Artif Intell Law
https://doi.org/10.1007/s10506-018-9223-3

On legal contracts, imperative and declarative smart contracts, and blockchain systems

Guido Governatori[1] · **Florian Idelberger**[2] · **Zoran Milosevic**[3] · **Regis Riveret**[1] · **Giovanni Sartor**[2] · **Xiwei Xu**[4]

Abstract This paper provides an analysis of how concepts pertinent to legal contracts can influence certain aspects of their digital implementation through smart contracts, as inspired by recent developments in distributed ledger technology. We discuss how properties of imperative and declarative languages including the underlying architectures to support contract management and lifecycle apply to various aspects of legal contracts. We then address these properties in the context of several blockchain architectures. While imperative languages are commonly used to implement smart contracts, we find that declarative languages provide more natural ways to deal with certain aspects of legal contracts and their automated management.

Keywords Legal contracts · Smart contracts · Blockchain

1 Introduction

The concept of a contract is used in business, commerce and everyday life to capture any agreement between parties, and to govern their interactions, either on an one-off basis or continuously over a time period. We shall speak of legal contacts to specifically denote agreements having legally binding effects.

✉ Regis Riveret
regis.riveret@data61.csiro.au

[1] Data61, CSIRO, Brisbane, Australia

[2] European University Institute, Florence, Italy

[3] Deontik, Brisbane, Australia

[4] Data61, CSIRO, Sydney, Australia

In the past, many commercial computer systems often labeled as ‘contract management applications’ have been developed to support automation of legal contracts. In this context, the term ‘e-contract’ has been used to refer to an electronic representation of a contract, suitable for contract automation activities, and there have been some standardization initiatives concerning e-contracts (OASIS 2007).

The term ‘smart contract’ was initially proposed in the early 90s for e-commerce applications (Szabo 1997) but has recently been widely used in the context of distributed ledger technologies and in particular blockchain technologies (Staples et al. 2017). In this context, a smart contract is any self-executing program running in the distributed ledger environment, and it is often meant to implement automated transactions agreed by the parties. The execution can be based on triggers provided by the users or extracted from the environment.

While not every smart contract has legal significance, many smart contracts are linked to legal contracts, namely with agreements meant to have legal effect. We may distinguish two cases. In the first case, a separate agreement, expressed in natural language, may exist between the parties, and the smart contract may be meant to implement automatically the content of that agreement. In this case the smart contract may provide evidence for the existence and the content of the agreement between the parties, while automating its execution (Staples et al. 2017). In the second case, when no other document exists recording the agreement of the party, the smart contract itself embodies the binding expression of that agreement. In this case, on which we shall focus on this paper, the smart contract itself is meant both to have certain legal effects and to implement them automatically.

Distributed ledger systems can support the implementation of smart contracts with regard to both storage and automated execution. Further, the availability of digital currencies enables the automated execution of money transfers, as needed to implement the contract. Thus, distributed ledger systems can go beyond mere distributed databases; they constitute computational platforms offering integrated services to run large numbers of smart contracts. As distributed ledger systems are operated by collectives, they may disrupt conventional organizations accommodating trusted third-parties. They open up new opportunities for automated agreements, leading to a wide range of useful applications, but also raising important legal issues.

Some programming languages for smart contracts may be more suitable than others to facilitate legal interactions. In that regard, while imperative languages are often used to code smart contracts, declarative languages may be interesting alternatives. Declarative smart contracts, and in particular logic-based smart contracts could provide advantages in representing smart contracts and reasoning upon them. For example, they can be more compact than their imperative counterparts, they can be easier to draft, their properties can be formally verified, parties can more easily understand the content of the contract and its implications.

These are some common arguments supporting the use of declarative languages to encode smart contracts, but there are also questions as to whether these arguments really hold in the context of distributed ledger.

Contribution This paper addresses the conceptual connection between legal contracts and smart contracts. It investigates the legal and technical issues relative to the use of smart contracts expected to have legal effects, and thus it contributes to the implementation and use of smart contracts as legally binding agreements. More specifically, we compare imperative and declarative smart contracts in order to understand their respective (dis)advantages, from a legal and technical perspective. The comparison is developed in relation to aspects such as legal validity, interpretation, and lifecycle of contracts. While computer-executable contracts are not a new matter, we reappraise the discourse in the context of distributed ledger technology. Our focus is on a particular distributed ledger technology, namely blockchain-based systems operated by a trusted collective. We also compare different architectures for leveraging logic-based languages, in order to assess their suitability for modeling contracts on blockchain platforms.

Outline This paper is organize as follows. In Sect. 2, we sketch out the fundamentals of contracts from a legal perspective. Section 3 introduces e-contracts, smart contracts and blockchain systems. In Sect. 4, we compare imperative and logic-based smart contracts in light of elements for legal validity, interpretative matters and common legal activities. Section 5 continues the comparison in the context of blockchain systems and investigate different options for the operation of smart contracts, before concluding.

2 Basics of contracts in the law

This section reviews fundamental and practical notions regarding contracts in the law. First, a short characterization of legal contracts is provided, and then the necessity for interpretation of legal contracts is discussed. Finally, the main events in a contract's lifecycle are distinguished.

2.1 Contracts—legally binding agreements

A contract is an agreement between two or more parties, meant to be legally binding and to effect a change in their legal positions. The law defines the conditions under which a contract is legally binding or legally valid (in this paper, legally binding and legally valid are used synonymously). Legal validity typically encompasses the following elements.

Agreement There must be an offer, seriously made by one party to another party, who must accept it seriously, clearly and without reservation, leading to an agreement. In addition, parties should make the agreement voluntarily, without restraint or undue influence, acting of their own free will. This offer and acceptance can only be formed by entities recognized by the law, such as natural persons or companies.

Consideration Usually, for a contract to be legally valid, both parties should have an interest in its implementation. Under common law, consideration is required, so that each party must give or promise something of value to the other.

For example, if an agent A signs a contract to buy a service from B for 6000, A's consideration is the 6000, and B's consideration is the service.

Competence and capacity Parties must have the competence and the capacity to make the decision to enter into a contract. A party has no competence or capacity to make the decision to enter into a contract if the party is unable, or is presumed to be unable, to understand the contract. For example, a person who is insane or below a certain age might not be deemed competent to make a contract; this person does not possess legal capacity.

Legal object and purpose The object and purpose of the contract must comply with the law. A contract cannot be enforced if the actions agreed upon violate the law or public morals. Most of the time, the relevant jurisdiction is the one where a contract was concluded or where it is supposed to be performed, or the juridiction specified in the contract.

If a contract includes all the above-mentioned elements and they are not defective (according to the law governing the contract), then it is legally valid, and it will bind the parties. If an element is missing or defective, then the contract may be *void* or *voidable*. A void contract has no effect (it does not have to be executed), while a voidable contract has effect (it can be executed) until it is terminated by a court. For example, an oral contract for the sale of a house is void, since the law requires a written form; a contract where a party was deceived by the other party (A sells to B a painting saying it is a van Gogh, knowing it is not) is voidable.

2.2 On legal interpretation

The parties entering a contract declare, by making the contract, that they intend to achieve certain legal arrangements between them. To the extent that the law recognizes these arrangements, they become legally binding for the parties. Therefore we can say that a contract is an institutional act, through which the parties declare certain legal arrangements concerning their relative positions and to which the law correlates, in principal, the realization of those very arrangements that the parties have declared (Gelati et al. 2004). In general, operational semantics of the contractual arrangements can be analyzed in relation to different elements, starting from its content.

Content The arrangements that the parties aim to achieve though a contract may be different. They may include

- The creation of obligations of one party towards the other (namely, the creation of a right to the performance of the latter party toward the first one). As an example of a contract creating obligations, consider a loan contract, according to which a person undertakes the obligation to provide a certain service (e.g., transfer a sum of money) toward the other, which may undertake a similar engagement (e.g., to give back that amount) toward the first party;
- The creation of liberties (permissions to do or omit). As an example of a contract creating a liberty, consider the case of a licence contract, according to which a

party authorizes another to download and use a certain piece of intellectual property (e.g., a movie). Without the authorization established by the contract, the latter party would be prohibited to download and reproduce the movie;
- The transfer of entitlements (such as in particular, property rights). As an example of a contract transferring an entitlement, consider a sale contract, e.g., John's sale of his old computer to Mary for 500 Euros. In this case Mary and John agree that the ownership of the computer is transferred from Mary to John, Mary undertakes the obligation to pay 500 to John and John to deliver the computer to Mary.
- Actions and penalties to be taken when the parties fail to comply with their obligations.

The contractual specifications of normative positions may be complemented by temporal specifications, for instance concerning the time from which or within which time-frame rights can be exercised and obligations have to be fulfilled.

Interpretation The parties to a contract may disagree on the meaning and application of the terms in their joint contractual declaration. When such a disagreement arises, an interpretation of the contract is required. To determine the binding meaning of a controversial term, various aspects may be considered: the literal sense of words used, the existing contractual practice, contextual clues pertaining to the previous interaction between the parties, etc. Different approaches are adopted by different legal systems, e.g., traditionally English law prioritizes the literal meaning, while continental laws give more importance to contextual clues. For instance in the famous case Smith v Hughes (1871), the English courts affirmed that the term "oat" in the contract would cover both old and green oat (which was delivered by the seller), even though the context indicated that the buyer intended to buy old oat only, and the seller was aware of that.

Open-textured terms Contracts in natural language often require compliance with open-textured standards (best effort, reasonable request, due diligence, without delay, etc.). In many cases, the significance of such standards can only be understood with reference to the socio-legal context in which the contract was formed and is to be implemented. Thus, the determination of what is needed to meet such standards may involve the assessment of complex factual circumstances, according to general practice and the interaction of the parties. As an example, consider a clause specifying that parties should each undertake their 'best efforts' to achieve some shared goal during the life of a contract. What this clause requires may vary depending on the tasks of each party, the resources available to them, and the stage at such tasks have to be accomplished, given the previous history of the interactions between the parties.

