University of Groningen

A framework for creating deployable smart contracts for non-fungible tokens on the Ethereum blockchain

Chirtoaca, Dan; Ellul, Joshua; Azzopardi, George

Published in:
IEEE

DOI:
[10.1109/DAPPS49028.2020.00012](https://doi.org/10.1109/DAPPS49028.2020.00012)

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2020

[Link to publication in University of Groningen/UMCG research database](#)

Citation for published version (APA):

Chirtoaca, D., Ellul, J., & Azzopardi, G. (2020). A framework for creating deployable smart contracts for non-fungible tokens on the Ethereum blockchain. In *IEEE* (pp. 100-105). IEEE.
<https://doi.org/10.1109/DAPPS49028.2020.00012>

Copyright

Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: <https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment>.

Take-down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): <http://www.rug.nl/research/portal>. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

A framework for creating deployable smart contracts for non-fungible tokens on the Ethereum blockchain

Dan Chirtoaca
University of Groningen

Joshua Ellul
University of Malta

George Azzopardi
University of Groningen

Abstract—Non-fungible tokens are an up and coming application domain for smart contracts. Ethereum is the first blockchain-based decentralized computing platform that has standardized this type of tokens into a well-defined interface, namely ERC721. We propose a framework that provides developers with a smart contract suite that offers complete implementations of the ERC721 standard and common extensions and features frequently encountered in ERC721-based applications. We introduce a specification language that enables customization and configuration of the smart contract suite by including and excluding the supported features and extensions. We evaluate the smart contract suite for its extensibility and reusability and compare the metrics with four reference implementations tackling a similar problem. In addition to this, we evaluate and analyze the effort and efficiency of the specification language in comparison to manual configuration of the smart contract suite. Our contribution lies in examining quality metrics for code extensibility and reusability and determining the more insightful metrics for assessing these quality attributes in the context of Solidity smart contracts. Additionally, from the lines of code metric, We conclude that our specification language offers a simple and efficient alternative to manual smart contract suite customization.

Keywords: ERC721, Non-fungible tokens, smart contracts, Ethereum, specification language

1. Introduction

The idea behind smart contracts was introduced by Nick Szabo [1] and following the original definition, a smart contract represents a set of promises, specified in digital form, including protocols within which the parties perform on these promises. Smart contracts have been implemented on blockchains and are used mostly as a general purpose computation that takes place on the blockchain (i.e. sets of actions on which parties agree to be executed). This enables decentralized computational logic which allowed for extending the application of Blockchain Technology to more than just cryptocurrencies [2]. Ethereum [3] is a global, open-source platform for decentralized applications. These applications are enabled through smart contracts. It is one of the most popular platforms when it comes to developing new blockchain based applications. The introduction and adoption of smart contract interface standards have aided

the development process of decentralized applications. This has facilitated a consistent application interaction as well as creating, managing and transferring tokens in a universal and uniform way. ERC20 [4] and ERC721 [5] are two of the most popular established token standards. The former is the initial token standard on Ethereum that provides basic functionality to transfer and manage fungible tokens. Since ERC20 tokens are fungible, they can be sub-divided. The latter standard has been officially accepted only recently and it has opened a new range of application opportunities.

The ERC721 standard introduced the concept of non-fungible tokens. Non-fungible tokens (NFTs) are a special kind of token that allows them to identify something unique, thus becoming non-interchangeable, unlike ERC20 tokens and other cryptocurrencies which are fungible. The standard is well documented and presents itself with detailed semantics about its functions by elaborating on preconditions and post-conditions. It does not, however, address nor impose any implementation details. These are left at the developer's discretion and requires a comprehensive understanding of smart contract programming in Solidity to ensure the quality of the smart contracts. Moreover, in order to effectively use non-fungible tokens, the contracts necessary for these applications generally require more than the basic ERC721 interface standard. Plenty of features and extensions to the standard (e.g. token creation and destruction, contract access control, etc.) that are necessary for different use cases are not part of the standard. These have to be developed and integrated separately. There exist libraries that provide bare bones implementations of the ERC721 standard, but the above mentioned issues are only partly addressed in available frameworks and platforms, which provide a limited/incomplete set of features/extensions and only help with minimal configuration of the underlying smart contracts.

