clause)`/`retract(?Clause)`
- inference invocation aimed at triggering a resolution process on a given goal—e.g., `solve(?Goal)`

At the same time, miners are re-interpreted as the entities in charge for *repeatedly* performing *partial* explorations of the proof tree or producing solutions for provable *sub*-goals. By keeping track of already proved (sub-)goals too – on the ledger – miners can avoid wasting their computational resources proving what has been proved by others.

Finally, economical incentives could be designed and tuned to stimulate a particular joint exploration strategy of the proof tree. For instance, the blockchain protocol may initially reward the most miners exploring novel branches of the proof tree or finding solutions for unproven (sub-)goals, but, after that, it may also start encouraging miners to keep exploring the same path of the proof tree, in order to minimise the probability of two ones competing to prove the same (sub-)goal.

Although our vision appears to be feasible in practice, further research is needed when facing the theoretical foundations of cooperative reasoning on top of a blockchain. In particular, the lesson learned from concurrent logic languages such as Concurrent Prolog [23] and Parlog [26] may provide important and useful insights: e.g., how shared variables may be handled along with AND-parallelism, or how to prevent concurrent updates to the shared logic theory. Also, the termination of recursive resolution processes too may be a critical aspect, since non-termination may imply live-locks between miners. As a future work, we plan to study whether the above issues can be mitigated through economic (dis)incentives—similarly to what cryptocurrencies essentially do.

B. Multiple theories

All the above blockchain benefits in guaranteeing properties like consistency can be re-thought in a multiple theories scenario, where consistency is referred to the *local* situation, for instance. There, diverse logical theories are scattered around, representing the local knowledge of each node situated in space and time [27], [28]. Even if the initial knowledge is the same, the KB is then likely to evolve over time in different ways, due to space-time situatedness (i.e. the temperature of a room). There, each theory is consistent with its local reality while not in contrast with other theories representing truth in a different *locality*—resembling paraconsistent logics [29].

In this context, each theory would be associated to a group of local agents, and possibly merged with other computations in the system only once the result is computed: there the blockchain technology could help in integrating data, and guaranteeing properties like local consistency.

Indeed, smart contracts could play a key role to guarantee the consistency (or other relevant properties) of a logic theory replicated on the subset of network nodes in a "logic neighbourhood": in fact, a smart contract running every time a change to the theory is proposed could help deciding whether the change should be accepted or not. There, the smart contract could also manage assert/retract on the knowledge base, ensuring the (eventual?) consistency of the shared theory. Other examples of properties that could be enforced by such a mechanism are invariants and computational properties like stratification [30].

V. Conclusion

On the one hand, the blockchain could benefit from logic programming to obtain system properties like observability, explainability, and evolvability. On the other hand, blockchain and smart contracts look as promising technologies to exploit the full potential of distributed LP. Accordingly, this paper elaborates on how blockchain and smart contracts could mix with logic programming, and foresees a number of lines of future research on the subject.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: http://bitcoin.org/bitcoin.pdf

[2] S. X. Zibin Zheng, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, In press.

[3] U. Chohan, "The decentralized autonomous organization and governance issues," *SSRN Electronic Journal*, Dec. 2017. [Online]. Available: http://ssrn.com/abstract=3082055

[4] I. Bashir, *Mastering Blockchain – Distributed ledgers, decentralization and smart contracts explained*. Packt Publishing, 2017.

[5] M. Swan, *Blockchain: Blueprint for a New Economy*. O'Reilly, 2015. [Online]. Available: http://shop.oreilly.com/product/0636920037040.do

[6] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, Sep. 1997. [Online]. Available: http://journals.uic.edu/ojs/index.php/fm/article/view/548

[7] S. Omohundro, "Cryptocurrencies, smart contracts, and artificial intelligence," *AI Matters*, vol. 1, no. 2, pp. 19–21, Dec. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685334

[8] F. Idelberger, G. Governatori, R. Riveret, and G. Sartor, "Evaluation of logic-based smart contracts for blockchain systems," in *Rule Technologies. Research, Tools, and Applications*, ser. LNCS, vol. 9718. Springer, 2016, pp. 167–183. [Online]. Available: http://link.springer.com/10.1007/978-3-319-42019-6_11

[9] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. ACM, 2016, pp. 91–96. [Online]. Available: http://dl.acm.org/citation.cfm?id=2993611

[10] M. P. Andersen, J. Kolb, K. Chen, G. Fierro, D. E. Culler, and R. A. Popa, "WAVE: A decentralized authorization system for IoT via blockchain smart contracts," EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2017-234, Dec. 2017. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-234.html

