frontiers in Blockchain

PERSPECTIVE
published: 04 June 2020
doi: 10.3389/fbloc.2020.00027

# Smart Contracts Contracts

*Massimo Bartoletti**

*Dipartimento di Matematica e Informatica, Università di Cagliari, Cagliari, Italy*

This paper explores the connection between software contracts and smart contracts. Despite the assonance, these two terms denote quite different concepts: software contracts are logical properties of software components, while smart contracts are programs executed on blockchains. What is the relation between them? We answer this question by discussing how to integrate software contracts in the design of programming languages for smart contracts.

**Keywords: blockchain, smart contracts, design by contract, formal models, verification**

## 1. INTRODUCTION

Smart contracts (Szabo, 1997) are pieces of software which regulate the exchange of resources (assets—including money—and services) between participants. The execution of smart contracts can take advantage of blockchain technologies, which allow mutually untrusted participants to agree on a global state, without the intermediation of a trusted authority. Smart contracts have been redefined and popularized by Ethereum, a public permissionless blockchain through which users can exchange a cryptocurrency, and tokens representing a multitude of other crypto-assets. Ethereum features a Turing-equivalent programming language for writing code that is stored on the blockchain, and which is called smart contract in Ethereum's terminology. Once a smart contract is published, its code cannot be changed, and anyone can interact with it. Hence, adversaries may try to exploit security vulnerabilities in the contract to steal crypto-assets, or cause other harm. The history of Ethereum is studded with attacks to its smart contracts, which overall have caused money losses in the order of hundreds of millions of dollars.

Could these attacks have been avoided? To answer this question, we must consider the vulnerabilities that have made them possible. These vulnerabilities are mainly attributable to design issues in the programming language used to write smart contracts (Luu et al., 2016; Atzei et al., 2017). One of the tools that have been used by software engineers to address similar issues is that of *software contracts*. Intuitively, a software contract is a formal specification of the behavior of a software component, which describes the duties that must be fulfilled by the users of the component, as well as the obligations of the component to its callers.

The main similarity between software contracts and smart contracts is that both are a form of commitment between a piece of software and its callers. For software contracts, the commitment is that the piece of software respects some properties, provided that the caller feeds it with correct inputs. For smart contracts, the commitment is somehow weaker: it just requires that all the executions of the piece of software are coherent with its semantics (i.e., code cannot be changed).

This difference is due to the diverse assumptions on the execution environment. In software contracts, the caller is typically assumed to control the environment where the piece of software is run: so, coherence with the semantics can be taken for granted. Instead, in smart contracts, the piece of software is concurrently executed by a network of mutually untrusted nodes, which may have economic incentives to cheat. In this setting, guaranteeing coherence with the semantics is not trivial: complex network protocols are in order to ensure that the only rational behavior for a node is not to cheat. The most significant outcome of the Bitcoin protocol was indeed to show that trusted executions in trustless environments were possible.

OPEN ACCESS

***Edited by:***
*Christopher Clack,*
*University College London,*
*United Kingdom*

***Reviewed by:***
*Reshma Kamath,*
*Blockchain Research Institute, Canada*
*Kristin B. Cornelius,*
*UCLA Department of Information Studies, United States*

***Correspondence:**
*Massimo Bartoletti*
*bart@unica.it*

***Specialty section:***
*This article was submitted to Smart Contracts, a section of the journal Frontiers in Blockchain*

***Received:*** *05 January 2020*
***Accepted:*** *05 May 2020*
***Published:*** *04 June 2020*

***Citation:***
*Bartoletti M (2020) Smart Contracts Contracts. Front. Blockchain 3:27. doi: 10.3389/fbloc.2020.00027*

Other remarkable differences between software contracts and smart contracts emerge. While software contracts may describe arbitrary properties of a software component, smart contracts typically define the rules through which participants exchange resources (crypto-currencies or other crypto-assets). This is why the participants of a smart contract can be considered as adversaries, as they may try to steal or freeze the resources controlled by the contract. Instead, in most incarnations of software contracts (with the exceptions discussed below), participants are assumed to behave honestly.