Implied terms A contract does not produce only the legal arrangements that the parties have declared, it also produces the arrangements that the law itself, or existing customary rules connect to that particular kind of contract. For example, a sale contract does not only entail the agreed transfer of property of the sold item and the seller's obligation to deliver in conjunction with the buyer's obligation to pay the agreed price. It also includes a seller's warranty that the item has the qualities that make it fit for its intended use and the buyer's power to rescind the contract in case the item does not have such qualities, as well as the seller's obligation to cover damages resulting from the defects of the sold item. Similarly, a contract may not produce all of the arrangements that the parties intended, since the law may establish the nullity of certain contractual clauses, such as clauses that exclude liability for recklessness.

Integration The contractual declaration may need to be integrated when some aspects were not addressed by the parties. For instance, a sale contract may not indicate the place and time of delivery, or even the price of the good. In this case the law may establish default terms, or it may provide judges with some more or less determined criteria to fill the gap (e.g., a reference to contractual usage, to equity or reasonableness). In a broader sense, a contract is susceptible to be 'incomplete', if it does not account for circumstances such that, had the parties foreseen them, they would have adopted additional or different terms (e.g., a contract for the sale of certain items to a company does not consider the possibility that the company's demand goes up or down to a considerable extent). In such cases the contract may need to be renegotiated.

Since the binding outcome of a contract also includes the interpretation of vague or ambiguous clauses (in particular from courts), implied terms and integrations, the mere wording of a contract may be insufficient to identify the resulting contractual arrangements.

2.3 Lifecycle of a contract

A contract goes through a series of possible stages, as presented below, from its formation to its execution or termination. This series of typical stages can be described as the 'lifecycle' of a contract.

Negotiation and formation Based on the 'freedom to contract', any legal entities are in principle free to form contracts with any content. A contract is usually concluded though the combination of an offer and an acceptance, by the proposal of one party to conclude a contract having certain terms, and the acceptance of those terms by the other party. Under such conditions, it is presumed that the parties have achieved a 'meeting of the minds', i.e., that both have understood the terms in the same way and agreed to them, and they share the intention to enter the contract accordingly. Before reaching an agreement, parties often negotiate the terms, making offers and counteroffers. Freedom of contract is limited by rules that prohibit contracts with certain content, or prescribe a certain form (e.g., in writing) for certain kind of contracts.

Contract storage and notarizing A contract can be formed in many different ways, such as by oral agreement, hand shake or intangible agreement, unless a specific formality (e.g., writing) is required by the law. Yet, a non-written contract may be difficult to prove: there may be no witnesses, or they may be unreliable or the law itself may require a written proof. Therefore, in many cases it helps to have a written record of what was agreed. Additional certainty can be provided through certification by a trusted third party, such as a notary. In legal contracts, usually the content is expressed in natural language, as in the example above, and a date is manually inserted.

Performance Once a contract has been formed, it has to be executed; the parties have to take appropriate actions to fulfil the contractual clauses. Parties can directly perform the appropriate actions, or they can delegate their performance to others (as long as it is permitted by the contractual terms).

Monitoring and (private) enforcement Each party will check whether the appropriate actions are taken by the other party. Private enforcement is the activity meant to induce the other parties to perform their required actions. It may involve requests, encouragements, or threats to take countermeasures, including recourse to judicial enforcement.

If one party does not fulfil her obligations and appropriate circumstance exist, other parties may withdraw from the agreement and thus retract from performing their activities. Depending on the circumstances, the innocent party may seek remedies such as compensatory or punitive damages, or even the modification or resolution (termination) of the contract. These remedies can be ordered by a court, or under certain conditions, be triggered directly by the innocent party.

Modification The parties can always agree to renegotiate the contract and modify its content. Under certain conditions, a party may have a right to withdraw, unless the other party accepts a change to the terms of the contract (e.g., when compliance has become excessively burdensome, due to unforeseeable exceptional circumstances, or when the contract was made under duress). The contract may be modified or resolved also when the performance by one party has become partially or totally impossible.

Dispute resolution A contract is susceptible to disputes, concerning the validity of the contract, the interpretation or integration of its terms, or breaches of its obligations, or ways to deal with unexpected circumstances. Two major types of dispute resolution exist: (1) adjudicative resolution, such as litigation or arbitration, where a judge, jury or arbitrator determines the outcome, and (2) consensual resolution, such as mediation, conciliation, or negotiation, where the parties attempt to reach mutual agreement.

Termination A contract is terminated when all contractual obligations are performed, when the parties parties agree to cancel the contract, or when the contract is voided or resolved.

3 Automation of contracts

There have been significant efforts over the last decades to support computer-based automation of legal contracts, both as commercial products (see e.g., Aberdeen Group 2005; Montgomery and Wilson 2015) and research proposals (see e.g., Daskalopulu and Sergot 1997; Grosof et al. 1999; Molina-Jimenez et al. 2004; Marjanovic and Milosevic 2001; Linington et al. 2004). This section outlines the main approaches to contracts automation, from e-contracts to smart contracts on blockchain systems.

3.1 E-contracts

The term ‘e-contract’ is often employed as an umbrella term covering any contracts that can be handled or processed by computer systems e.g., through exchanges of e-mails or through websites. In a more specific sense, the term e-contract is used for representations of contracts that include computable parts, such as data fields, rules and similar. Such computable parts are then intended for subsequent contract automation operations, such as drafting, negotiation, monitoring and enforcing.

There have been a range of commercial contract management systems developed to support better management of e-contracts, including the commitments of parties to contracts. These systems address different aspects of contract automation. Some focus on document management functions, while others focus on compliance. The latter functionality, often referred to as commitment management, is used, for example, in supply contracts, service level agreements, banking and insurance contracts. Commitment management systems link contract terms to transactions and processes, to increase efficiency and monitor the actions of the parties. These systems may address various aspects of contract automation, with functions having different levels of sophistication, associated with different stages in the contract lifecycle (Milosevic 1996). Typically, these systems are deployed by one party, traditionally within their internal systems. Recently they are also offered by third parties, using the model of ‘Software as a Service’.

Existing contract management systems have various levels of representation for e-contracts: sometimes an explicit expression of contractual rules is provided; more often contractual rules are translated into computer procedures across different enterprise systems, possibly leading to automated contracts, such as smart contracts.

3.2 Smart contracts

The term ‘smart contract’ was originally introduced in the 90’s by Szabo (1997), stemming from the idea that a technological legal framework would help commerce, reduce costs and disputes. The idea of a smart contracts emerged at a time where some systems and standards for entering into contracts were already well-established, such as electronic data interchange (EDI) for the communication of structured data between the computers of trading partners. These systems and standards focused on the message structures and protocols in support of commercial transactions. They addressed only a limited number of possible contractual transactions, resulting in limited expressiveness compared to the possibilities which are offered today by smart contracts.

As we mentioned in Sect. 1, the term ‘smart contract’ is often used to mean any computable specification of agreed transactions between parties. There are smart contracts which are, as often remarked, “neither smart nor contracts” in the legal sense. Nevertheless, there are smart contracts that are legally binding, since their code is the only valid expression of a contractual agreement between the parties.

Thus, a smart contract having legal effect is a computer program that both performs certain operations and expresses, by specifying such operations, the intention to create the legal results (obligations and entitlements) that are presupposed by such an operation. For instance, consider a smart contract that is labeled as a loan, specifying the operations consisting in transferring \$100 from A to B, at time t, or transferring 1 unit from B to A each month after t until $t + 1$ year, and to transfer 100 from B to A, at $t + 1$ year. Here the specification expresses that A and B by agreeing on the smart contract have made indeed a legally binding loan contract, involving A's obligation to transfer the money t and B's obligation to pay back the loan plus interest to A.

While smart contract having legal effect are 'real' legal contracts, it is important to remark that important differences exist between the content of such a smart contract and the content of a corresponding contract expressed in natural language. On the one hand, parts of the smart contract may have no equivalent terms in the natural language contract. For instance, some terms in the smart contract may be concerned with details that are left implicit in the natural language formulation. On the other hand there may be clauses in the natural language contract that are not included in the smart contract, since their automation is not necessary, possible or even desirable. For instance, a clause including indeterminate standards (reasonable case, undue delay, etc.) may not be meaningful in the context of a smart contract.

In general, the code of a smart contract refers to contractual clauses which can take the form of some conditional logic, where both antecedent conditions and resulting effect are precisely specified. Thus, a contractual clause that attaches certain legal consequence to a specific condition is expressed as a piece of code that checks for the conditions and triggers changes (e.g., the transfer of a sum of money at a certain time) that correspond to the legal effect (e.g., an obligation to pay that sum by that time).

3.3 Blockchain smart contracts

Smart contracts have gained prominence due to their use on distributed ledger infrastructures based on blockchain technologies, which have overcome the need for trusted third parties to implement and take responsibility for automated transactions. When a smart contract is executed on a centralized system run by intermediary agents, contracting parties have to trust and deal with these agents. On the contrary, when smart contracts are run on a distributed ledger, the execution and recording of transactions are provided by a cryptographically enabled decentralized infrastructure. In this paper, the focus is on a particular distributed ledger technology, namely blockchain systems.

The most prominent example of a blockchain system is the platform for digital currency Bitcoin (Nakamoto 2008), which established a peer-to-peer network of accounts and transactions (a ledger). Since Bitcoin is the most prominent example, our description of the functioning of blockchain is based on this system (as implemented in Feb. 2018); the functioning of different blockchains is likely to differ in detail that fall outside of the scope of this paper.

