Safe evolution of smart contracts

Augusto Sampaio[1], **Pedro Antonino**[2], **Juliandson Ferreira**[1],
Filipe Arruda[1], **A. W. Roscoe**[2,3,4]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Recife, Brazil

[2]The Blockhouse Technology Limited
Oxford, UK

[3]Department of Computer Science, University of Oxford
Oxford, UK

[4]University College Oxford Blockchain Research Centre
Oxford, UK

acas@cin.ufpe.br, pedro@tbtl.com, jef@cin.ufpe.br

fmca@cin.ufpe.br, awroscoe@gmail.com

Abstract. *We present a framework that supports the safe deployment and upgrade of smart contracts based on the design-by-contract paradigm. The starting point is an interface specification with invariants and pre- and postconditions for each function. The first deployed smart contract must conform to this specification. Specification evolution might involve both changing the data representation as well as extending the interface with new functions, provided the evolved specification is a refinement of the original one. Implementation evolution must conform to the corresponding specification. We report on the applicability of the framework in the verification of smart contracts that implement some Ethereum standards.*

1. Introduction

Smart contracts emerged to play the role of digital contracts on blockchains. As such, they implement consensual rules that are automatically triggered when some expected conditions are satisfied. In this way, smart contracts extend the blockchain initial capabilities by allowing users to program the processing of transactions. As classes in object orientation, a smart contract has its properties defined by attributes and behaviour defined by its functions. So the challenges to reason about smart contracts are similar to those of formal verification of object-oriented programs. Nevertheless, smart contracts have some additional features (like implicit parameters for senders and receivers, and persistent states in the blockchain) that make them even more complex to verify.

Needless to say, smart contracts impose a practical demand for verification strategies and tools since uncovered bugs have given rise to significant financial losses [Siegel 2016, Vollmer 2016]. Furthermore, once deployed, smart contracts are expected to be immutable. In practice, however, such contracts need to be eventually updated to fix bugs or to perform optimisations with the aim of reducing the execution cost (like saving gas in the execution of transactions in the Ethereum platform).

Breaking immutability can be minimised with solutions such as the adoption of the *proxy pattern* [OpenZeppelin 2018] in Ethereum. Using this pattern, the caller of a transaction always uses the (fixed) address of the proxy, which itself has a reference to the current contract it represents. In this way, when the contract needs to be upgraded, only the attribute value of the proxy that references the contract is changed, but the proxy's address invoked by callers remains the same. Yet, relying only on the proxy pattern does not ensure the preservation of the contract functional behavior after an upgrade.

We build on these ideas to develop a process and framework to ensure that only smart contracts that conform to their respective interface specifications can be deployed. Although, in principle, our approach is not inherently bound to any particular smart contract language or blockchain platform, we focus on Solidity and Ethereum.

In Section 2, we present the proposed framework. In Section 3 we present some case studies. Finally, in Section 4 we discuss related work, summarise our results and suggest some topics for future directions.

2. The proposed framework

As an initial step, the framework takes an interface specification (based on the *design-by-contract paradigm* [Meyer 1992]) and an implementation. The specification might have invariants, and each of its functions is annotated with pre- and postconditions that capture its behaviour. The first deployed smart contract must conform to this specification. Then we allow both the specification and the implementation to evolve.

Specification evolution might be in the form of changing the data representation (possibly changing attributes and their types) or via extending the interface with new functions. As a proof obligation, the evolved specification must be a refinement of the original one. Implementation evolution must always conform to the corresponding specification. In order to provide this flexibility of changing the data representation of a smart contract, the standard proxy pattern is not suitable, since a proxy holds a fixed state representation. We then adopt the proxy diamond pattern [Mudge 2020] to capture the updates in a flexible and safe way. We also rely on a *trusted deployer*, which is an off-chain service, to ensure safe evolutions of implementations and specifications.