Given the above differences, which ideas from software contracts can be applied to make smart contracts more reliable? In this paper, we discuss some possible research directions to answer this question. We start by providing a brief overview of software contracts, and then we discuss a few research perspectives.

## 2. SOFTWARE CONTRACTS

The idea of software contract dates back at least to Hoare's seminal paper (Hoare, 1969), that introduced the axiomatic approach to computer programming. Since then, software contracts have been developed over the past three decades both in the academy and in the industry. In Hoare's approach, the semantics of a programming language is formalized as a set of axioms, and the behavior of a program is given in terms of a precondition and a postcondition. These are logical assertions which describe, respectively, the properties that must hold before and after executing the program. Assertions can be interpreted as contracts: the precondition establishes the obligations that the caller must fulfill before running the program, while the postcondition dictates the obligations of the program. For instance, for a program which merges two arrays, the precondition requires that the values of two input arrays are sorted, while the postcondition is that the output array contains, sorted, the elements of the two arrays in input.

Although the original goal of the axiomatic approach was to prove the correctness of programs, in software development practice the use of assertions as contracts has become widespread for a weaker goal, that is to make program testing more rigorous. Assertions are spread over large programs, dynamically checked in massive tests, and possibly removed before the actual deployment. In particular, the *design by contract* principle fostered by Meyer since the late eighties (Meyer, 1986, 1992) requires that the interfaces between modules of safety-critical software systems should be governed by contracts. These contracts are decoupled from the rest of the program logic, so that they can be easily discharged after the testing phase. Another landmark feature of contracts is that they enable to assign *blame* in case of contract violations. Assume that a program module has a precondition $P$ and a postcondition $Q$. During the execution of the program, if $P$ is violated then the blame is assigned to the caller; instead, if $P$ is respected but $Q$ is violated, then the blame is assigned to the callee. The first programming language with native support for design by contract was Eiffel, in the object-oriented paradigm (Meyer, 1991). Since then, almost all mainstream programming languages offer some support for contracts, either natively (e.g., Scala and Clojure), or through external libraries (e.g., JML for Java, Code contracts for C#).

In the functional paradigm, where program modules can be higher-order functions, contract checking and blame assignment are more complex than in the procedural and object-oriented paradigms. For instance, assume that a function $g$ takes as input a function $f$ from integers to integers, and produces an integer as output. While the postcondition of $g$ can be a first-order predicate $R$ on integers, its precondition cannot: rather, it should consist of a predicate $P$ restricting the acceptable arguments passed to $f$, and a predicate $Q$ restricting its possible outputs. So, the contract of $f$ can be rendered as a function $P \to Q$, and that of $g$ as a higher-order function $(P \to Q) \to R$. By decidability issues it is not possible, in general, to determine whether $f$ respects its contract or not when $g$ is applied. A possible way to tackle this problem, first explored in Findler and Felleisen (2002), is to check $f$'s contract when $f$ is applied. At that moment it is possible to know if the argument passed to $f$ respects $P$, and if the value produced by the application of $f$ respects $Q$, and to assign the blame accordingly. Findler and Felleisen's seminal work has given rise to a proliferation of studies on higher-order contracts, leading to a wide variety of languages and analysis techniques. The most notable functional language supporting the design by contract approach is Racket (Flatt and PLT, 2010), a dialect of Scheme with higher-order assertions.

Going beyond the incarnation of contracts as pre- and post-conditions on program modules, Beugnard et al. (1999) proposed a taxonomy of software contracts consisting of four kinds of contracts:

- *basic contracts*, which describe the syntactic constraints that must be respected to use a program module (e.g., the basic contract of a function that takes as input an integer and produces as output a string is a type int → string);
- *behavioral contracts*, which impose semantic constraints on program modules, in the form of pre- and post-conditions as discussed so far;
- *sequencing contracts*, which impose constraints on how the different modules offered by a software component can be used (e.g., the `withdraw` method of a `BankAccount` object cannot be executed concurrently with the `deposit` method);
- *quality of service contracts*, which impose time, space, and precision constraints on the execution of a program module (e.g., any call to the `openAirbag` procedure must terminate within 10 ms).

