Toward an Integrated Process Model for Smart Contract Engineering

Conference Paper · December 2018

CITATIONS
0

READS
155

5 authors, including:

Bernhard Waltl
Technische Universität München
18 PUBLICATIONS **35** CITATIONS
SEE PROFILE

Horst Treiblmaier
MODUL University Vienna
114 PUBLICATIONS **764** CITATIONS
SEE PROFILE

Ulrich Gallersdörfer
Technische Universität München
4 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Toward an Integrated Process Model for Smart Contract Engineering

Christian Sillaber
christian.sillaber@acm.org
Zicklin Center at Wharton
Pennsylvania, USA

Bernhard Waltl
b.waltl@tum.de
Technical University Munich
Dept. of Information Systems
Munich, Germany

Horst Treiblmaier
horst.treiblmaier@modul.ac.at
MODUL University Vienna
Dept. of Int. Management
Vienna, Austria

Ulrich Gallerdörfer
ulrich.gallersdoerfer@tum.de
Technical University Munich
Department of Informatics
Munich, Germany

Michael Felderer
michael.felderer@uibk.ac.at
University of Innsbruck
Department of Computer Science
Innsbruck, Austria

Abstract

Engineering smart contracts for trustless, append-only, and decentralized digital ledgers allows mutually distrustful parties to transform legal requirements into immutable and formalized rules. We present an integrated process model for engineering blockchain-based smart contracts, which explicitly accounts for the immutability of the trustless, append-only, and decentralized digital ledger ecosystem and overcomes several limitations of traditional software engineering process models. Applying such a model when engineering smart contracts will help software engineers and developers to streamline and better understand the overall engineering process of decentralized digital ledgers in general and the blockchain in particular.

Keywords: Smart Contract, Development Process Model, Software Engineering, Blockchain, Distributed Ledger Technology

Introduction

Blockchain technology and distributed ledger technology (DLT) has recently gained a lot of attention in the IS community. The immutable, trustless model of decentralized computation and transaction handling that is provided by the blockchain strives to ensure fairness for all participating parties. The huge amount of monetary value involved increases the demand for structured software development processes and quality assurance. Highly publicized incidents such as the DAO attack illustrate that (1)

the ad-hoc style of engineering is not suitable for such high value transactions, (2) current software engineering approaches do not ensure a sufficient level software quality, and (3) these approaches are either unsuitable or misaligned for the idiosyncrasies of blockchain technology (Atzei et al., 2016).

Traditional software engineering focuses on principles for developing high-quality software systems and maintaining the systems as they evolve in real-world environments. Software that does not evolve will not be able to keep up with changing requirements and will become outdated over time. This has profound implications for existing software process models. They address the increasing need for change and evolution by introducing iterative, incremental, and evolutionary approaches (Beck et al., 2001). Post-deployment changes are typically realized by re-entering the regular development activities, which eventually result in a new version or a patch that is released during scheduled down-times. This process is structured by change management activities (Stark, 2015). All changes that happen after the development time are no longer possible when smart contracts are published in a DLT environment and become immutable. In this extended abstract, we develop a smart contract engineering process that clearly outlines the different elements and artifacts. It can serve as a tool for various strategic and operational activities since it helps in defining priorities, managing risks, expectations and time frames.

Related Work

Although the term blockchain is relatively new, its underlying concepts are not and have been studied extensively from different perspectives. Smart contract engineering is built on the technological foundation of smart contracts as well as on state-of-the-art software engineering with a focus on blockchain technology. The term smart contracts was coined by Nick Szabo (1997) and describes how the computer-based execution of contracts between two parties can be secured without a third party. In a DLT context, the correct execution is enforced among other mechanisms, by a so-called consensus protocol.