In [Antonino et al. 2022] we presented an initial version of the verification framework where the specification is fixed for a given contract evolution history: an initial deployment or subsequent upgrade is allowed only if the implementation conforms to this specification. Our main current focus is to extend the framework in [Antonino et al. 2022] to allow the interface specification to change, provided the evolved version is a refinement of the previous one in the evolution history. In this way, we capture both data refinement and interface extension. We present here a small summary of this extension; the details will appear in [Antonino et al. 2023]. We are also investigating techniques to automatically verify contracts including loops, not addressed in [Antonino et al. 2022, Antonino et al. 2023].

The key element of the framework is a *Trusted Deployer* which ensures that upgrades entail only refined specifications and conforming implementations. It is implemented as an off-chain service. An overview of the Trusted Deployer is depicted in Figure 1. The core component is a *Verifier* whose interface includes the functions

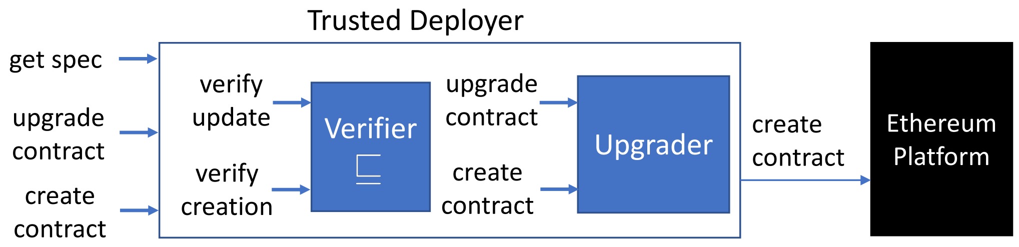

Figure 1. Trusted deployer architecture.

verify-creation and *verify-upgrade*. Another component is an *Upgrader* that includes the functions *create-contract* and *upgrade-contract*. Overall, these functions guarantee safe deployment and updates of smart contracts.

Upon receiving a request to create a new contract, via the function *create-contract* with a specification S and an implementation C as parameters, the Trusted Deployer invokes the function *verify-creation* of the Verifier. This boolean function yields true if C conforms to S, and false otherwise. If conformance holds, then the Deployer requests to the Upgrader that the contract be created in the Ethereum platform, and the pair (S, C) is stored as a reference for subsequent upgrades.

Similarly, when requested to upgrade a contract via the function *upgrade-contract*, both a specification S and a contract C are given as parameters. In this case, however, extra information must be provided: a relation (specifically, an abstraction function, say Af) and a function that we denote by (`init`). Since we allow the specification of a contract to change (both its representation and interface), we need a relation to establish the refinement of contracts with distinct data representations. Currently, this is provided by the developer, but we are working on a strategy to automatically infer such relations. The function `init` is a concretisation function, to transform abstract states into concrete ones. The need for this function is also a consequence of the flexibility of allowing specifications to change their data representation, but, particularly, due to the fact that a smart contract has a persistent state in a blockchain. Therefore, when a contract is upgraded by one with a different representation, we use this function to generate a new concrete representation from the abstract one. As in the case of contract creation, an upgrade requires that C conforms to S, and that S refines the currently deployed specification. Then the pair (S, C) becomes the current specification and implementation, and C is effectively deployed in the blockchain. Finally, the purpose of the *get-spec* function is to check whether a contract instance has been deployed by the Trusted Deployer and which specification it currently satisfies.

2.1. Specification refinement and implementation conformance

Formally, a specification can be characterised as follows.

Definition 1 *A specification is a triple (D, I, A) such that the* data representation *D is a finite sequence that defines the member variables of the specification, each of which with its unique name n_i and associated type t_i. The* interface *I is a finite mapping from function names to their signatures. A signature is a pair of sequences $(ins, outs)$ with the input and output parameters of the corresponding function. Each interface has a special function* *named* ***cons****, which stands for the constructor. The* annotation $A = (inv, Pre, Post)$ *is a triple containing the contract invariant, preconditions, and postconditions.*

The semantics of a specification is given by a labelled transition system (LTS) where the states capture the values of the contract variables and the transitions are labelled by function calls that must obey the classical semantics of the associated contract annotations. The initial state ensures that only constructors can be called. Another relevant aspect is that if a precondition is not met, a transition in the LTS can lead to an arbitrary state, except for the initial state. The specification refinement relation we define is based on conventional notions of program refinement [Morgan 1994] and behavioural subtyping [Liskov and Wing 1994]: it allows for invariant and postcondition strengthening whereas preconditions can be weakened. Our definition is intentionally flexible concerning specification evolution. It allows for simple strengthening of postconditions/invariant and weakening of preconditions, but it also enables more complex evolutions such as data-representation changes and interface extensions.