We propose a framework, NFTY (which stands for Non-Fungible Tokens made easY), that provides a smart contract suite of implemented smart contracts for both the ERC721 standard and the identified frequently used extensions and features. Additionally, we propose a *Specification Language* (SL) that allows to perform smart contract coupling (i.e. smart contract suite configuration) without the need for in-depth Solidity knowledge. In other words, the SL provides a simple abstraction which increases the efficiency in the process of configuration and customization of the smart contract suite. This can help developers speed-up the development

time as well as obtain a more consistent implementation of the ERC721 standard. Moreover, the SL of the framework wraps and abstracts most of the complex smart contract logic and it should help in reducing the effort required to start developing an ERC721 compliant smart contract. We synthesize the objectives and results of this research through the following research questions:

- RQ1** What are currently the most popular application domains for NFT based (ERC721 compliant) decentralized applications on the Ethereum Blockchain?
- RQ2** Which are the features and extensions used in conjunction with the ERC721 standard that are frequently encountered in NFT based smart contracts on the Ethereum Blockchain?
- RQ3** Would establishing a framework that supports the identified feature set have a positive impact on the efficiency with which developers can configure the underlying smart contracts?
- RQ4** Would this framework offer developers easily extensible and highly reusable smart contract code?

This paper is organized as follows. In Section 2 we elaborate on the notions on which we base our work and present 4 reference implementations that tackle a similar problem in the area of NFTs. In Section 3 we showcase our approach to solving the identified issues. In Section 4 we evaluate our smart contract suite and the reference platforms as well as assess the SL. In Section 5 we discuss the results and draw the conclusions with respect to this work.

2. Background and Related Work

The addition of smart contracts to the Blockchain Technology [6] enables general purpose computations to be run on it. Smart contracts are well-defined programs located on the blockchain. They implement functions that can be executed on-demand, thus interacting with the data structures of the smart contract. In this way, their functionality can be tailored according to specific business logic, which ensures that all the business requirements are fulfilled whenever a user interacts with a smart contract. The fact that Blockchain Technology is inherently decentralized, the addition of the smart contract layer on top of it made it possible to harness the benefits of decentralized computational logic.

Solidity, introduced in 2015 [7] opened the doors for diversity in the area of Dapps (decentralized applications). However, without a consistent way to allow different contracts to communicate and exchange data, the interoperability between these applications is hindered considerably, potentially ruining useful use cases. The idea of interface standardization is a very welcomed addition to the Ethereum ecosystem. ERC721 [5] defines a minimum interface a smart contract must implement to allow NFTs to be managed, owned and traded. NFTs are distinguishable by their ID and can represent ownership over digital or physical assets, such as land, unique artwork, virtual collectibles, negative value assets (e.g. loans), among others. Nevertheless, the core interaction with ERC721 compliant smart contracts is

always occurring through nine functions defining the base API, and six additional ones for extensions to the standard.

We identified four reference implementations that tackle a similar issue in the application area of NFTs, namely providing the means for an easier creation of ERC721 compliant smart contracts together with extending these with various practical features and configuring them accordingly. The identified platforms that try to solve this problem are: OpenZeppelin¹, Oxcert [8], Mintable² and Smartz [9].

OpenZeppelin is a library for Solidity smart contract development. It provides tested and community-audited code. It offers implementations of standards (e.g. ERC20, ERC721) which can be extended and deployed. These are in the form of generic contracts that need to be manually integrated and configured as well as mock implementations showing how certain features should be coupled together.

Oxcert [8] is an open-source TypeScript framework to create, manage and swap digital assets (ERC721 tokens) and value tokens (ERC20 tokens) on the Ethereum and Wanchain blockchain. It aims to provide a standard way to build and deploy decentralized applications through the Oxcert protocol. Additionally, it helps simplify the development process and allow programmers to avoid low-level details of smart contract programming and only interact through the high level API that the Oxcert framework provides.