A blockchain system consists of a network of geographically distributed computing nodes, sharing a common append-only data structure (the blockchain) to record blocks of transactions, where revisions or tampering are made prohibitively difficult due to the modus operandi of the infrastructure: consensus amongst the nodes about the state of the data structure determines its content and evolution, and modifying past nodes is prohibitively expensive.

The data structure underpinning a blockchain system is distributed because it is replicated amongst the processing nodes of the system. As new blocks of recent transactions are added to the distributed data structure, they include a hash reference back to the previous blocks, so that any node can consequentially verify the integrity of the data structure. This chain of blocks of transactions is called a blockchain.

In Bitcoin blockchain, a block also includes an answer to a computational problem which is (adjustably) hard to solve, but easy to verify. A block cannot be attached to the blockchain without the correct answer. Every computational problem is solved by ‘miners’ which compete to be the first to find the answer for the current block.

It is possible for the blockchain to have temporary forks, for example, when two miners find two different solutions for the same block at the same time. The peer-to-peer network resolves these forks by considering the ‘longest’ and the most difficult to solve chain of blocks as valid. Consequently, it is difficult for someone to fork from the chain by creating a ‘longer’ more difficult chain, and having it accepted by the network as the ‘longest’. This mechanism prevents tampering or revision of the (Bitcoin) blockchain.

Transactions on the blockchain are not cost-free due to the economic consensus system. Miners have to spend computing power (tied to hardware) and energy to integrate blocks of transactions into the blockchain. When a miner successfully adds a block, having solved the corresponding computational problem, the miner receives a reward, in the form of new coins (block reward) and transaction fees.

Transaction fees are an incentive for a miner to include this transactions in their block. For advanced blockchain systems, the fee *may* also cover the cost of the computational steps required to operate a transaction, in particular when the transactions are associated with extra instructions. The computation of the amount of the fee is outside of the scope of this paper, but as rule of thumb, the simpler a transaction in terms of computational complexity and the smaller in terms of bytes, the less it costs.

Since transactions can be costly, it is often the practice that heavy computation should be performed ‘off-chain’ instead of ‘on-chain’. In off-chain scenarios, computation is performed outside the blockchain-based system, e.g., on the server of an intermediation service, while in on-chain scenarios, computation is performed and validated in the blockchain-based system by the miners. Off-chain computation results can be recorded in a blockchain, however parties may prefer to avoid off-chain intermediation services that can be performed on-chain, for example to increase trust.

Public blockchains can be accessed by anyone on the Internet without permission. Consortium and private blockchains, on the contrary require permission to join. One can deploy the same technology of a public blockchain, restricting access to authorized nodes. Using a public blockchain results in better transparency and auditability, but sacrifices performance and has a different cost model. In a public blockchain, data privacy and security relies on encryption or cryptographic hashes. A consortium blockchain is used across multiple organizations. The consensus process in a consortium blockchain is controlled by pre-authorized nodes. The right to read the blockchain may be public or may be restricted to specific participants. In a private blockchain network, write permission is usually kept within one organization, but this may include multiple divisions of a single organization, and the blockchain can still be fully distributed.

Blockchain technology, initially used as a distributed ledger for payment in digital currencies (primarily Bitcoin transactions), can also be used to manage other transactions—e.g., escrow, notarization, registration, process coordination or voting. These transactions are currently managed by trusted third-parties, such as banks, governmental agencies, legal firms or other specific service providers. Instead of relying on the third-parties, contracting agents may implement such transactions as smart contracts on a blockchain-based system.

The storage and execution of smart contracts in a blockchain-based system can be achieved in many different ways, see e.g., Ethereum Foundation (2016), Canesin et al. (2018), Hess et al. (2017), Nxt (2018), Jelurida (2017). In the platform Ethereum (Ethereum Foundation 2016; Wood 2014), for example, a smart contract takes the form of script. The script is compiled into bytecode, and executed in a virtual machine, it is also stored in the Ethereum blockchain.

The script can be triggered by messages or transactions, resulting into its execution, and the triggered operations (except reading, which is executed on the local copy of blockchain) are executed on every node of the network. As a result, all nodes will reflect the state changes resulting from executing the operations. This replicated execution, which is not particularly efficient, has a cost. To cover this cost, the smart contract can be charged with some amount of resources, so called ‘gas’ in Ethereum. When the operations are executed, the gas is gradually depleted to pay the executions. If the operations cannot be paid anymore, then they halt, otherwise they continue, possibly leading to smart contracts which can be difficult to stop. A smart contract may be also ‘destroyed’, for example Solidity smart contracts in Ethereum have an operation, *selfdestruct*, which can be triggered to destroy the contract.

When smart contracts interact with each other, some sorts of techno-social structures can emerge, possibly hardly stoppable and without any central control. These aspects of smart contracts—given the immense range of smart contracts’ applications, from the most useful to the most risky—raise serious issues. On the one hand it allows for large unpredictable financial consequences to be generated, and on the other hand it makes it difficult to implement legal remedies that reverse the outcome of illegal transactions. In the remainder of this paper, we investigate smart contracts in the eyes of the law, and with respect to two programming paradigms for smart contracts: imperative and declarative languages.

4 On imperative and declarative smart contracts

Most smart contract systems today adopt an imperative approach. According to this approach the smart contract directly states the computational operations to be performed to implement the contract. In research on AI & law, it has often been argued that a declarative approach to modeling normative knowledge is preferable. The latter approach should state the legal arrangements that the parties have agreed, abstracting from the computations that are needed to implement them. This should allow for a more compact representation, that is closer to natural language and to human understanding.

In this section, we compare imperative and declarative approaches in light of their capabilities to support the creation of valid smart contacts having legal effect, and to sustain the stages in the lifecycle of such contracts

4.1 Imperative and declarative smart contracts

Smart contracts are essentially computer programs, and thus they can be programmed with different languages falling in the family of imperative (also sometimes called procedural) or declarative languages. To exemplify our comparative analysis, we will consider a contract in natural language (based on Governatori 2015).

Example 1 License for the evaluation of a product

Article 1. The Licensor grants the Licensee a license to evaluate the Product.

Article 2. The Licensee must not publish the results of the evaluation of the Product without the approval of the Licensor; the approval must be obtained before the publication. If the Licensee publishes results of the evaluation of the Product without approval from the Licensor, the Licensee has 24 h to remove the material.

Article 3. The Licensee must not publish comments on the evaluation of the Product, unless the Licensee is permitted to publish the results of the evaluation.

Article 4. If the Licensee is commissioned to perform an independent evaluation of the Product, then the Licensee has the obligation to publish the evaluation results.

Article 5. This license terminates automatically if the Licensee breaches this Agreement.

Suppose that the licensee evaluates the product and publishes on her website the results of the evaluation without having received an authorization from the licensor. Suppose also that the licensee realizes that she was not allowed to publish the results of the evaluation, and removes the published results from their website within 24 h from the publication. Is the licensee still able to legally use the product? Since the contract contains a remedial clause (removal within 24 h remedies unauthorized publication), it is likely that the license to use the product still holds.

Suppose now, that the licensee, right after publishing the results, posted a tweet about the evaluation of the product and that the tweet counts as commenting on the evaluation. In this case, we have a violation of Article 3, since, even if the results were published, according to Article 2 the publication was not permitted. Thus, she is no longer able to legally use the product under the term of the license.

The final situation we want to analyze is when the publication and the tweet actions take place after the licensee obtained permission for publication. In this case, the licensee has the permission to publish the result and thus they were free to post the tweet. Accordingly, she can continue to use the product under the terms of the license. □

4.1.1 Imperative smart contracts

When programming in an imperative language, the programmer writes an explicit sequences of steps to be executed to produce the intended result. The programmer has to write *what* has to be done and *how* to perform it. Imperative smart contracts are typically programmed in a procedural or an object-oriented language.

Example 2 Algorithm 1 gives a simplified code (in the programming language Solidity) of how a procedural smart contract can implement the contractual clause of Example 1. The smart contract includes a sequence of instructions updating the normative states (obligations, prohibitions and permissions in force) depending on what actions have been performed and the state of the program at a given time. The program has to set the initial state for the contract, then the procedure EVALUATELICENSECONTRACT has to be invoked every time there is a trigger for the program. Due to the lack of space, the code is simplified. For example empty variables should trigger some exceptions, and some functions/methods (such as the constructor, settler and getter functions) are omitted; temporal constraints (such as those stemming from the deadline) are also omitted. It is assumed that the contract is for one licensee, and for one work. □

As Example 2 illustrates, in imperative smart contracts, the order of instructions does not reflect the natural order of the contract clauses expressed in natural language. The programmer has to come up with such an order, has to manually determine how a trigger changes the state of the normative provisions (i.e., obligations, permissions and prohibitions) and has to propagate the changes according to their meaning. This means that the programmer is responsible for anticipating the legal reasoning implied by the contract clauses. For example, when a permission becomes true, the corresponding prohibition should be set to false; when an obligation is set to true, the corresponding permission should be set to true as well.

The process of writing a procedural program corresponding to a contract can be cumbersome and error prone since the order of instructions affects the correctness of the resulting smart contract. A possible solution to alleviate this problem is to create a state machine for the contract (Fig. 1 shows a state machine for the contract in Example 1). Then, the programmer can use the state machine as a guide to derive the procedural code. Alternatively, the state machine could be represented directly in the program and a state machine engine could then be used to execute the resulting smart contract. This approach requires to take into account a number of states and transitions that becomes exponentially large for non-trivial contracts.