Similarly, we formalise the notion of conformance of a smart contract with respect to a specification. Informally, the contract must declare the same functions with matching signatures, and the transition relation in the induced LTS of the implementation must be a subset of that of the specification. The intuition is that the behaviours of the implementation are restricted to those allowed by the specification.

We use solc-verify [Hajdu and Jovanović 2020] to check whether a solidity contract obeys its design contract (including pre-, postconditions and invariants). Additionally, we use solc-verify to mechanise the verification of specification refinement and implementation conformance, by encoding the associated proof obligations as annotated contracts in Solidity.

3. Case Studies

First we consider the simpler scenario of smart contract evolution that must obey a fixed specification. We consider the ERC20, ERC3156, and ERC1155 Ethereum standards. Our framework was able to identify errors in the following categories: Integer Overflow and Underflow (IOU); Non Specification Conformance (NSC), when a function does not meet a specific mandatory requirement defined in its ERC specification; Nonstandard Token Interface (NTI), when the contract does not meet the syntactic restriction defined by the standard; wrong operator (WOP), for instance, when the $<$ operator would be expected but $\leq$ is used instead; and Verification Error (VRE), when the verification process cannot be completed or the results were inconclusive. It also established conformance for some of the samples analysed. Table 1.a shows the results we obtained for the ERC20 standard. Similar flaws were identified for ERC1155 and ERC3156. Scenarios involving data refinement and interface extension are addressed in Table 1.b.

4. Conclusion

The main contribution of this work is an automated framework to ensure that smart contracts are deployed and upgraded in a safe way, in the sense of conforming to a reference specification. Considering a typical development history, the first step the deployment of a reference specification and an implementation that is checked to conform to the specification. Progressively, the specification can evolve, provided it is

Table 1. Framework results

a. Simple Evolutions (ERC20)

Repository	Commit	Time	Output	Repository	Commit	Time	Output
0xMonorepo	548fda	2.85s	WOP	Uniswap	e382d7	3.57s	IOU
DigixDao	5aee64	7.60s	NTI	Uniswap	55ae25	3.43s	WOP
DsToken	08412f	4.14s	WOP	SkinCoin	25db99	0.99s	NTI
Klenergy	60263d	1.70s	VRE	SkinCoin	27c298	1.94s	NTI

b. Changes of Data Representation and Interface Extension (ERC20, ERC721)

Repository	Commit	Time	Output	Repository	Commit	Time	Output
DigixDao	0a9709	6.07s	IOU	Token ERC20	44c3a1	5.33s	WOP
DigixDao	0550e8	7.10s	IOU	OpenZeppelin ERC721	b7d60f	7.43s	IOU

a refinement of the previous one in the evolution chain, and the implementation can be upgraded, provided it conforms to the associated specification. Furthermore, we address the consistent migration of the persistent storage of smart contracts in a blockchain, throughout the evolution history. We have also evaluated the proposed framework considering real smart contracts available in several repositories. We have found bugs in implementations of ERC20, ERC1155, ERC 3156 and ERC721.