Smarktz [9] has the goal to bring decentralized applications to a new level, by providing developers, end-users and business entities with a convenient, simple yet powerful and cost-effective way to materialize their ideas through apps. It does this by offering a variety of types of smart contracts, which can be easily configured and deployed. An ERC721 contract is one of these options. The way Smartz operates is by allowing to pick the type of smart contract to deploy and then the users are presented with an interface where input data fields and check-boxes are used in order to fill in the necessary details to configure the contracts. This is a straight forward approach that does not require knowledge of programming as it is handled in an abstract fashion.

Mintable is an in-development platform that provides the means to tokenize and manage digital assets (ERC721 tokens) on the Ethereum blockchain. It aims to simplify and streamline the interaction with ERC721 tokens. It supports a contract generator through which users create their own ERC721 compliant contracts, deploy them and mint tokens. Mintable is in its beta phase and the features it supports are limited. It takes a similar approach to contract customization that the Smartz platform has, but with fewer customization options. Unlike Smartz, it does not allow to see the code of the final version of the configured contract.

In all of the reference implementations there is a trade-off between depth of configurability and ease of customizability. While OpenZeppelin offers the basic needs and provides full flexibility and control, it requires complete knowledge of Solidity. In a similar category is Oxcert as it gives lots of options for customization but that comes with the

1. <https://openzeppelin.com/contracts/>

2. <https://www.mintable.app/>

prerequisite of being able to program in TypeScript. On the opposite spectrum we can place both Smartz and Mintable. They have the lowest skill entry level, but also offer the least amount of options for customization. Our framework, NFTY, aims to strike a reasonable balance between the two aspects of concern: ease and depth of configurability.

3. Proposed Approach

We present an analysis of several popular and currently operating decentralized applications which are based on the ERC721 standard. We then determine a list of frequently encountered features and extensions in this application domain. These serve as the basis for the smart contract suite that we build. We then define a SL that uses the suite to obtain different deployable configurations of the contracts.

In order to identify popular and currently operating ERC721 smart contracts, we have selected the top most active smart contracts based on the recent number of transactions. Etherscan allows us to explore Ethereum blockchain based on transaction activity and filter on ERC721 compliant contracts. We selected the top ten smart contracts when sorting by the number of transactions performed in the last seven days (as of 01/08/2019) and also ensuring that there is some level of activity on the smart contract in the last twenty four hours. We collected the following list (in decreasing order of activity): *CryptoKitties*, *DozerDoll*, *GodsUnchained*, *LucidSight-MLB-NFT*, *MARBLE-NFT*, *MyCryptoHeroes:Extension*, *CryptoFlowers*, *Ethereum Name Service*, *MyCryptoHeroes:Hero*, *Spheroid SPACE*.

Eight of the ten applications are blockchain-based games. The remaining two are augmented reality entertainment and advertising, and a decentralized name service. Gaming applications are the most prominent and popular application domain for ERC721 smart contracts while the advent of tokenization of real life goods shall create new opportunities for non-fungible token application domains. A shift in ideology that will help us start to value digital goods will allow for more services such as Ethereum Name Service to prosper on the blockchain.

3.1. ERC721 Features and Extensions

Besides MARBLE-NFT and Spheroid SPACE, the source code of the smart contracts is published and available. After studying the code, the set of frequently used features and extensions was constructed, which helps us answer the second research question. The mapping in Table 1 presents the support for the identified features in all of the analyzed contracts, and we describe them below.

1. *Contract ownership and transfer* refers to the ability of assigning an address as an owner to a contract, usually upon its creation and the capacity to transfer the ownership to another address. 2. *Metadata* standard extension defines an optional extension for ERC721 smart contracts. It allows a smart contract to be queried for its name, symbol and details (referenced by a Uniform Resource Identifier) about the assets the NFTs represent. 3. *Enumeration* standard

	CryptoKitties	DozerDoll	Gods Unchained	LucidSight	My Crypto Heroes	CryptoFlowers	Ethereum Name Service	My Crypto Heroes:Hero
Contract ownership/transfer								
Metadata Extension								
Enumeration Extension								
Custom Token Data								
Limited Token Supply								
Token Minting								
Token Burning								
Contract Pausability								
Contract Access Control								
Auctioning								

TABLE 1. MAPPING OF ERC721 EXTENSIONS AND FEATURES TO SUPPORTING SMART CONTRACTS (SUPPORT IS MARKED IN GRAY).