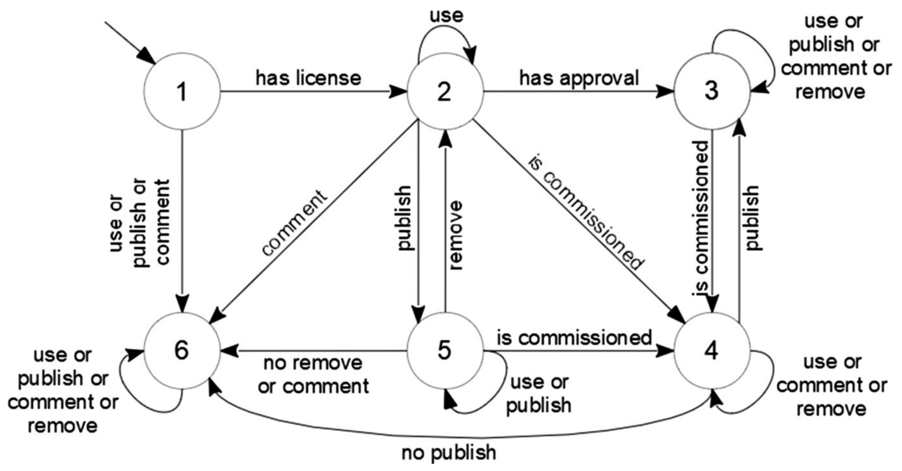

Fig. 1 Licensing contractual clauses as a state machine

In addition, if a contract involves some temporal knowledge (such as deadlines), then the intricate imperative approach may lead to further difficulties in terms of coding, verification and intelligibility. The code in Fig. 2 does not cater for the deadline in Example 1, and a code including it would certainly appear more complicated.

Eventually, an imperative smart contract has to specify the programming language in which it is coded. Given the possibility of the evolution of the programming language, the contract has to contain information about the version of the language (and eventually provide information or configuration for the computing infrastructure in which it is supposed to be executed). Without this, the interpretation of the contact is ambiguous, given that the execution of the code could be different on different machines with different versions of the execution framework required by the programming language.

4.1.2 Declarative smart contracts

Declarative languages are alternatives to imperative languages. Endorsing Kowalski's equation 'algorithm = logic + control' (Kowalski 1979), conventional algorithms can be analyzed in terms of two components: a logic component specifying what is to be done and a control component determining how it is to be done. The logic component is meant to express the knowledge which can be used in an algorithm, while the control component only affects its efficiency. As a consequence, when programming in a declarative language, ideally, the programmer does not have to write explicitly the sequence of steps to specify what has to be done. The programmer only describes what has to be done, without specifying how to do it.

Declarative smart contracts can be written in different declarative languages, such as functional languages and logic-based languages. Whilst a functional

```
pragma solidity ^0.4.19;

contract license {
  address author;              address licensee;
  bytes32 work_hash;           string name;
  bool hasLicense;
  bool use;                    bool perm_use;
  bool forb_use;
  bool publish;                bool perm_publish;
  bool forb_publish;           bool obl_publish;
  bool comment;                bool perm_comment;
  bool forb_comment;
  bool hasApproval;
  bool isCommissioned;
  bool remove;                 bool obl_remove;
  bool violation;

. // Constructor of the contract.
. // Relevant setter and getter functions.
. // Relevant ‘actuator’ functions.

function evaluateLicenseContract() public returns (int) {
  if(hasLicense){
        forb_use = false;
        perm_use = true; }                               // Art 1

  if(hasLicense && (hasApproval || isCommissioned)){
        forb_publish = false;
        perm_publish = true; }                           // Art 2, 4

  if(hasLicense && !hasApproval &&
    !isCommissioned && publish){
        obl_remove = true; }                             // Art 2

  if(perm_publish){
        forb_comment = false;
        perm_comment = true; }                           // Art 3

  if(hasLicense && isCommissioned){
        forb_publish = false;
        perm_publish = true;
        obl_publish = true; }                            // Art 4

  if(forb_use && use ||
     forb_publish && publish ||
     obl_publish && !publish && !remove ||
     forb_comment && comment ||
     obl_remove && !remove) {
        violation = true; }

  if(violation){
        forb_use = true;
        forb_publish = true;
        forb_comment = true;
        perm_use = false;
        perm_publish = false;
        perm_comment = false;
        obl_publish = false; }                          // Art 5
} }
```

Fig. 2 Sketch of an imperative smart contract in solidity

approach can be effectively used to program smart contracts (see e.g., Jones and Eber 2003), the logic-based approach has provided a more popular framework to represent and reason upon normative knowledge (see e.g., Prakken and Sartor 2015 and Gabbay et al. 2013 for overviews). In this paper, we focus thus on logic-based languages to represent and reason upon contractual clauses.

When employing the logic approach, the terms of a smart contract are written in a logic-based language (possibly relying on substantial research endeavours on deontic logics and normative systems (Prakken and Sartor 2015; Gabbay et al. 2013). Statements of this language represent contractual clauses, and inferences are applied to these statements to generate further statements, eventually leading to some actions meant to be performed.

Example 3 Logic-based models of defeasible reasoning are often advocated to capture legal knowledge and reasoning, so that conclusions entailed from some particular knowledge can be revised in light of new pieces of information (see e.g., Sartor 2005). Let us consider for instance the representation of the contract given in Example 1 provided by the (deontic) defeasible logic (Formal Contract Logic, FCL) of Governatori. (2005) and Governatori and Milosevic (2005, 2006). The clauses can be represented by the following rules and preferences:

```
Art1.0: => [Forb_licensee] use
Art1.1: hasLicense => [Perm_licensee] use
Art2.1: => [Forb_licensee] publish [Compensated] [Obl_licensee] remove
Art2.2: hasLicense, hasApproval => [Perm_licensee] publish
Art3.1: => [Forb_licensee] comment
Art3.2: [Perm_licensee] publish => [Perm_licensee] comment
Art4.0: hasLicense, isCommissioned => [Obl_licensee] publish
Art5.1: violation => [Forb_licensee] use
Art5.2: violation => [Forb_licensee] publish

% Superiority relation
Art1.1 > Art1.0,
Art2.2 > Art2.1, Art3.2 > Art3.1,
Art5.1 > Art1.1, Art5.2 > Art4.0.
```

The rules are meant to be deployed in conjunction with a rule engine evaluating them, for example the defeasible logic engine SPINdle (Lam and Governatori 2009). If no triggers are activated, then `Art1.0`, `Art2.1` and `Art3.1` fire and we conclude the prohibitions of `use`, `publish` and `comment`. If on the other hand `hasLicense` and `publish` hold, then `Art1.1` fires and overrides `Art1.0` thus we have the permission of `use`, but we continue to have the prohibition to `publish`. Therefore the publication contravenes Article 2, and we can use rule `Art2.1` to derive the mandated compensation, that is the obligation of removing the material is now in force and we conclude `[Obl_licensee] remove`. Rules `Art5.1` and `Art5.2` specify that if a violation occurs then we conclude the prohibitions of `use` and `publish`, and also `comment` because, in this case, the permission to `use` is discarded and `Art3.2` is not applicable. See Governatori. (2005) for details of FCL. □

A compelling advantage of the logic approach is that programs can be seen as a set of specifications, and thus the programs are guaranteed to be correct with respect to the specifications. A logic program is correct if it is sound and complete. Sound means that any result in an execution is valid with respect to the specification. Complete means that any valid result can be reached by the program. As correctness is guaranteed, the logic approach is then meant to facilitate the verification of the programs. Verification can be also facilitated because declarative specifications are usually shorter than the corresponding imperative programs, and easier to understand.

If a contract includes some temporal knowledge, then one can take advantage of a temporal logic (see e.g., Farell et al. 2005 or Governatori and Rotolo 2013 amongst other works) where temporal reasoning is comprised within the logic at the inferential level, facilitating thus the programming task through clear temporal specifications.

In general, given a set of statements, inferences can be performed in different manners depending on the semantics of the logic to be used. Typically, a declarative smart contract would specify the semantics to be employed, so that an inference system can reason upon the contract and execute it. Of course, it must be ensured that the implemented inference system is correct with respect to the semantics of the language. Different reasoning mechanisms for the same semantics can exist, and in most cases, it is preferable to use the algorithm with the lowest computational complexity, but one may prefer other algorithms for some reasons, e.g., to present more 'natural' or intelligible inferences and explanations to end-users.

4.2 Comparing imperative and declarative smart contracts

The remainder of this section describes a comparative analysis of imperative and declarative languages in terms of their suitability to express different properties of legal contracts, as introduced earlier.

4.2.1 On legal validity

To understand how the two paradigms may affect the legal validity of a smart contract, the four elements identified in Sect. 2 (agreement, consideration, competence, legal purpose) are considered.

Agreement A valid agreement implies that the terms of the contract should be stated in a language that is understandable to the parties agreeing to the contract. If an imperative contract is so convoluted and complicated that a party cannot make sense of it, it may fail to constitute a valid agreement. On the contrary, a declarative contract may more easily pass the understandability test, being simpler and closer (more isomorphic) to natural language.

Consideration Both imperative and declarative smart contracts can explicitly specify the consideration of a given contract. Beyond the mere explicit representation of consideration, there is a lot of case law regarding whether something is consideration or not. Thus, some sort of 'consideration checking system' could be devised to reason upon consideration statements to determine whether such consideration is valid. A declarative approach might ease the verification of consideration statements, for example by benefiting from conceptual interoperability between declarative systems using Semantic Web resources such as ontologies (Berners-Lee et al. 2001).

Competence and capacity The competence and capacity of parties are typically evaluated through identification services. In business, organizational policies usually define the capacity of parties to do business, and these may also involve the use of role-based access control, as for example discussed in Milosevic et al. (1996) to restrict system access to authorized users and thus to enter into a contract. Here, 'rule-based access controls' are sometimes advanced to provide a more flexible and granular control (see e.g., Kern and Walhorn 2005). While access controls can be used to ensure the competence of contracting parties, such systems are usually separated from the proper representation of a contract and reasoning upon it.