Concerning related approaches, there is much discussion on the need for systematic support for the safe evolution of smart contracts (see, for instance, [Tolmach et al. 2021]) as well as the use of design patterns that partially address the problem [Lu 2018]. However, there are only a couple of approaches close-related to ours. Due to the lack of space, we consider two of these works. The work in [Dickerson et al. 2018] proposes an on-chain solution based on the use of special contracts that carry a proof that they conform to their specifications. Unlike our approach, this requires significant changes in the smart contract platforms themselves; also, being on-chain makes the use of such verification methods more difficult since these methods would slow down consensus. In [Rodler et al. 2021], the authors propose a mechanism to upgrade contracts in Ethereum that works at the EVM-bytecode level. Their framework takes vulnerability reports issued by the community as an input and tries to patch affected deployed contracts automatically using patch templates. While they simply test patches that are reactively generated based on vulnerability reports, we require the user to provide a specification (and possibly an abstraction function) of the expected behaviour of a contract and carry out formal and automated reasoning. Furthermore, it is worth noting that while we define the semantics in terms of LTSs and use the solc-verify as an underlying tool for verification, alternative strategies and notations exist. For instance, [Patrick et al. 2021] discuss Propositional Dynamic Logic as a formal system designed to reason about smart contracts. As far as we are aware, the systematic treatment of a safe evolution of smart contracts in blockchains that we propose is neither supported by Ethereum nor by any other deployment framework.

Despite the promising results, there are some opportunities for improvement. Currently, we are restricted to partial correctness notions. In ongoing work, we are considering termination so that we can fully reason with smart contracts containing loops. The current implementation requires that the developer provides the abstraction function when a specification is upgraded. We are looking at approaches, based on model learning, to automatically infer such relations. We are also working on the automatic generation of the pre-, postconditions, and invariants from the textual descriptions of the token standards, using natural language processing techniques.

References

Antonino, P., Ferreira, J., Sampaio, A., and Roscoe, A. W. (2022). Specification is law: Safe creation and upgrade of ethereum smart contracts. In Schlingloff, B. and Chai, M., editors, *Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings*, volume 13550 of *Lecture Notes in Computer Science*, pages 227–243. Springer.

Antonino, P., Ferreira, J., Sampaio, A., Roscoe, A. W., and Arruda, F. (2023). A refinement-based approach to safe smart contract deployment and evolution. *SoSyM—International Journal on Software and Systems Modeling*. Submitted.

Dickerson, T. D., Gazzillo, P., Herlihy, M., Saraph, V., and Koskinen, E. (2018). Proof-carrying smart contracts. In *Financial Cryptography Workshops*.

Hajdu, Á. and Jovanović, D. (2020). solc-verify: A modular verifier for solidity smart contracts. In *VSTTE*, pages 161–179. Springer.

Liskov, B. H. and Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841.

Lu, A. (2018). Solidity DelegateProxy Contracts. `https://medium.com/gnosis-pm/solidity-delegateproxy-contracts-e09957d0f201`.

Meyer, B. (1992). Applying 'design by contract'. *Computer*, 25(10):40–51.

Morgan, C. (1994). *Programming from Specifications (2nd Ed.)*. Prentice Hall International (UK) Ltd., GBR.

Mudge, N. (2020). EIP-2535: Diamonds, Multi-Facet Proxy. `https://eips.ethereum.org/EIPS/eip-2535`.

OpenZeppelin (2018). Proxy Patterns. `https://blog.openzeppelin.com/proxy-patterns` accessed on 30 May 2023.

Patrick, A., Coelho, I. M., and Lopes, B. (2021). Automatic program verification in dynamic logic with applications to smart contracts. In *Anais do II Workshop Brasileiro de Lógica*, pages 1–8. SBC.

Rodler, M., Li, W., Karame, G. O., and Davi, L. (2021). Evmpatch: Timely and automated patching of ethereum smart contracts. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1289–1306. USENIX Association.

Siegel, D. (2016). Understanding the dao attack. `https://www.coindesk.com/understanding-dao-hack-journalists` accessed on 22 July 2021.

Tolmach, P., Li, Y., Lin, S.-W., Liu, Y., and Li, Z. (2021). A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)*, 54(7).

Vollmer, J. (2016). The biggest hacker whodunnit of the summer. `https://www.vice.com/en/article/pgkzqm/the-biggest-hacker-whodunnit-of-the-summer` accessed on 22 July 2021.