extension allows a contract to be queried for its full list of NFTs and thus making them discoverable. Additionally, it allows to browse NFTs owned by a specific address. 4. *Custom token data* refers to additional token data that is stored on the blockchain and it is referenced by the token ID that corresponds with this data). 5. *Limited token supply* allows to set a limit that enforces how many tokens a smart contract can release (create). This acts as a maximum supply of tokens in the contract, meaning that the contract does not track ownership of more tokens than this limit specifies. 6. *Token minting* reflects the ability of a smart contract to create new tokens. 7. *Token burning* reflects the ability of a smart contract to delete existing tokens. 8. *Contract pausability* determines the ability of a contract to have a fail safe mechanism to stop any state modifying functions from being accessible. This can happen for reasons such as security risks being identified within a smart contract and thus it allows to reduce the amount of damage that can be caused by such events. 9. *Contract access control* introduces the idea of separate layers of contract accessibility such that it can be controlled which functions are accessible by which addresses. 10. *Auctioning* attributes the extension which allows to trade NFTs directly on the blockchain.

From this analysis we observe that all of the identified extensions and features are used in most of the considered applications. This is backed by the fact that any of the extensions is supported in at least 50% of the analyzed contracts (4/8 contracts support any given feature). This goes to show that the identified extensions and features are likely to be useful for ERC721 compliant smart contracts as they are generally needed in this context. The second thing to notice is that most of the presented decentralized applications make use on average of more than 7/10 features from this list. This shows that most of the features and extensions that we have identified are used in combination.

3.2. Framework

Our framework is structured into three distinct layers that depend on each other, namely the *Smart Contract Suite*, *Smart Contract Templates* and *Specification Language*.

The *smart contract suite* layer is represented by the complete smart contract suite that attributes all of the identified

extensions, specified in Section 3.1.

The intermediary layer consists of the *smart contract templates*. Based on the complete smart contract suite, the templates enable us to manipulate the contracts in an accessible fashion without the need for Solidity. These template contracts allow us to inject code in order to configure the smart contract suite to a specific set of required features.

Solidity is specifically tailored to accommodate the needs of smart contract development. We decided that a *specification language* that allows to customize the underlying smart contracts is a suitable approach to achieve a low knowledge entry level to the ERC721 based smart contracts as well as an efficient customization process. The SL enables a developer to focus only on the final set of contracts obtained as a result of the configuration process and not be concerned with understanding the entire smart contract suite. This SL has two components. The first is the grammar which is defined to allow to easily include or exclude features and/or extensions and be sufficiently verbose. The second is the compiler which parses a specification file and extracts the necessary data and supplies it to the appropriate templates in generated Solidity code format. The end result is a deployable smart contract that follows the configuration elements defined in the input specification file.

We elaborate on the SL grammar through an use-case example. Below, we present a specification file which includes several of the available extensions, adds a few token data fields and modifies two functions:

```
contract
derive owner, admin, minter, pause
include meta('Tokies', 'TK'), mint(20), enum, auction
extend with address neighbour, bool active
modify ownerOf with onlyContractOwner
modify balanceOf with onlyTokenOwner
```

First, the keyword `contract` is used in order to allow for the compiler to receive a zero-configuration file, where it only extracts the base ERC721 contract together with its default dependencies. Next, the `derive` keyword is used for the access control features (5 in total). These features are attached to the base ERC721 contract template since they define function modifiers which can be used across all the other contracts present in the configuration. The `include` keyword is used for adding any of the remaining features and extensions. Some of these features ('meta' and 'mint') can also take parameters. The `extend with` keywords are intended for additional/custom token data. We can define comma-separated fields that contain a Solidity type and a field name. The `modify` keyword enables to attach modifiers to different functions in the contracts.

4. Evaluation

We define an evaluation scheme which is created based on relevant measures of interest. We focus on extensibility and reusability, as these are two of the main quality attributes that can attest the usefulness of a framework as they assess the structural and non-structural aspects of a code fragment. Additionally, Solidity has no organized and maintained repository for smart contract development.

Developers have to rely on third party sources to find useful pieces of code that can be reused and extended.