Legal object and purpose The legality of a smart contract can be established by analyzing conditions in the contract and checking their compliance against the rules of a specific jurisdiction, which in turn can be captured and stored in a separate architecture component such as a legal rules repository. This is facilitated if both the contract and the legal rules have a logical representation.

On the basis of the considerations we have developed with regard to specific aspects of a contract, we can conclude that declarative languages can be more convenient than imperative languages to address fundamental elements affecting the validity of smart contracts.

It is worth pointing out that, while the benefits offered by a programming paradigm shall be appreciated by the technical and legal parties concerned by these elements, end-users may actually ignore such benefits. For example, a layman would have little concerns on whether their smart contracts have a clearer implementation. The reason is that an agreement about what a contract should perform is usually defined in a natural language, and then a smart contract is instantiated and managed through some templates, see e.g., Smart Contract Templates (Clack et al. 2016a, b). Similarly as end-users have usually no interests in or are ignorant of the programming languages of their computer applications, one may question why end-user parties should consider the translated contract if they trust the translation. In general, benefits of a language paradigm are thus meant to be appreciated by technical and legal practitioners, so that they can ensure (or not) end-users about legal elements of their smart contracts.

4.2.2 On legal interpretation

A smart contract is a Janus-faced entity: one face is a set of agreed legal arrangements, the other face is the executable instructions implementing these arrangements. For instance, a smart contract for a loan creates the lender's commitment to provide the money and the borrower's commitment to pay the instalments, while the executable instructions perform the transfers needed to implement these commitments. A smart contract for the sale of stock will provide for the transfer to the stock and the payment of the money. Hence the implementation of a smart contract implies the full understanding of legal transitions that the contract is meant to implement.

Let us consider what kinds of doubts the parties may raise concerning the meaning of a term in a contract, so that interpretation is needed, considering that the terms of the contract are constituted by programming instructions, and that the commitments of the parties have to be inferred (abducted) from such instructions.

Firstly, an interpretation issue may arise concerning what kind of commitments a party undertook by agreeing on the instructions of the contract. For instance, the parties may debate whether the instructions transferring a certain amount of stock from one party to the other versus a modest payments are meant to implement a loan of the stock (so that the receiving party has the obligation to return the stock back) or rather the sale of it (so that no obligation to return is implied).

Secondly, another interpretation issue concerns the implementation of the instructions in case they are differently interpreted in relation to diverse computational environments (e.g., compilers or interpreters for different versions of the language in which the instructions are written). The issue would then concern in what environment the parties intended, or should have reasonably intended, the contract to be executed.

Thirdly, an interpretation issue can emerge when the outcome that a party intended to obtain through the contract is different from the outcome that is delivered by the execution of the contract (e.g., in a loan contract the instalments transferred are higher than expected). We may however doubt whether these cases really involve an interpretation issue—should the contract be understood in its literal sense, i.e., in the operational semantics of its terms, or rather according to what appears to be the intentions of the party, as it was recognizable to the counterparty. We may also consider this case as involving not an interpretation issue, but rather a party's mistake in understanding the non-ambiguous, but complex terms of the contract, to which the party has agreed. Thus the legal issue would be whether this mistake entitles the concerned party to ask for the annulment or the modification of the contract.

Example 4 Let us consider for instance a loan smart contract, and let us assume that they are implemented in an opportune imperative language. The contract consists in a set of instructions:

– at time 1.1.2016 transfer 50 tokens from John to Mary!
– every month, from 1.2.2016 until 1.1.2017, transfer 2 tokens from Mary to John!
– at time 1.1.2017 transfer 50 tokens from Mary to John!

One of the party may not have fully understood what transitions were implied by the program set up by the other party (e.g., it did not understand that compound interests were established): should the party be bound by a term she did not understand? Should we construct the content of the contract as including that term as well or should it rather be considered as an unlawful unilateral imposition? The smart contract may implement illegal transitions. For instance, the interests established could exceed the legal maximum. The smart contract may fail to implement some of the legal effect associated with that kind of contract. For instance, it may not provide for a parties' power to stop her performance in case the other party does not comply. □

As observed in Sect. 2.2, the interpretation of many open-textured terms is context dependent. Different mechanisms can be defined to determine the meaning of such context-dependent interpretations, i.e., to determine what they require in particular contractual interactions. In some cases learning mechanisms could be defined (i.e., looking at past cases, and learn what behaviours were classified as the open texture terms, or, again for a reoccurring conditions, looking at past instances and determine acceptable conditions). Such mechanisms could in principle be implemented in a smart contract, or in the corresponding programming environment, though this would go beyond the state of the art.

To sum up, understanding a smart contract, and assessing its legal validity and effects, require determining what legal transitions the contract is meant to implement. The set of legal transactions and the executable instructions need to be kept aligned. One may argue that the practice of expressing smart contracts through imperative languages can introduce a vast gap between legal transactions and executable instructions, which determines uncertainties and may cause litigation and costs. This makes it interesting to adopt a different modeling strategy, namely, the use of declarative representations which minimize the distance between the specification of the intended legal effects and their executable implementation.

4.2.3 On lifecycle

Let us now compare imperative and declarative smart contracts in light of the different stages of legal contracts.

Negotiation and formation As any other contract, also a smart contract can be the outcome of a negotiation which takes place before the contract is deployed in machines (see e.g., Hanson and Milosevic 2003). Often the negotiation concerns a text in natural language (as in the case of the creation using a template), which is then translated into a smart contract. However, a smart contract program can be created without a natural language counterpart. In any case the type of programming language used in the final smart contract may affect the negotiation and the formation of the contract.

Using imperative languages, fairly sophisticated smart contracts can be formed already, but imperative coding may appear difficult to apprehend, slowing down negotiation and formation. Moreover, the parties may continue to wonder whether the contractual clauses have been properly coded. Of course, an imperative code can be 'validated' (unit testing etc.) to determine whether it is fit for use, but testing is time consuming and error prone.

On the other hand, logic-based smart contracts constitute executable specifications, which are often more comprehensible for humans and which can be directly executed, thereby decreasing the risks of errors in implementation. This does not imply that declarative smart contracts do not need to be tested; yet the validation of logic representation can be eased by using logic-based techniques, such as formal verification, to detect whether certain properties hold, in particular consistency (see e.g., Fenech et al. 2009). Furthermore, a logic representation may facilitate the negotiation and formation of contracts by artificial agents, since such agents can use artificial intelligence to reason upon contractual terms expressed in a logical form (see e.g., Governatori and Pham 2009).

Contract storage and notarizing Whatever the programming language, contract storage can be straightforwardly related to the storage of smart contracts, e.g., using file systems. One of the potential advantages of these systems is that any contract or data entered can be cryptographically signed as part of the normal functioning of the system. Thus, as long as the electronic signature is recognized, some part of notarizing and attestation is performed in an automated way. In this regard, no special advantages seem to be provided by procedural or declarative approaches.

Execution The execution of a smart contract requires the careful review of non-functional properties, such as those concerning performance (throughput and latency) and costs. These properties depend much on the computational environment in which is run a smart contract. At the level of the programming paradigm, important properties are computational complexity and interoperability.

– *Computational complexity* The efficient execution of a smart contract is a necessary condition for the use of such a contract, in particular when considering worst-case scenarios. While the computational complexity of the execution of an imperative smart contract depends much on the programmers' attention to cater for such property, the complexity of a logic-based smart contract relies on the complexity of the underlying inference mechanisms (we further analyze this point in the next section). Besides, given an arbitrary computer program (written in a Turing-complete language, and thus possibly including loops) as an input, it may be tricky to decide whether the program will finish running or continue to run forever (halting problem). For this reason, a logic-based language is interesting when its inference mechanism can mitigate such an issue, e.g., by ignoring (unintentional) infinite sequences and thus ensuring termination.
– *Interoperability* The execution of a contract is not necessarily meant to occur in isolation. On the contrary, contractual clauses have to be considered with respect to exogenous (legal) information, such as rules from other contracts or the embedding normative environments (the law in a particular jurisdiction). While imperative smart contracts can interact with each other rudimentary, a logicapproach can rely on rule interchange languages, e.g., LegalRuleML (Athan et al. 2015), to express rules and support interoperability amongst the contracts and other rule systems.

Hence, declarative smart contracts may offer interesting advantages compared to imperative counterparts when considering computational complexity and interoperability.

Monitoring and enforcement Since smart contracts are automatically executed, monitoring their execution and enforcing them may appear superfluous. However, the execution of a smart program can be interfered with, e.g., by hacking the contract or executing additional operations not foreseen by the contract. Different monitoring and enforcement solutions can be proposed: some checks can be simply included in the code of a procedural contract to monitor its execution, more sophisticated integration with architectures of specific information systems can be devised, e.g., for business contract architectures (Dimitrakos et al. 2003), while general formal run-time compliance techniques are also available to check the compliance of the execution of logic contracts (see e.g., Lomuscio et al. 2011).

Modification With imperative code, repeated modifications by several people may lead to an unordered combination of possibly interfering instructions (the so-called spaghetti code), in particular in case of (a network of) large contracts. Well-structured imperative code may mitigate the issue, but programmers have to spend some efforts to do so. Order and consistency can be more easily maintained in logic contracts, which are shorter and have a clearer meaning.

Dispute resolution The language paradigm used in the contract can affect the emergence and the resolution of disputes. The difficulty to understand an imperative code, can lead to disagreements concerning the meaning of the contract. Moreover the difficulty to understand procedural programs can also hinder the understanding of differences and commonalities between the issues at stake in different cases, and thus impede the emergence of a consistent case law. On the contrary, logic contracts can mirror natural language, and thus they should lead to fewer interpretative disagreements and facilitate the work of lawyers, in particular to the comparison of the clauses addressing similar issues. Nevertheless, if a human agreed on a given legal code or rules, then there is likely to be a natural language counterpart, and logic contracts may not be close enough to natural language to be a substitute of it, particularly for people having no technical expertise.

