A Graph-Based Framework for Analysing the Design of Smart Contracts

Bram Vandenbogaerde
Vrije Universiteit Brussel
Belgium
bram.vandenbogaerde@vub.be

ABSTRACT

Used as a platform for executing smart contracts, Blockchain technology has yielded new programming languages. We propose a graph-based framework for computing software design metrics for the Solidity programming language, and use this framework in a preliminary study on 505 smart contracts mined from GitHub. The results show that most of the smart contracts are rather straightforward from an objected-oriented point of view and that new design metrics specific to smart contracts should be developed.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Smart Contracts, Metrics, Mining Software Repositories

ACM Reference Format:
Bram Vandenbogaerde. 2019. A Graph-Based Framework for Analysing the Design of Smart Contracts. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3338906.3342495

1 RESEARCH PROBLEM AND MOTIVATION

Originally limited to the infrastructure for cryptocurrencies, blockchain platforms nowadays enable executing and recording smart contracts. These protocols were formally coined as a digital way to facilitate, verify, or enforce the negotiation or performance of contracts [7] and are currently used to guarantee transactions between multiple parties that would otherwise require a trusted man-in-the-middle. As valuable assets are often involved, most existing tool support for developing smart contracts (*e.g.*, [1, 3, 8, 10]) focuses on finding security vulnerabilities and weaknesses. Although these contracts follow an object-oriented design and their code is becoming increasingly complex, there is little tool support for evaluating and improving their maintainability. For object-oriented software, software design metrics [4] have been proposed that can be used to assess the quality of an implementation. Tonelli et al. [9] have studied some initial size-related metrics for smart contracts implemented in Solidity, a popular object-oriented language for implementing smart contracts executed on the Ethereum blockchain platform. However, the study does not include software design metrics.

We propose a new framework for computing object-oriented software design metrics about Solidity smart contracts. To represent these contracts, we employ a graph-based semantic meta-model inspired by Mens and Lanza [5]. We use the framework in a preliminary analysis of object-oriented design metrics for contracts mined from GitHub projects.

2 A GRAPH-BASED FRAMEWORK FOR ANALYSING SMART CONTRACTS

We introduce a framework for computing software design metrics for Solidity smart contracts. The framework uses a three-step process for its computation: (1) parsing the Solidity source code into abstract syntax trees (ASTs); (2) analysis of the ASTs to create meta-model instances; (3) computation of the software metrics from these instances. The framework parses the code using and an ANTLR4[1]-generated parser, and uses visitors on the produced ASTs for a semantic analysis to resolve types and call relations.

The resulting semantic, language-agnostic[2], meta-model is graph-based, which enables the use of declarative graph queries to compute design metrics and the use of a graph database for persistence. Figure 1 depicts this meta-model, while fig. 2 depicts an instance of the meta-model extracted from the contracts listed in listing 1. The meta-model defines its entities as nodes in the graph, and their relationships as edges. It stores containment as well as semantic information such as where types are declared and where functions are called from.

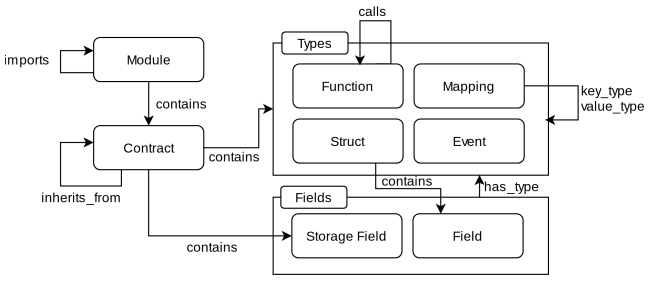

Figure 1: Solidity Smart Contracts Meta-Model

[1]https://www.antlr.org/
[2]Currently only Solidity is supported.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08.
https://doi.org/10.1145/3338906.3342495

```
pragma solidity ^0.5.0;

contract account {
   uint balance = 0;
   mapping (address => uint) owners;

   modifier onlyowners() { /* ... */ _; }

   function deposit() payable external { /* ... */ }
   function withdraw(address receiver, uint amount) external;
}

contract multisigaccount is account {
   function withdraw(address receiver, uint amount) external {/* ... */}
}

contract bank {
   account[] wallets;

   function withdraw(uint walletid, address receiver, uint amount) payable
   ↪ external {
      wallets[walletid].withdraw(receiver, amount);
   }
}
```

Listing 1: An example of Solidity code for a banking system

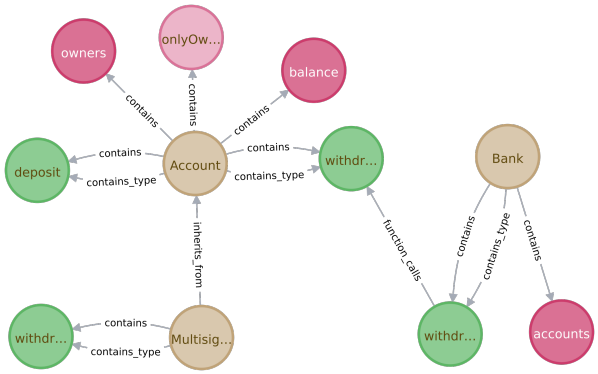

Figure 2: Meta-model instance of Listing 1.