To the best of our knowledge there are no studies on the topic of evaluating Solidity smart contracts from the perspective of high level quality attributes (besides those focused on security [10], [11]). The studies published so far [12] only look at general metrics and do not group these in order to represent more abstract quality attributes. Nevertheless, Solidity provides programming structures that follow the object-oriented programming paradigm. We thus aim to translate the knowledge from evaluating object-oriented components to the smart contract field.

The extensibility quality attribute refers to the presence and usage of properties in an existing design that allow for the incorporation of new requirements. It is measured in terms of four metrics, namely Measure of Functional Abstraction (MFA), Direct Class Coupling (DCC), Average Number of Ancestors (Avg. NOA) and Number of Polymorphic Methods (NOP) [13]. The reusability quality attribute is defined by five reuse factors: incurred reuse, external quality, documentation, availability and complexity. Consecutively, there are 7 metrics that influence reusability, namely Number of Dependencies (NDEP), Number of Open Bugs (OP_BUGS), Documentation (DOC), Number of Components (NCOMP), Lack of Cohesion of Methods (LCOM), Number of Classes (NOC) and Weighted Method per Class (WMC) [14]. We consider only the latter five due to unavailable tracking information for the first two.

For evaluating the SL we take a use-case approach where we analyze the difference in efficiency and effort required to create a configuration of the smart contract suite. The comparison is performed between the automated approach achieved through the SL and the manual approach by picking and tweaking the required smart contracts. We take into consideration three use cases in an increased order of difficulty and use the Lines of Code (LOC) metric as an approximate assessment for the effort and efficiency.

4.1. Smart Contract Suite Evaluation

Coupling gives a good indication of the design complexity of a piece of software and lower DCC values correlate with code that is easier to understand and manipulate. In the smart contract context, DCC is considered as relations that appear between independent smart contracts (besides the ones in the inheritance list as this is treated separately). In Table 2 the average DCC values for the five smart contract suites are presented. We easily notice that these are generally 0 or very close to 0. Coupling is very limited among smart contracts [12] and we conclude that the contracts define a self-contained programming unit without any connection to other contracts. These results suggest that coupling is not meaningful enough when evaluating the extensibility of a smart contract suite, mainly because it generally has low values and thus it has an artificially positive impact on whether a set of contracts are easily extensible or not.

Following the idea that smart contracts act like a self-contained programming units, inheritance is the mechanism

that allows for abstracting functionality into different layers. We observe that inheritance is used regularly and generally half or more of the contracts from a smart contract suite are a descendant of at least another contract. Inheritance levels can reach high values (up to 10 in our analysis) whenever a contract extends upon several different features. On average, each contract has around 2 ancestors (Table 2). We consider the use of inheritance to be important when developing smart contracts that rely on several different levels of logic.

We have specified that inheritance enables to design smart contracts by separating them into several layers of functionality and logic. In order to assess how well the layer abstraction is performed, the MFA metric comes into play. We noticed that MFA is consistently high (good values) for all five smart contract suites we analyze and half or more of the contracts in a suite have an MFA of at least 0.5. We do however observe that the MFA metric is rather sensitive to classes (contracts) that are high in the hierarchy since at these levels the number of inherited functions is low (0 for root ancestors) and each defined function has a big negative impact on the value of MFA. On average (Table 2), MFA values are around 0.5 which show that a good level of functional abstraction is present in the majority of contracts. Therefore, MFA is a relevant metric to assess the abstraction levels but since it is rather sensitive to certain smart contracts, it can be useful to reconsider the definition and restrict it to a subset of the original set of contracts (e.g. by not considering libraries or root contracts).

Polymorphism, in simple terms, refers to the ability to redefine methods for derived classes. It is thus highly relevant when it comes to extending a smart contract as it enables an easy redefinition of ancestor functionality. Given that we consider five platforms that are focused on a similar application domain, assessing extensibility through the NOP metric is sufficiently insightful. However, the metric itself, in a generic scenario does not provide sufficient information on whether it has a positive or negligible impact on extensibility. It would thus require to be calibrated to take other aspects of the smart contracts into consideration (e.g. the size of the smart contract suite).