In the setting of concurrent programming models, contracts are used to describe the possible sequences of messages that can be processed by concurrent components (Hüttel et al., 2016). For instance, the contract of a payment service may require that the service first receives a `cardDetails` message, then either a `proceed` or an `abort` message; in case of `proceed`, the service sends back to the client either a message `paymentOk` or a message `paymentError`. Using a syntax inspired to that of Web Service

contracts (Castagna et al., 2009), we can express this contract as follows:

$$\texttt{?cardDetails}\,.\,(\texttt{?abort} + \texttt{?pay}\,.\,(\texttt{!paymentOk} \oplus \texttt{!paymentError}))$$

where ? denotes a received message, ! a sent message, . sequencing, + a choice made by the client, and $\oplus$ a choice made by the service. In this setting, a typical goal is to determine if two contracts are *compliant*: intuitively, compliance between contracts guarantees the absence of certain kinds of communication errors in any execution of services which respect their contracts. Many different notions of compliance can be used for this purpose, based e.g., on deadlock-freedom, on may- and must-testing relations, etc. (Bordeaux et al., 2004; Bartoletti et al., 2015b).

Starting from Honda's seminal paper on dyadic session types (Honda, 1993), many works address the problem of statically verifying if a service (formally specified in some calculus for concurrency) respects its contract. The overall result is that if two services are statically verified to respect their contracts, and these contracts are compliant, then no communication errors can happen at runtime. Extensions of this kind of results to the multi-party setting have been studied by Honda et al. (2008) and many subsequent works.

The work (Bocchi et al., 2010) nicely integrates assertion-based contracts with concurrency contracts. The idea is to project a service choreography decorated with assertions to a set of local assertions: services abiding these local assertions are guaranteed to have correct interactions at run time. A dual approach is to start from an arbitrary set of services, and to dynamically compose those with compliant contracts (Bartoletti et al., 2012). A remarkable difference between this approach and the other approaches to concurrency contracts discussed before is that the latter assume that all services are *trusted*, in that they will not change their code after deployment (as this would invalidate their static verification). Instead, in Bartoletti et al. (2012) this assumption can be limited to a subset of services, while all the others are considered as *adversaries*. Under this weaker hypotheses it is no longer possible to ensure the absence of contract breaches, as an adversary could diverge from the declared contract, or just stop cooperating at any time. Rather, the goal of the static analysis is to guarantee that the trusted services always respect their contracts, whatever the behavior of the adversaries. This property, called *honesty*, becomes relevant when services interact through communication middlewares which sanction those not abiding to their contracts (Bartoletti et al., 2015a). In this setting, honest services are guaranteed to never be sanctioned.

## 3. CONTRACTS FOR SMART CONTRACTS

We now discuss some possible research directions on connecting software contracts with smart contracts.

### Route #1: Embedding Assertions Into Existing Smart Contract Languages

A straightforward idea would be to add assertions to existing smart contract languages, in the spirit of the design by contract methodology.

A first step towards this direction has been taken by Solidity, the mainstream high-level language for writing smart contracts for the Ethereum platform. Starting from version 0.4.10[1], Solidity features two functions (called `assert` and `require`), which throw a state-reverting exception if the given conditions are not respected. The two functions slightly differ on the handling of *gas* (i.e., the fee paid by participants for the execution of contract calls). When the condition of `assert` is violated, all the gas provided by the caller is consumed; instead, a failing `require` refunds any remaining gas. Although these functions provide useful syntactic sugar for the if-throw pattern, they do not really capture the essence of assertions in the design by contract methodology. For instance, they do not separate the duties between the caller and the smart contract: in design by contract, preconditions and postconditions represent, respectively, duties of the caller and of the callee; instead, both `assert` and `require` punish the caller in case of violations, by consuming the gas he provided to perform the call.