The framework is implemented in Scala, and currently relies on the Neo4J graph database[3] for persisting the semantic meta-models. In the interest of the framework's generality, we do not use the Cypher graph query language provided by Neo4J. Instead, we opted for Gremlin [6] in which queries express path traversals through a graph. We used Scala's support for `implicit` classes to extend this language with domain-specific operators. The following example shows these operators in a query that extracts all functions from the *Account* contract depicted in Figure 2:

```
g.V().contract("Account").functions()
```

Our framework uses similar queries to compute software design metrics from the graph-based semantic meta-model. The following example shows how to calculate the number of function calls for all contracts in a project:

```
g.V().contract().functions().isCalled().count()
```

3 A PRELIMINARY EMPIRICAL STUDY ON THE QUALITY OF SMART CONTRACTS

To evaluate the overall design of Solidity smart contracts, we used our framework on 505 smart contracts extracted from 83 GitHub repositories. We started from the 1,917 repositories on GitHub that are written in Solidity, and narrowed the corpus down to 313 repositories that contain a Truffle[4] project, from which we discarded those projects that did not compile or that could not be analysed by our current implementation. The final corpus consists of 83 Truffle projects and 505 Solidity smart contracts. Table 1 provides an overview of the analysed projects, along with descriptive statistics for their design metrics.

Table 1: An overview of the calculated metrics

	Mean	Std. Dev	Median	Min	Max
Cyclomatic complexity	4.54	5.72	3.0	0.0	25.0
# of Lines	329.87	303.40	225.0	23.0	1712.0
# of Functions	26.78	24.08	18.0	2.0	142.0
# of Contracts	6.08	4.86	4.0	1.0	29.0
# of Modules	6.07	6.28	5.0	2.0	30.0
Fanout	0.91	1.49	0.0	0.0	7.0
Calls	5.65	6.09	3.0	0.0	28.0
Average Depth of Inheritance Tree	1.66	1.82	2.0	0.0	6.0
Average # of Children per contract	0.54	0.52	1.0	0.0	2.0
# of Methods / # of Contracts	4.51	2.45	4.0	1.0	12.5
# of Contracts / # of Modules	1.01	0.67	0.0	0.5	4.5
# Fanout / Calls	0.11	0.16	0.0	0.0	0.75
# Calls / # of Functions	0.22	0.12	0.22	0.0	0.55
Cyclomatic Complexity / # of Lines	0.01	0.01	0.0	0.0	0.04
# of Lines / # of Methods	15.92	22.15	11.85	4.5	203.66

Table 1 provides an overview of the analysed projects, along with descriptive statistics for their design metrics.

The metrics show that public Solidity smart contract projects are relatively small. Usually a module contains exactly one contract, which is unlike the Scala but similar to the Java practice of defining one class per file.

The *Fanout* metric, which indicates the number of distinct contracts that are called from other contracts, is also relatively low. The number of calls to other functions is, moreover, low compared to the number of functions in a contract. This might be counter-intuitive at first, as uncalled methods can be indications of dead code and are therefore discouraged in traditional software systems. However, we argue that this is quite commonplace in Solidity smart contracts, as the functions are designed to be mostly called from external applications.

Finally, the inheritance mechanisms that Solidity provides seems to be underutilized as inheritance trees are not deep, and on average a contract does not have many children.

4 CONTRIBUTIONS AND FUTURE WORK

In this paper, we presented a graph-based framework for representing, querying, and analysing Solidity smart contracts. Its contract representation is inspired by the work of Mens and Lanza [5]. Using this framework, we transposed well-known design metrics for object-oriented programs [4] to Solidity smart contracts, and evaluated them empirically on a corpus of projects mined from GitHub.

The framework can be used to implement additional tooling for Solidity smart contracts based on software metrics, such as the automatic detection and refactoring of bad smells [2]. In future work, we plan to develop design metrics that are specific to smart contracts as well as to study the similarities and differences between bad smells in smart contracts and object-oriented software.

[3] https://neo4j.com/

[4] Truffle is a collection of command-line tools for developing, testing, deploying and managing Solidity smart contracts and their dependencies.

REFERENCES

[1] Santiago Bragagnolo, Henrique Rocha, Marcus Denker, and Stéphane Ducasse. 2018. SmartInspect: solidity smart contract inspector. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 9–18.

[2] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[3] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[4] Michele Lanza and Radu Marinescu. 2007. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.

[5] Tom Mens and Michele Lanza. 2002. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science* 72, 2 (2002), 57–68.

[6] Marko A Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. ACM, 1–10.

[7] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First Monday* 2, 9 (1997).

[8] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 9–16.

[9] Roberto Tonelli, Giuseppe Destefanis, Michele Marchesi, and Marco Ortu. 2018. Smart contracts software metrics: a first study. *arXiv preprint arXiv:1802.01517* (2018).

[10] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 67–82.