The number of classes (contracts) is assumed to represent the amount of functionality available within a software asset [14]. In our case, four of the five implementations have about 20 classes (Table 2). Following this reasoning, it goes to show that these implementations bear a similar level of functionality. However, our smart contract suite provides support for all ten identified frequent features and extensions, while the others only for a subset of it and still manage to score higher. It is a matter of design and implementation but ultimately, directly assessing reusability based on this metric can lead to misjudgment. It does not look at raw functionality and the NOC is often misleading in terms of the contents of those classes.

One of the aspects that facilitates usability is code that is of low complexity. WMC assesses the complexity of a contract based on its control flow graph. Thus, the more elaborate the control flow graph is, the more instructions are performed within that class and therefore it bears more

	Oxcert	Smartz	Zeppelin	Mintable	NFTY
Avg. DCC	0.00	0.00	0.13	0.17	0.00
Avg. NOA	1.17	2.65	2.22	2.04	2.40
Avg. MFA	0.30	0.60	0.56	0.45	0.54
NOP	6	13	7	6	6
NOC	12	17	23	23	20
Avg. WMC	5.00	4.47	4.12	5.22	4.75
Avg. LCOM	0.28	0.17	0.09	0.27	0.15
Avg. DOC	0.83	0.5	0.78	0.76	0.9

TABLE 2. THE CALCULATED METRICS FOR THE 5 ANALYZED SMART CONTRACT SUITES (DARK GRAY - BEST VALUE; GRAY - SECOND BEST; LIGHT GRAY - THIRD BEST; WHITE - FOURTH OR WORSE).

complexity. In Table 2, we notice that the average WMC metric is under 5 units for all of the analyzed implementations. On average, the complexity of the contracts is low. The extensions to the ERC721 standard are also of low complexity which denotes that reusing them is straightforward.

Lack of Cohesion indicates that a class is violating the Single Responsibility Principle [15]. Following this principle, each class should provide the system with only one functionality. We observed that more than half of the Solidity contracts appear to be highly cohesive, denoted by values close to 0 for the LCOM metric. This occurs for the extensions that only introduce new logic into the contracts and no new data structures. The contracts with higher lack of cohesion correspond to the base parts of the ERC721 standard. From Table 2 we understand that on average the smart contract suites are highly cohesive. The LCOM metric is useful to be analyzed in the context of a contract overview such as to identify the classes that are insufficiently cohesive and determine whether that can affect the reusability of the software asset. It does seem that LCOM is sensitive to smaller classes that provide an extended functionality to an already present feature. LCOM values are then low and the average statistics can be skewed due to the bias from these multiple extensions that achieve high cohesion.

The NCOMP metric follows the idea of counting candidate components which represent well-formed subsets of classes. The identified sets of classes are not meant to be used as black box components in the reuse process, since they are not examined from a functionality standpoint and thus might be uncompileable. However, these components are intended for white-box reuse since they are meant to exhibit minimum external dependencies with good functionality and reusability [16]. We notice that this metric suffers a considerable increase in values whenever it is applied to hierarchical structures, especially with multiple inheritance and for the classes with a large amount of ancestors. This sensitivity in the metric enables contracts that only combine functionality from their ancestors to achieve extremely high values in the number of components. In turn, it means that looking at total values for the NCOMP metric for a smart contract suite will not be sufficiently insightful. Having a more balanced distribution of candidate components over the contracts in a suite is more relevant.

Documentation is highly valuable when an asset is to be reused. However, it currently has to be performed manually and can be considered more subjective than objective. We used a three-scale rating for the contracts based on the level of documentation: 1 - fully documented; 0.5 - moderate/-

Complexity	Low	Medium	High
LOC - manual	6	14	38
LOC - automated	3	6	8

TABLE 3. THE LOC METRIC FOR 3 LEVELS OF USE CASE COMPLEXITY.

good documentation; 0 - poor/absent documentation. We observe that most of the analyzed smart contract suites have a good level of documentation (Table 2).