The IEEE 1074-1995 Standard defines a process as a set of steps that can be executed in a certain predefined sequential, parallel, or conditional order (IEEE, 1995). Various process models cover the order and frequency of phases in software projects. Those phases typically include planning, analysis, design, implementation, testing, and maintenance. Waterfall models progress sequentially through these phases, iterative models are typified by repeated execution of the waterfall phases, in whole or in part (Braude and Bernstein, 2016). Other than these phase-oriented process models, agile process models are based on the principles of individuals and interaction, working software, customer collaboration as well as fast response to change (Beck et al., 2001; Lee and Xia, 2010). A more recent trend is to combine phase-oriented with agile process models to obtain hybrid software engineering process models. Modern software engineering approaches rely heavily on the (re-)use of software patterns. Patterns are collections of abstract best practices of software code that engineers can easily adapt. These best practices are the result of previous software engineering experience and allow faster, more secure, and more reliable software development.

Process Model Development

Methodology

We conducted interviews on smart contract development with eleven industry experts. The primary goal of the interviews was to get a better understanding of how the study participants develop smart contracts and which processes, artifacts, and tools they apply. We used a Delphi study approach and the findings from the first round were evaluated and refined in a second round. Each interview was recorded and transcribed. Based on our findings, we develop the smart contract engineering process in a stepwise manner. First, we discuss the conceptual base and describe the main types of artifacts that emerged from the interviews. Second, we integrate the findings from the qualitative interviews with several software developers. Third, we discuss various roles, activities, and artifacts and, fourth, we incorporate these components into one integrative model.

Conceptual Base

We followed a design science approach to precisely define the respective steps of the process model. March and Smith (1995) differentiate between four types of artifacts, namely constructs, models, methods, and instantiations. Constructs, which consist of language and vocabulary specifying problems and solutions, form the baseline design science vocabulary. They specify the general entities including the attributes and relationships among each other. Models are descriptions and representations of real-world phenomena with a focus on utility. The steps needed to execute a specific process are called methods, which are procedures solving problems and developing solutions. Instantiations are the realizations of artifacts within their respective environments. The mapping of these artifacts to the domain of smart contract engineering is shown in Table 1.

Table 1. Mapping of Artifacts

Table 1. Mapping of Artifacts			
Artifact	**Manifestation**	**Artifact**	**Manifestation**
Construct	• Trustless, append-only, decentralized, digital ledgers (TADDL) • Cryptocurrency assets • Smart contract execution engine • Smart contract expression language • Actors • Wallets	Method	• Smart contract engineering (sub-) activities • Iterations of the engineering process • Simulation activities • Test methods for smart contracts
Model	• Smart contract code, templates and patterns • Transaction schemes • Digital representation of assets • Consensus and reward algorithms • Interactions via transactions, function calls, oracle inputs	Instantiation	• Instance of the smart contract engineering process with its activities • Operationalized smart contracts • Results from smart contract test scenarios • Results from smart contract executions and simulations

Roles, Activities, and Artifacts

The Rational Unified Process (RUP) can be used to formally describe an engineering process. It is document-centric and reflects the smart contract development process. It can be used to differentiate between three distinctive elements: First, roles pertain to individuals or groups performing activities of the process. Second, activities summarize a unit of work that must be performed. The outcome results in the creation or update of artifacts, which can be concepts, smart contract code, or performance reports. Third, artifacts denote the input and output of activities. Artifacts are created, modified, and used by the roles during the procedure and are either the final product, parts of it, or intermediate results.

Smart contract lifecycles start with an implementation phase during which requirements are transformed into an implementation, verified against the requirements, and either approved for release or modified again (Sillaber and Waltl, 2017). Once the smart contract is approved, it is published on the TADDL in the submission stage. In this phase, the smart contract is submitted and distributed within the TADDL network. From now on, every entity with access to the TADDL can retrieve the contract and share it with other nodes. Once the smart contract has been spread throughout the network and is accepted by general consensus (i.e., it persists on the network), reverting or changing the contract requires high effort. The contract is now ready to be executed. The execution stage of smart contracts is performed by miners or other participants of the TADDL, since the smart contract code is now accessible for all participants. The smart contract is retrieved from the TADDL and executed by the respective node. Based on a given input, the output (e.g., a return value, a state transition, or a set of transactions) of a smart contract is computed, which is then stored and distributed within the network. In the finalization stage the smart contract expires. This can happen

either because the parties actively declare the smart contract as invalid or because of intrinsic conditions that make further executions impossible (e.g., time expiration).