Termination Whatever the programming language, a smart contract and its infrastructure should provide mechanisms to terminate the agreement, or change its content. This is needed since the law prescribes the termination or the modification of legal contracts, under certain conditions. If contractual terms have to be modified (by renegotiation or as a result of dispute resolution), then one can consider previous remarks on the modification of imperative and declarative contracts.

To sum up our comparison of imperative and declarative smart contracts, the declarative approach has significant advantages over its imperative counterpart. However, it is arguable that a full representation of a smart contract has to explicitly establish and link the normative effects (rights, obligation, transfers of entitlement) resulting from the contract with the procedure for implementing these rights and obligations through the computational actions performed by the contract, in the given infrastructure. Hence, a hybrid approach combining imperative and declarative components would help to bridge the gap between smart contracts and their legal counterparts.

5 On imperative and declarative smart contracts in blockchain systems

While previous sections indicate that programming smart contracts and the encountered issues are not a new matter, we reappraise in this section the comparison of imperative and declarative languages for smart contracts in the context of blockchain-based systems. We structure the comparison in terms of the same categories introduced in the previous sections, in order to perform a systematic analysis and to highlight possible differences implied by blockchain systems. We then investigate different options to integrate logic-based smart contracts with such systems. A detailed comparison would suffer from the fact that blockchain-based systems are diverse and offer different characteristics. For this reason, the comparison is kindly kept general, leaving the possibility to accommodate some statements with respect to the choice of a specific platform.

5.1 Comparing imperative and declarative blockchain smart contracts

In the previous section, we compared imperative and declarative smart contracts in relation to matters concerning legal validity, legal interpretation and lifecycle of ordinary contracts. We continue this comparison in the context of blockchain systems.

5.1.1 On legal validity

To determine whether blockchain technology could affect the legal validity of a blockchain smart contract, we consider the four elements identified in Sect. 2, namely agreement, consideration, competence, and legal purpose.

Agreement As previously argued, a declarative language can improve the clarity of an agreement with respect to imperative languages, and the use of blockchain systems does not appear to affect this conclusion.

Consideration Blockchain systems facilitate the specification of monetary transfers, through the use of digital currencies but also 'tokens' that can be tied to anything. Therefore, they contribute to satisfy the requirement that both parties have to exchange something for a contract to be enforceable by a court. However, there is no obvious advantage of the use of a particular programming paradigm for consideration in the context of a blockchain system.

Competence and capacity Blockchain may leverage (the interoperability of) identification, authentication and authorization services, and one can devise services to check the competence and capacity of parties in blockchain systems. Anyhow, there seem to be no substantial benefits of any particular programming paradigm in that regard, besides those previously mentioned.

Legal object and purpose Current blockchain systems offer no particular services to establish the legality of smart contracts. Nonetheless, one can envision services to check the legal purpose of smart contracts in blockchain systems, and as previously indicated, declarative contracts are more suitable for automated checking than imperative contracts.

Blockchain systems can facilitate the implementation of consideration, but the use of such systems do not seem to affect any conclusions previously drawn on the legal validity of smart contracts, may they be imperative or declarative.

5.1.2 On legal interpretation

As previously argued, the distance between the specification of the legal effects and their executable implementation can be minimized by adopting declarative languages, and such a conclusion also holds for smart contracts running on blockchain systems. In fact, blockchain systems are often developed collaboratively (e.g., an open source approach), and in addition the code of a smart contract is made public. This design makes it possible for anyone to check the implementation of terms of legal contracts. This public access to the expression of contract conditions facilitates public verification, and in turn mitigates misinterpretations. As we pointed out before, smart contracts should contain information about the version of the blockchain platform in which they are supposed to be executed (for example, to prevent misinterpretations depending on evolution of the features available in the language at the time of execution).

5.1.3 On lifecycle

Let us now compare imperative and declarative blockchain smart contracts with regard to different stages of legal contracts.

Negotiation and formation The negotiation and formation can occur off-chain or on-chain. An agreement about what a contract should perform is usually defined in a natural language (as in the case of the creation using a template), and then this contract is translated into a smart contract. This translation can occur on-chain or off-chain, but since translation is often a costly operation, it is likely to occur off-chain. However, if a blockchain network is established among the participants, it provides a shared infrastructure for participants to negotiate legal contracts and track all steps of the negotiation. If participants to a contract are unknown, then the programmer could implement a framework contract, which in turn contains all information needed for instantiating the smart contract. In this case, the template of the smart contract is created like a normal program and is then instantiated on-chain. As the smart contract meant to be instantiated on-chain is in bytecode on the blockchain, one can argue that instantiation is more a programming interface issue of the considered smart contract than a matter of language paradigm.

Whatever the way a smart contract is negotiated and created, its well-formedness is important to ensure an acceptable execution. The DAO attack to siphon funds (DAO was a decentralized autonomous organization featuring a form of investor-directed venture capital fund) in Northern Summer 2016 has shown the serious consequences from insufficient verification and testing of smart contracts. In that regard, the validation and verification of smart contracts running over blockchains and distributed ledgers give rise to new interesting issues and research challenges (Magazzeni et al. 2017), where the choice of the programming language may be decisive.

Contract storage and notarizing Instead of storing the smart contract into the machine(s) of particular entities (such as the parties and intermediaries), one can use a blockchain system to store it (its bytecode) with a relatively accurate timestamp. Although blockchains are not meant to store big data, there are no particular restrictions on the types of data that can be stored in blockchains, and therefore smart contracts with logic statements can be stored in them. As a set of logic statements (e.g., the set of rules stored within a smart contract and meant to be passed to a rule engine) are generally more compact than its imperative counterpart, the logic approach may decrease the cost of storage if a public blockchain is used, in particular when there is an explosion of possible states on which rules can be applied.

Execution As blockchain systems provide a computational environment to execute smart contracts, one may be interested in (optimising) non-functional properties, concerning for example performance, interoperability and costs.

- *Performance* Blockchain systems where all smart contract transactions are recorded into a single blockchain do not compete with well-established conventional transaction processing systems (such as payments network) in terms of throughput, i.e., the total number of transactions a system can process within a time window. For latency, i.e., the time required to respond to a single transaction, reading can be improved but writing is worsened. The computational complexity of the programming language is likely to affect these aspects, and previous remarks on the complexity of the programming language still hold; these remarks are actually stressed out by possible mining costs and incentives.
- *Interoperability* When a blockchain system is used as a shared computational platform to execute smart contracts, then interoperability between the smart contracts may be better supported. However, if the smart contracts run on different systems, and even if declarative smart contracts can aid semantic interoperability, then the ability to exchange and use information may suffer, in particular because calls from blockchain systems are usually prohibited.
- *Costs* One may be interested in minimizing the cost of execution through blockchain systems. Since imperative smart contracts are usually less demanding than declarative smart contracts in terms of actual computational complexity (provided that the programmers properly catered for it), imperative smart contracts have a clear advantage here.

Concerning the halting problem, current blockchain systems can stop a smart contract at some point when some computational budget is reached (e.g., as in Ethereum), but one may prefer to avoid such scenarios. In that regard, declarative smart contracts may prevent them, for example when the inferences mechanisms prevent infinite loops.

Monitoring and enforcement Though blockchain smart contracts are presumably more difficult to breach, some portions of the underlying legal contract may remain to be executed beside the smart contract, and thus monitoring may remain a necessity. Actually, monitoring can appear challenging in blockchain systems. A reason in particular is that there are often discrepancies between legal and blockchain times. Legal contracts often include temporal constructs such as deadlines, and the time from the paced validation of blocks does not necessarily fits this legal time. Though declarative representations are arguably more ingenious to capture temporal legal knowledge (Governatori et al. 2007), they are subject, as imperative representations, to the temporal granularity of blockchain systems. Besides, as on-chain monitoring has a monetary cost, one can argue that imperative smart contracts are less expensive to monitor. Nevertheless, a declarative monitoring can better ensure soundness and completeness compared to an imperative monitoring, preventing thus breaches and disputes.

Modification In current blockchain systems, a contract cannot be directly modified but the data stored in it can be updated. Typically, 'public' variables in imperative smart contracts can be updated to bring some flexibility. For example, flexible solutions are enabled by the 'hub and spoke' model, where one main smart contract holds addresses/pointers to all other necessary contracts that contain the specific clauses and functionality. Public variables, possibly along with a hub and spoke model, allow the modification of imperative smart contracts, but this solution may appear quite coarse, and possible modifications are limited to the range of variables anticipated by programmers. In logic-based smart contracts, the statements of the knowledge base can be coded as 'public' variables too. As the range of such variables can be immensely large and fine-grained, logic-based smart contracts can offer a larger range of fine-grained updates. A modified knowledge base can also be passed to an existing contract, which then acts accordingly, similar to how in the hub and spoke model addresses of subcontracts are exchanged. A logic-based language can thus greatly help to tackle modifications.

Dispute resolution Blockchain smart contracts can be disputed too, and adjudicative resolution as well as consensual resolution can be attempted. Based on Bitcoin case law and the freedom to contract, one may argue that smart contracts are binding (Wright and De Filippi 2015, pp. 11–24). The final arbiter of legal technological innovation is always acceptance by the courts. At the moment there is no useful case law on smart contracts, and acceptance would also depend strongly on the nature of the smart contract, i.e., whether it is linked to a contract in natural language as well as other (above-mentioned) factors.