Design by contract extensions of Solidity have been proposed by external tools, like e.g., SOLC-VERIFY (Hajdu and Jovanovic, 2020). This tool allows developers to annotate Solidity code with contract invariants, loop invariants, pre- and post-conditions. These conditions can be arbitrary Solidity expressions without side effects. A static analysis of the annotated code, based on SMT solving techniques, discharges the conditions which are found to be always satisfied.

A limitation of working with general-purpose languages like Solidity is that they make it difficult to specify and verify properties which capture high-level properties of smart contracts (for instance, that a participant loses at most a certain amount of money in every maximal interaction with the contract). One of the reasons of this difficulty is that the interaction between the participants and the contract is not rigidly structured, since participants can call any contract method, at any time. Using more structured smart contract languages would allow to increase the level of abstraction of contract properties.

### Route #2: Designing Domain-Specific Languages for Smart Contracts and Assertions

Going beyond Ethereum and Solidity, a more radical research direction is the study of new domain-specific languages for smart contracts, with native support for design by contract. When following this direction, two crucial choices are the primitives of the smart contract language, and the formalism for describing assertions and contract properties.

The choice of the smart contract language may be dictated by the choice of the underlying blockchain platform: for instance, it

[1] https://solidity.readthedocs.io/en/develop/control-structures.html#error-handling-assert-require-revert-and-exceptions

would be inappropriate to design a Turing-equivalent language for Bitcoin contracts, since the current restrictions on Bitcoin scripts and transactions make the expressiveness of Bitcoin contracts quite limited (Atzei et al., 2019). Even when the underlying blockchain supports Turing-equivalent contracts, like e.g., in Ethereum and Cardano, it could still be useful to restrict the expressiveness of high-level contract languages, since this would improve their verification capabilities, besides making it simpler for humans to understand programs. Several contract languages have been proposed along these lines, with different expressiveness degrees: see e.g., Harz and Knottenbelt (2018), Tikhomirov (2020), and Miller et al. (2018) for some references.

To clarify how design by contract may take advantage of restricted domain-specific languages, consider a financial contract involving three participants: an investor, a bank and an insurance company. The contract behaves like a zero-coupon bond: the investor pays (say) EUR 1000 upfront to the bank, and the bank returns EUR 2000 to the investor after a maturity date (say, 10 years). To mitigate the risk that the bank fails to repay the investor, the insurance company guarantees to cover the full amount of EUR 2000 for an annual premium of EUR 100 paid by the bank. Focussing on the investor, the desired high-level behavior is that, in any possible interaction with the contract, the investor will gain EUR 1000 within 10 years. In restricted smart contract languages like those proposed by Atzei et al. (2019) and Seijas et al. (2020) it is possible to craft a contract that is statically verified to respect this high-level property. The actual contract actually depends on the assumptions on the other participants (e.g., if the bank, or the insurance company, or none of them is considered honest), and on the amounts initially deposited in the contract.

## Route #3: Taking Into Account Participants' Strategies

Most current analysis tools for smart contracts assume that, at any time, any participant can do any action. While this assumption perfectly makes sense when one wants to detect certain vulnerabilities of smart contracts (e.g., re-entrancy bugs in Ethereum), taking into account the *strategies* followed by participants may substantially improve the precision of the analysis.

For instance, consider a two-players gambling game with the following rules. First, each player puts a bet and commits to a move. Within 1 week, each player must reveal his move, or otherwise lose the bet. Once both moves have been revealed, the game determines the winner, according to some logics. A desirable property of this game is *fairness*, i.e., each player has a strategy that guarantees him to have at least the same winning probability of the other player. From the point of view of a player, it would be irrational to follow a strategy where he waits more than 1 week before revealing, since doing so would make him loses the bet. Rather, he would like to verify that the game is fair, assuming that his strategy is to reveal within the deadline, while not making any assumptions on the strategy of the other player (who should be considered as an adversary).