An Integrated Smart Contract Engineering Process

Figure 1 combines and summarizes all our findings and shows the integrated smart contract engineering process. In the conceptualization phase the preliminary scope and the goals of the smart contract are defined. All involved parties agree on what will and what will not be part of the contract and can be directly derived from traditional contractual requirements. The problem definition should also state the desired economic outcome(s). In the next phase, the conceptual model is created. The conceptual model defines classes of objects (e.g., wallets) and the desired relations between these objects and outcomes (e.g., transactions). The construction of the conceptual model will most likely uncover incomplete and contradictory aspects of the problem definition. Additionally, the modeling process may raise new questions for the involved parties to answer and resolve through negotiation. In either case, the problem definition should be adjusted.

After the conceptual modeling phase, the implementation phase starts. Here, the conceptual model is mapped onto an executable model as existing smart contract patterns are identified, adapted and combined. An executable smart contract is not necessarily immediately correct and has to be reviewed, tested, and verified. Verification and simulation of the smart contract against the scope and stakeholder requirements are necessary to check whether the code contains errors, including programming errors and wrong parameters. For verification purposes ("Simulation, testing, code review"), various scenario-based executions can be simulated step-by-step in a private blockchain. Apart from verification, validation of the smart contract is also required.

Starting from the consolidated and validated smart contract, an instance of the smart contract can be frozen and submitted for execution in the live environment. Finally, in the approval and execution phases, the published smart contract is approved by the parties and executed in the TADDL and has to be monitored during runtime. In case its behavior deviates from the requirements, change management mechanisms have to be activated – in extreme cases the deactivation of the smart contract – and a new smart contract hast to be created to better meet the stakeholders' requirements. Although the smart contract becomes immutable after it has been submitted to the TADDL environment, the environment itself often provides capabilities to influence the outcome of smart contracts (e.g., through a function registry or call delegation). The smart contract's runtime behavior is constantly monitored and managed in a change management process. Once the smart contract has reached the end of its life (e.g., by executing the "self-destruction" operation in the Ethereum blockchain), proper finalization can be confirmed in the finalization phase by validating whether the desired outcomes have been reached. In practice, many phases will overlap.

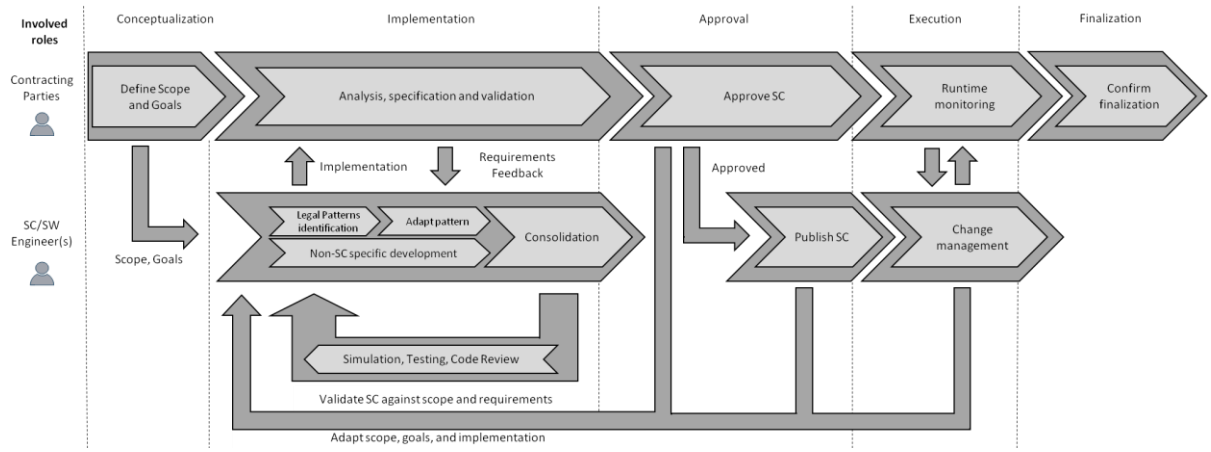

Figure 1. Integrated Smart Contracts Engineering Process. (SC … Smart Contract)