Termination In blockchain systems where smart contracts cannot be blocked (unless the relevant account of money is emptied, or the smart contract has some programming interfaces to block it, etc.), a party who is unsatisfied with the content or the execution of the contract has no way to stop the contract performance, for example by claiming that the contract is invalid or that the other party has failed to comply. If the contract is unstoppable and immutable, then one has to rely on the initial code and the programmers who must anticipate all the states reached by parties during the life of the contract, with no errors in coding (in particular with no infinite loops extremely costly in blockchain systems). Such a perfect anticipations are unlikely to occur for most contracts in practice, and for this reason, blockchain smart contracts can be highly problematic, whatever the language paradigm. Nevertheless, we have to emphasize that a smart contract may be easily blocked in practice by using appropriate interfaces (ultimately by 'destroying' it if necessary).

To sum up, blockchain systems can be beneficial to some contractual activities, in particular for the storage/notarizing and execution of smart contracts, without central control. However, one can point out some fundamental problems raised by such systems, especially to manage modifications and termination. As to the programming paradigm, a declarative language may facilitate the modification of blockchain smart contracts, but it may be less attractive in terms of computational complexity compared to imperative counterparts. This point is further investigated next.

5.2 Integrating logic-based smart contracts in blockchain systems

While imperative languages are commonly used to program smart contracts in blockchain systems, declarative languages have been hardly explored. To get more insights on the possible use of declarative smart contracts in combination with blockchain systems, we have to discus their integration. Blockchain systems have many configurations and variants (Xu et al. 2017). For our purposes we discuss here two main options, where inferences occur on-chain or off-chain.

- On-chain: inferences are made within the blockchain platform;
- Off-chain: inferences are made outside the blockchain system, e.g., on a third party server.

The distinction between on-chain and off-chain inferences leads to the distinction of on-chain and off-chain options for logic-based smart contracts. We will investigate these different technical options with regard to the legal occurrences we previously identified.

5.2.1 Off-chain options

When miners are processing transactions into blocks to append to the blockchain, the security model of the virtual machine (in which smart contracts on existing blockchain platforms operate) and the co-processing by nodes do not usually allow to call outside resources. Thus, unless some kinds of oracles are available in a way or another, we must discard the option where an off-chain inferential mechanism is called by the smart contract.

Another off-chain option simply consists in executing the declarative smart contract off-chain. Possibly, the smart contract (i.e., the declarative specification of its content and the reference to the semantics) or the inferential conclusions are recorded on the blockchain. On the basis of the inferential conclusions, imperative code can then execute particular transactions. Such off-chain options are illustrated in Fig. 3. Contract activities that we identified in the previous sections are accomodated as in Table 1.

An advantage of this off-chain option is the lower cost of associated transactions, since the inferences are performed off-chain, cf. Rimba et al. (2017) for a cost

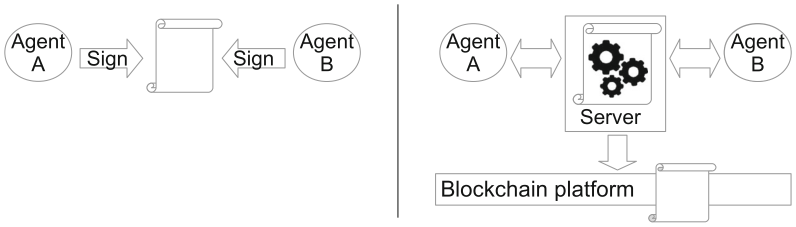

Fig. 3 Off-chain option. Agents A and B form a (smart) contract which is stored on a blockchain. The contract is executed in a server external to the blockchain system, and transactions can be recorded in the blockchain

Table 1 Contract lifecycle activities for the off-chain option

Formation and negotiation	The contract can be formed and negotiated off-chain or on-chain
Contract storage and notarizing	A contract is stored off-chain (to be executed off-chain) and in the blockchain (without being executed on-chain)
Execution	The execution is performed off-chain
Monitoring and enforcement	Monitoring and enforcement are achieved off-chain, the results can be stored in the blockchain
Modification	If a contract is modified, then the off-chain smart contract can be updated, and stored in the blockchain. If the knowledge base can be updated, then the smart contract can be updated without interrupting associated processes
Dispute resolution	One can check whether an off-chain contract matches a blockchain code (bytecode). Thus in case of a dispute, the parties can check whether the recorded conclusions are proper conclusions of the smart contract
Termination	As a smart contract runs off-chain, it is presumably easily stoppable

comparison for computation and storage on early blockchain vs. a popular cloud service. Furthermore, when the latency or the throughput appears problematic in blockchain systems where smart contracts are meant to be run on-chain, such an off-chain option can improve performance by bypassing blockchain bottlenecks.

A disadvantage concerns the possible costs and trust issues in such off-chain solutions. If the off-chain inferential mechanism is run by an intermediary service, then the parties shall have to deal with such intermediation, and associated costs or trust issues. Nevertheless, the contract may be executed without a central integration point, as for example in a peer-to-peer architecture. This option has to provide mechanism to ensure the correct execution of smart contracts (see the discussion about information about the language and execution environment above).

Another solution to mitigate trust issues consists in off-chain smart contracts interacting with blockchain systems through core functionalities, see e.g., Nxt or Ardor application programming interfaces (Nxt 2018; Jelurida 2017), yielding thus solutions between off-chain and more full-fledged on-chain options.

5.2.2 On-chain options

Instead of an off-chain inferential mechanism, one may prefer an on-chain mechanism. The availability of a logic-based language to program smart contracts shall facilitate such options, but an imperative language can also be used to write meta-programs (i.e., programs with the ability to treat programs as their data). For example, a rule engine can be integrated in a smart contract to derive conclusions given a particular knowledge base. Based on the results, some imperative code can execute the transactions. The rule engine can also be a smart contract script of its own, so that smart contracts can always refer to this smart contract. Having the inference engine as an immutable contract on the blockchain allows participants' confidence into the smart contract engine to increase over time (test once, utilize *n*-fold). Such on-chain options are illustrated in Fig. 4. Contract lifecycle activities are accomodated as in Table 2.

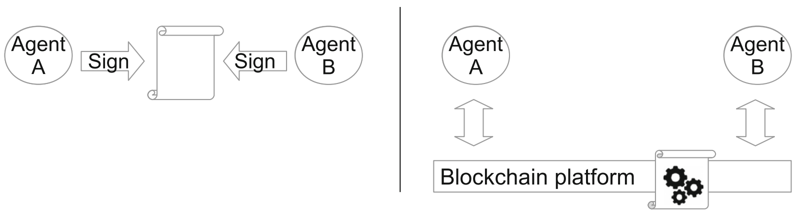

Fig. 4 On-chain option. Here, agents A and B form a (smart) contract which is stored and executed in a blockchain platform

Table 2 Contract lifecycle activities for the on-chain option

Formation and negotiation	The contract can be formed and negotiated off-chain or on-chain
Contract storage and notarizing	A contract can be stored off-line, but it has to be stored in the blockchain (so that it can be executed on-chain)
Execution	Execution is performed on-chain
Monitoring and enforcement	Monitoring and enforcement are achieved on-chain, the conclusions can be stored in the blockchain
Modification	If the knowledge base can be updated, then the contract can be updated without interrupting associated processes
Dispute resolution	One can check whether an off-chain contract matches a blockchain code. Thus in case of a dispute, the parties can check whether the recorded conclusions are proper conclusions of the smart contract
Termination	Unless means are given to block the smart contract, it is presumably unstoppable

The major advantages of on-chain solutions are that some off-chain intermediation services are eliminated, and the inferential mechanisms (e.g., the rule engine) are themselves recorded in the blockchain, resulting into more scrutable and trustful inferences.

Main disadvantages of such on-chain solutions are possibly those of blockchain systems as previously mentioned. They may concern latency, throughput or costs. To address the costs of on-chain inferences, algorithms with low computational complexity shall be favoured. If the selected algorithm provides inferences which appears sufficiently efficient but insufficiently intelligible for human operators, then more intelligible inferences can be used to explain the results off-chain.

It is also possible to propose an on-chain option, that we may call the 'on\off-chain' option where, given a declarative specification or knowledge base, this knowledge is converted into imperative representations (e.g., Solidity code) or to intermediate bytecode embedded into the blockchain (this compiled code is part of the smart contract), and this smart contract is eventually recompiled to run on the virtual machines of the blockchain network. Such an on\off-chain option is illustrated in Fig. 5. Lifecycle activities are accomodated as in Table 3.

Compared to the off-chain option, the need for intermediation services is mitigated since inferences are achieved on-chain. Compared to the on-chain option, the costs of transactions may be decreased because the compiled knowledge base is

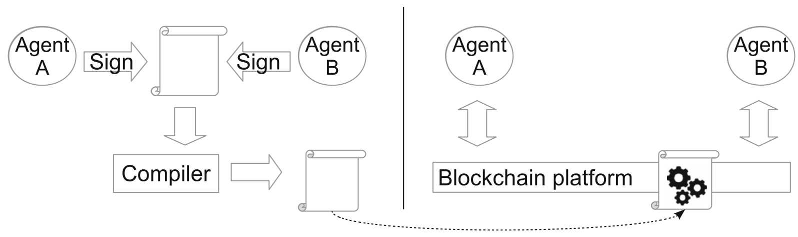

Fig. 5 On\off-chain option. Agents A and B form a (smart) contract which is compiled. The compiled contract is stored and executed in a blockchain platform

Table 3 Contract lifecycle activities for the on\off-chain option

Formation and negotiation	The contract can be formed and negotiated off-chain or on-chain. The compiled code can be generated off-chain or on-chain. If compilation occurs off-chain then third party services may again appear, along with the associated disadvantages. If compilation is done on-chain then the compiler may be scrutinized and gain trust from the parties, at the expense of extra costs for compilation
Contract storage and notarizing	A contract and its compiled code can be stored off-chain, but the compiled code has to be stored in the blockchain (so that it can be executed on-chain)
Execution	Execution is performed on-chain.
Monitoring and enforcement	Monitoring and enforcement are achieved on-chain, the conclusions can be stored in the blockchain
Modification	If a contract is modified, then the logic statements have to be recompiled. If the compiled knowledge base can be updated, then the contract can be updated without interrupting associated processes
Dispute resolution	One can check whether the compiled off-chain contract matches a blockchain code. Thus in case of a dispute, the parties can check whether the recorded conclusions are proper conclusions of the smart contract (w.r.t. the given semantics)
Termination	Unless means are given to block the smart contract, it is presumably unstoppable

meant to lower the computational complexity. The costs can be presumably higher than the off-chain option, therefore, such on\off-chain approaches shall have a cost intermediate between off-chain and on-chain solutions.