This kind of properties can be verified on Bitcoin contracts expressed in BitML, by exploiting the property-preserving reduction of contracts to finite state systems proposed in Bartoletti and Zunino (2019). Since considering all the possible strategies of the adversaries may lead to an exponential number of states, alternative analysis techniques to pure model checking are a possible goal for future research.

Another research direction is that on formal models for participant's strategies. For instance, Laneve et al. (2019) specify both smart contracts and participants in a unified process calculus. Given a system containing a smart contract and all the involved participants, they devise a static analysis which evaluates the maximum profit of each participant.

## Route #4: Projecting Global Contracts to Participants' Strategies

Another inspiration for further research comes from the literature on concurrency multi-party contracts. There, the idea is that there is a global type which describes the overall behavior of a set of components, and we want to guarantee that, at run-time, the point-to-point interactions between these components respect the global type. One way to achieve this goal is to project the global type into a set of local types, and then verify that each component respects its local type (Honda et al., 2008).

We can transpose this approach to the realm of smart contracts, by imagining that the global type is a contract among a set of participants, and that the components are participant strategies. A crucial difference with the setting of concurrency contracts is that, in that of smart contracts, one can no longer assume participants to be honest, i.e., their run-time behavior may diverge from the local types against which they have been verified. One way to tackle this problem, inspired by earlier works on concurrency contracts (Bartoletti et al., 2012), is to make the projection construct only the strategies of the participants we are considering honest, guaranteeing that some desirable properties hold (e.g., that the honest participants never lose money), whatever the behavior of the other participants.

# 4. CONCLUSIONS

Current designs of smart contract languages seem to have been influenced more by marketing principles like "time-to-market," than by technical principles like reliability and security. This paper discusses some research directions to incorporate design by contract principles in the development of smart contracts.

To conclude, I would like to quote C.A.R. Hoare from his Turing Award Lecture (Hoare, 1981). In this lecture, Hoare was criticizing the design of the programming language PL/I, but I believe that his warning against a poor software design is always appropriate, also in the setting of smart contracts:

> "At first I hoped that such a technically unsound project would collapse but I soon realized it was doomed to success. Almost anything in software can be implemented, sold, and even used given enough determination. There is nothing a mere scientist can say that will stand against the flood of a hundred million dollars.

But there is one quality that cannot be purchased in this way–and that is reliability. The price of reliability is the pursuit of the utmost simplicity. It is a price which the very rich find most hard to pay."

## AUTHOR CONTRIBUTIONS

The author confirms being the sole contributor of this work and has approved it for publication.

## REFERENCES


Atzei, N., Bartoletti, M., and Cimoli, T. (2017). "A survey of attacks on Ethereum smart contracts (SoK)," in *Proc. POST, Vol. 10204 of* LNCS (Uppsala: Springer), 164–186.

Atzei, N., Bartoletti, M., Lande, S., Yoshida, N., and Zunino, R. (2019). "Developing secure Bitcoin contracts with BitML," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE* (Tallinn), 1124–1128.

Bartoletti, M., Cimoli, T., Murgia, M., Podda, A. S., and Pompianu, L. (2015a). "A contract-oriented middleware," in *Proc. Formal Aspects of Component Software, Vol. 9539 of LNCS* (Niterói: Springer), 86–104.

Bartoletti, M., Cimoli, T., and Zunino, R. (2015b). "Compliance in behavioural contracts: a brief survey," in *Programming Languages With Applications to Biology and Security, Vol. 9465 of LNCS* (Pisa: Springer), 103–121.

Bartoletti, M., Tuosto, E., and Zunino, R. (2012). "On the realizability of contracts in dishonest systems," in *Proc. COORDINATION* (Stockholm), 245–260.