[11] C. Cachin and M. Vukolić, "Blockchain consensus protocols in the wild (keynote talk)," in *31st International Symposium on Distributed Computing (DISC 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 91. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 1:1–1:16. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/8016

[12] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=564601

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985. [Online]. Available: http://portal.acm.org/citation.cfm?id=214121

[14] J. R. Douceur, "The Sybil attack," in *IPTPS '01 Revised Papers from the First International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 251–260. [Online]. Available: http://dl.acm.org/citation.cfm?id=687813

[15] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul. 1982. [Online]. Available: http://dl.acm.org/citation.cfm?id=357176

[16] S. Haber and W. Stornetta, "How to time-stamp a digital document," *Journal of Cryptology*, vol. 3, no. 2, pp. 99–111, 1991. [Online]. Available: http://link.springer.com/10.1007/BF00196791

[17] G. Wood, "Ethereum: a secure decentralised generalised transaction ledger," Ethereum Project, Yellow Paper 151, 2014. [Online]. Available: http://ethereum.github.io/yellowpaper/paper.pdf

[18] Q. Dupont, "Experiments in algorithmic governance. a history and ethnography of "the dao," a failed decentralized autonomous organization," in *Bitcoin and Beyond. Cryptocurrencies, Blockchains, and Global Governance*, 1st ed. London, UK: Routledge, Nov. 2017, ch. 8. [Online]. Available: https://www.taylorfrancis.com/books/e/9781351814089/chapters/10.4324%2F9781315211909-8

[19] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM*, vol. 12, no. 1, pp. 23–41, Jan. 1965. [Online]. Available: http://dl.acm.org/citation.cfm?id=321253

[20] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. ACM Press, 2016, pp. 254–269. [Online]. Available: http://dl.acm.org/citation.cfm?id=2978309

[21] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *Principles of Security and Trust*, ser. LNCS. Springer, Jul. 2017, vol. 10204, pp. 164–186. [Online]. Available: http://link.springer.com/10.1007/978-3-662-54455-6_8

[22] L. Monteiro, "A proposal for distributed programming in logic," in *Implementations of Prolog*, ser. Artificial Intelligence. Chicester, UK: Ellis Horwood Limited, 1984, pp. 329–340.

[23] E. Y. Shapiro, *Concurrent Prolog – Vol. 1: Collected Papers*, ser. Logic Programming. Cambridge, MA, USA: The MIT Press, 1987.

[24] K. L. Clark and S. Gregory, "A relational language for parallel programming," in *1981 Conference on Functional Programming Languages and Computer Architecture (FPCA '81)*. New York, NY, USA: ACM, 1981, pp. 171–178. [Online]. Available: http://dl.acm.org/citation.cfm?id=806776

[25] A. Brogi and R. Gorrieri, "A distributed, net oriented semantics for Delta Prolog," in *TAPSOFT '89: Proceedings of the International Joint Conference on Theory and Practice of Software Development Barcelona, Spain, March 13–17, 1989*, ser. LNCS. Springer, 1989, vol. 351, pp. 162–177. [Online]. Available: http://link.springer.com/10.1007/3-540-50939-9_131

[26] K. L. Clark, "PARLOG: The language and its applications," in *PARLE Parallel Architectures and Languages Europe. Volume II: Parallel Languages. Eindhoven, The Netherlands, 15–19 Jun. 1987. Proceedings*, ser. LNCS. Springer, 1987, vol. 259, pp. 30–53. [Online]. Available: http://link.springer.com/10.1007/3-540-17945-3_2

[27] R. Calegari, E. Denti, S. Mariani, and A. Omicini, "Logic programming as a service in multi-agent systems for the Internet of Things," *International Journal of Grid and Utility Computing*, In press.

[28] ——, "Logic Programming as a Service (LPaaS): Intelligence for the IoT," in *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC 2017)*. IEEE, May 2017, pp. 72–77. [Online]. Available: http://ieeexplore.ieee.org/document/8000070/

[29] H. A. Blair and V. S. Subrahmanian, "Paraconsistent logic programming," *Theoretical Computer Science*, vol. 68, no. 2, pp. 135–154, 1989. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397589901266

[30] P. Glasserman, P. Heidelberger, and P. Shahabuddin, "Gaussian importance sampling and stratification: Computational issues," in *1998 Winter Simulation Conference*, vol. 1. IEEE, Dec. 1998, pp. 685–693. [Online]. Available: http://ieeexplore.ieee.org/document/745051/