Whatever the option, and as previously mentioned, verification of conclusions should be possible, and understandable by humans. In this regard, given some semantics, conclusions of efficient but unintelligible approaches can be verified off-chain by more comprehensible proof systems.

6 Conclusion

This paper provides an analysis of the links between smart contracts and legal contracts. It also provides a comparative analysis of imperative and declarative (logic-based) languages for smart contracts expression, considered from the context of blockchain systems. We structured the investigation in terms of legal validity, factors of legal interpretation and a common contract lifecycle.

We showed that declarative smart contracts can offer technical and legal benefits compared to their imperative counterparts. Declarative smart contracts can better fit fundamental elements of legal contracts. Declarative contracts are also advantageous in terms of closing the gaps of interpretation between the actual clauses and their implementation. Finally, they can better deal with common stages of a contract lifecycle.

The comparative conclusions generally also apply to smart contracts deployed on blockchain systems, with particular compelling benefits. Blockchain systems with their integrated digital currencies can facilitate contractual consideration. Misinterpretation of terms may be also reduced if smart contracted are published. Blockchain systems can also be beneficial to the storage/notarizing and execution of smart contracts, without central control. However, fundamental problems can appear in relation to the modification and termination of smart contracts. A declarative language may simplify the modification of blockchain smart contracts, but, compared to imperative counterparts, it may fall short of expectations in matters of computational complexity and associated costs. For these reasons, we have to emphasize that imperative and declarative approaches are not incompatible, on the contrary, they have the potential to complement each other, possibly leading to interesting theoretical and practical opportunities.

Acknowledgements The paper is an extended and revised version of Idelberger et al. (2016) presented at RuleML 2016 conference.

References

Aberdeen Group (2005) The contract management solution selection report. In: Handbook for CLM strategy and solution selection. Aberdeen Group

Athan T, Governatori G, Palmirani M, Paschke A, Wyner A (2015) LegalRuleML: design principles and foundations. In: Proceedings of the 11th reasoning web summer school. Springer. pp 151–188

Berners-Lee T, Hendler J, Lassila O (2001) The semantic web. Sci Am 284(5):34–43

Canesin FC, Xiang YJ, Lim J, Fast E, Lowenthal J, Fong A, van den Brink E (2018) NEO white paper

Clack CD, Bakshi VA, Braine L (2016a) Smart contract templates: essential requirements and design options. CoRR, abs/1612.04496

Clack CD, Bakshi VA, Braine L (2016b) Smart contract templates: foundations, design landscape and research directions. CoRR, abs/1608.00771

Daskalopulu A, Sergot MJ (1997) The representation of legal contracts. AI Soc 11(1):6–17

Dimitrakos T, Djordjevic I, Milosevic Z, Jøsang A, Phillips CI (2003) Contract performance assessment for secure and dynamic virtual collaborations. In: Proceedings of the 7th international enterprise distributed object computing conference. IEEE Computer Society, pp 62–75

Ethereum Foundation E(2016) Thereum's white paper

Farell ADH, Sergot MJ, Salle M, Bartolini C (2005) Using the event calculus for tracking the normative state of contracts. Int J Cooper Inf Syst 14(02n03):99–129

Fenech S, Pace GJ, Schneider G (2009) Automatic conflict detection on contracts. In: Proceedings of the 6th international colloquium on theoretical aspects of Computing. Springer, pp 200–214

Gabbay D, Horty J, Parent X, van der Meyden R, van der Torre L (eds) (2013) Handbook of deontic logic and normative systems. College Publications, London

Gelati J, Rotolo A, Sartor G, Governatori G (2004) Normative autonomy and normative co-ordination: Declarative power, representation, and mandate. Artif Intell Law 12(1–2):53–81

Governatori. G (2005) Representing business contracts in RuleML. Int J Cooper Inf Syst 14(2–3):181–216

Governatori G (2015) Thou shalt is not you will. In: Proceedings of the 15th international conference on artificial intelligence and law. ACM, pp 63–68

Governatori G, Milosevic Z (2005) Dealing with contract violations: formalism and domain specific language. In: Proceedings of the 9th IEEE international EDOC enterprise computing conference. IEEE Computer Society, pp 46–57

Governatori G, Milosevic Z (2006) A formal analysis of a business contract language. Int J Cooper Inf Syst 15(4):659–685

Governatori G, Pham DH (2009) DR-CONTRACT: an architecture for e-contracts in defeasible logic. Int J Bus Process Integr Manag 5(4):187–199

Governatori G, Rotolo A (2013) Computing temporal defeasible logic. In: Theory, practice, and applications of rules on the web. Springer, pp 114–128

Governatori G, Hulstijn J, Riveret R, Rotolo A (2007) Characterising deadlines in temporal modal defeasible logic. In: Proceedings of the 20th Australian joint conference on artificial intelligence. Springer, pp 486–496

Grosof BN, Labrou Y, Chan HY (1999) A declarative approach to business rules in contracts: courteous logic programs in xml. In: Proceedings of the 1st ACM conference on electronic commerce. ACM, pp 68–77

Hanson JE, Milosevic Z (2003) Conversation-oriented protocols for contract negotiations. In: Proceedings of the 7th international enterprise distributed object computing conference. IEEE Computer Society, pp 40–49

Hess Z, Malahov Y, Pettersson J (2017) Aeternity blockchain

Idelberger F, Governatori G, Riveret R, Sartor G (2016) Evaluation of logic-based smart contracts for blockchain systems. In: Proceedings of the 10th international web rule symposium. Springer, pp 167–183

Jelurida (2017) Ardor white paper

Jones SLP, Eber JM (2003) How to write a financial contract

Kern A, Walhorn C (2005) Rule support for role-based access control. In: Proceedings of the 10th ACM symposium on access control models and technologies. ACM, pp 130–138

Kowalski R (1979) Algorithm = logic + control. Commun ACM 22(7):424–436

Lam HP, Governatori G (2009) The making of SPINdle. In: Proceedings of the international symposium on rule interchange and applications. Springer, pp 315–322

Linington PF, Milosevic Z, Cole JB, Gibson S, Kulkarni S, Neal SW (2004) A unified behavioural model and a contract language for extended enterprise. Data Knowl Eng 51(1):5–29

Lomuscio A, Penczek W, Solanki M, Szreter M (2011) Runtime monitoring of contract regulated web services. Fundam Inform 111(3):339–355

Magazzeni D, McBurney P, Nash W (2017) Validation and verification of smart contracts: a research agenda. Computer 50(9):50–57

Marjanovic O, Milosevic Z (2001) Towards formal modeling of e-contracts. In: Proceedings of the 5th international enterprise distributed object computing conference, pp 59–68

Milosevic Z (1996) Enterprise aspects of open distributed systems. Ph.D. thesis, University of Queensland

Milosevic Z, Arnold D, O'Connor L (1996) Inter-enterprise contract architecture for open distributed systems: security requirements. In: Proceedings of the 5th workshop on enabling technologies: infrastructure for collaborative enterprises, pp 68–73

Molina-Jimenez C, Shrivastava S, Solaiman E, Warne J (2004) Run-time monitoring and enforcement of electronic contracts. Electron Commer Res Appl 3(2):108–125

Montgomery N, Wilson DR (2015) Market guide for contract life cycle management, Gartner, ID: G00276707

Nakamoto S (2008) Bitcoin: A peer-to-peer electronic cash system

Nxt Community (2018) Nxt white paper

OASIS (2007) eContracts Version 1.0, Committee Specification

Prakken H, Sartor G (2015) Law and logic: a review from an argumentation perspective. Artif Intell 227:214–245

Rimba R, Tran AB, Weber I, Staples M, Ponomarev A, Xu X (2017) Comparing blockchain and cloud services for business process execution. In: Proceedings IEEE international conference on software architecture. IEEE, pp 257–260

Sartor G (2005) Legal reasoning: a cognitive approach to the law. Springer, Berlin

Staples M, Chen S, Falamaki S, Ponomarev A, Rimba P, Tran AB, Weber I, Xu X, Zhu J (2017) Risks and opportunities for systems using blockchain and smart contracts. Technical report, Data61 (CSIRO), Sydney

Szabo N (1997) The idea of smart contracts

Weber I, Xu X, Riveret R, Governatori G, Ponomarev A, Mendling J (2016) Untrusted business process monitoring and execution using blockchain. In: Proceedings of the 14th international conference on business process management, vol 9850. Springer, pp 329–347

Wood G (2014) Ethereum: a secure decentralised generalised transaction ledger

Wright A, De Filippi P (2015) Decentralized Blockchain Technology and the Rise of Lex Cryptographia. SSRN scholarly paper ID 2580664, Social Science Research Network

Xu X, Weber I, Staples M, Rimba P (2017) A taxonomy of blockchain-based systems for architecture design. In: Proceedings IEEE international conference on software architecture. IEEE, pp 243–252