Bartoletti, M., and Zunino, R. (2019). "Verifying liquidity of Bitcoin contracts," in *Proc. POST, Vol. 11426 of* LNCS (Prague: Springer), 222–247.

Beugnard, A., Jézéquel, J., and Plouzeau, N. (1999). Making components contract aware. *IEEE Comput.* 32, 38–45.

Bocchi, L., Honda, K., Tuosto, E., and Yoshida, N. (2010). "A theory of design-by-contract for distributed multiparty interactions," in *Proc. CONCUR, Vol. 6269 of* LNCS (Paris), 162–176.

Bordeaux, L., Salaün, G., Berardi, D., and Mecella, M. (2004). "When are two Web services compatible?," in *Proc. TES* (Toronto, ON), 15–28.

Castagna, G., Gesbert, N., and Padovani, L. (2009). A theory of contracts for Web services. *ACM Trans. Progr. Lang. Syst.* 31, 19:1–19:61. doi: 10.1145/1538917.1538920

Findler, R. B., and Felleisen, M. (2002). "Contracts for higher-order functions," in *ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA), 48–59.

Flatt, M., and PLT (2010). *Racket.* Technical Report Technical Report PLT-TR-2010-1, PLT Design Inc.

Hajdu Á., and Jovanovic D. (2020). "Solc-verify: a modular verifier for solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments. VSTTE 2019. Lecture Notes in Computer Science*, Vol. 12031, eds S. Chakraborty and J. Navas (Cham: Springer), 161–179. doi: 10.1007/978-3-030-41600-3_11

Harz, D., and Knottenbelt, W. J. (2018). Towards safer smart contracts: a survey of languages and verification methods. *CoRR* abs/1809.09805.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM* 12, 576–580.

Hoare, C. A. R. (1981). The emperor's old clothes. *Commun. ACM* 24, 75–83.

Honda, K. (1993). "Types for dyadic interaction," in *Proc. CONCUR* (Hildesheim), 509–523.

Honda, K., Yoshida, N., and Carbone, M. (2008). "Multiparty asynchronous session types," in *POPL* (San Francisco, CA).

Hüttel, H., Lanese, I., Vasconcelos, V. T., Caires, L., Carbone, M., Deniélou, P., et al. (2016). Foundations of session types and behavioural contracts. *ACM Comput. Surv.* 49, 3:1–3:36. doi: 10.1145/2873052

Laneve, C., Coen, C. S., and Veschetti, A. (2019). "On the prediction of smart contracts' behaviours," in *From Software Engineering to Formal Methods and Tools, and Back, Vol. 11865 of LNCS*, eds M. H. ter Beek, A. Fantechi, and L. Semini (Springer), 397–415.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016). "Making smart contracts smarter," in *ACM CCS* (Vienna), 254–269.

Meyer, B. (1986). *Design by Contract.* Technical Report Technical Report TR-EI-12/CO, Interactive Software Engineering Inc.

Meyer, B. (1991). *Eiffel: The Language*. Prentice-Hall.

Meyer, B. (1992). Applying "design by contract". *IEEE Comput.* 25, 40–51.

Miller, A., Cai, Z., and Jha, S. (2018). "Smart contracts and opportunities for formal methods," in *Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation Vol. 11247 of LNCS* (Limassol: Springer), 280–299.

Seijas, P. L., Nemish, A., Smith, D., and Thompson, S. J. (2020). "Marlowe: implementing and analysing financial contracts on blockchain," in *Financial Cryptography Workshops* (Kota Kinabalu).

Szabo, N. (1997). Formalizing and securing relationships on public networks. *First Monday* 2. doi: 10.5210/fm.v2i9.548

Tikhomirov, S. (2020). *A Curated Collection of Resources on Smart Contract Programming Languages.* Available online at: https://github.com/s-tikhomirov/smart-contract-languages


**Conflict of Interest:** The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.