Discussion, Implications, and Limitations

The integrated process model for smart contracts can help to improve the quality of smart contracts. This is crucial since immutable bugs in smart contracts have been exploited in previous attacks. Our proposed smart contract engineering process is generic and is applicable to a wide variety of distributed ledger technologies. It is based on traditional software engineering process models and methodologies that have been successfully applied in a wide variety of use cases. While the analysis, design, and implementation phases align with our proposed conceptualization and implementation phases, special care has to be given to the testing phase, which has to be conducted and finished before publishing the smart contract. Iterative software engineering process models typically iterate sequentially through the aforementioned four phases. The implementation phase proposed in this paper iterates through a pattern selection and adaption, development, consolidation, review, testing, and simulation phase, aligning this process activity with iterative software engineering process models. The integrated model can immediately be applied in real-world industry settings. It can improve applied smart contracts engineering processes and serve as a basis for critically investigating such processes in great detail, which is of practical value for any industry that needs to react fast while at the same time ensuring sufficient software quality. We see two major limitations of this research which deserve further attention. First, there are no validated measurements for the concepts of the process activities. Second, due to a lack of established best practices in smart contracts engineering, an empirical evaluation of the hypothesized artifacts was not feasible at the time of publication.

Conclusion and Future Research

In this paper, we develop an integrative process model for smart contract engineering and describe its activities, roles and artifacts. We argue that conventional software engineering process models do not provide adequate support for the trustless, append-only, and decentralized environment in which smart contracts are executed. Traditional process models do not account for the immutability of smart contracts after they are submitted because they assume a (mostly) frictionless transition between software releases and allow for modifications of existing software releases. Our smart contract engineering process accounts for these peculiarities of blockchain-based software development and consists of five sequential phases: conceptualization, implementation, approval, execution and finalization. These phases result from the properties of the underlying blockchain ecosystem. Future research needs to investigate if and how the engineering process model can be tailored to different software engineering methodologies (e.g., Scrum, V-model). Furthermore, it is necessary to integrate this framework with existing work on testing and quality assurance in software engineering.

References

Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A survey of attacks on ethereum smart contracts. In Principles of Security and Trust (pp. 164-186). Springer, Berlin, Heidelberg.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D. 2001. "Manifesto for Agile Software Development," http://agilemanifesto.org/, accessed April 20, 2018.

Braude, E. J. and Bernstein, M. E. 2016. *Software Engineering*. Long Grove, IL: Waveland Press.

IEEE. 1995. 1074-1995 - IEEE Standard for Developing Software Life Cycle Processes, IEEE, https://ieeexplore.ieee.org/document/490501/, accessed March 20, 2018.

Lee, G. and Xia, W. 2010. "Toward Agile: An Integrated Analysis of Quantitative and Qualitative Field Data on Software Development Agility, " *MIS Quarterly* (34:1), pp. 87-114.

March, S. T. and Smith, G. F. 1995. "Design and Natural Science Research on Information Technology," *Decision Support Systems* (15:4), pp. 251–266.

Sillaber, C. and Waltl, B. 2017. "The Life Cycle of Smart Contracts in Blockchain Ecosystems," *Datenschutz und Datensicherheit – DuD* (41:8), pp. 497-500.

Stark, J. 2015. *Product Lifecycle Management*, London: Springer.

Szabo, N. 1997. The Idea of Smart Contracts. Nick Szabo's Papers and Concise Tutorials. http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html, accessed May 1, 2018.