4.2. Specification Language Evaluation

We define three use cases of increasing complexity in order to assess the difference in efficiency and effort required for configuring the smart contracts. We compare the LOC metric for the automated (i.e. SL) approach and the manual approach of directly editing the smart contracts. In Table 3 we observe the evolution in terms of the LOC metric for the three use cases. We notice that an increase in complexity affects the manual configuration considerably more than the SL, where for the most complex considered use case, the difference in the LOC values is in the range of a factor of 5, meaning that five times as many lines of code are needed in the manual approach to achieve the same configuration.

The difference of a factor of 5 is where the LOC metrics stabilize (when setting the SL as a boundary for configuration ability) and thus anything more complex will be in the same range. Nevertheless, this is a substantial difference and we can confirm that the SL does indeed provide a meaningful increase in efficiency and effort decrease for customizing and configuring the smart contracts.

5. Discussion and Conclusion

There are several discussion points with respect to evaluating the extensibility and reusability of Solidity smart contracts. First is that we can argue in favor of considering both these quality attributes together since the metrics can generally be discussed upon from the perspective of both extensibility and reusability. Additionally, this is supported by the fact that black box reusing is not yet a property of Solidity smart contract development, hence reusability is typically followed by a level of extensibility. Second is that some of the considered metrics are not directly appropriate for smart contracts in the frame of the definitions we have used. Thus, it is important to tweak those metrics to make them more applicable since the smart contract domain is more specific than general OO programming. Third is that documentation, which is dependent on a subjective opinion of the evaluator, has to be derived from a set of domain relevant requirements. Finally, the average or total statistics on the analyzed metrics are not always sufficient during an analysis as interesting patterns and valuable insights can be observed by looking at individual classes (contracts), while an aggregated index requires more research on the matter together with building and validating a model.

We have identified the currently most popular application domains for NFT smart contracts and have constructed a list of frequently encountered features and extensions used

in these smart contracts. These features and extensions have been compiled in a comprehensive smart contract suite that supports the ERC721 standard. Our analysis helped us to identify that for extensibility, the most insightful metrics are NOA, MFA and NOP, where the latter two could be tweaked to be less sensitive to smart contract specific aspects. For reusability, the most applicable metrics are WMC, DOC, LCOM and NCOMP, where the latter two are more valuable when analyzed in a per class/contract fashion. The framework we have built is focused on the SL that uses the smart contract suite in template form and offers a simple and more efficient alternative to customize and configure the smart contracts, which is underlined by the LOC metric, and couple them into a readily deployable contract.

References

- [1] N. Szabo, "Smart contracts: building blocks for digital markets," *EXTROPY: The Journal of Transhumanist Thought*, (16), vol. 18, 1996.
- [2] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press, 2016.
- [3] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," *URL: https://github.com/ethereum/wiki/wiki/White-Paper*, vol. 7, 2014.
- [4] V. Buterin and F. Vogelsteller, "Erc20 token standard," *URL: https://theethereum.wiki/w/index.php/ERC20_Token_Standard*, 2015.
- [5] W. Entriken, D. Shirley, J. Evans, and N. Sachs, "Erc-721 non-fungible token standard," *Ethereum Foundation*, 2018.
- [6] I. Bashir, *Mastering blockchain*. Packt Publishing Ltd, 2017.
- [7] C. Reitwiessner and G. Wood, "Solidity," *URL: http://solidity.readthedocs.org*, 2015.
- [8] K. Sedlak, J. Zih, M. Pirc, and U. Osvald, "0xcert protocol whitepaper," *https://0xcert.org/whitepaper.pdf*, 2018.
- [9] S. Prilutskiy, Y. Vasilchikov, A. Makeev, V. Khramov, and V. Buyanov, "Smartz: Multi-chain smart contracts management system," 2018.
- [10] R. M. Parizi, Amritraj, and A. Dehghantanha, "Smart contract programming languages on blockchains: An empirical evaluation of usability and security," in *Blockchain – ICBC 2018*, S. Chen, H. Wang, and L.-J. Zhang, Eds. Cham: Springer International Publishing, 2018, pp. 75–91.
- [11] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.
- [12] P. Hegedus, "Towards analyzing the complexity landscape of solidity based ethereum smart contracts," *Technologies*, vol. 7, no. 1, 2019. [Online]. Available: <https://www.mdpi.com/2227-7080/7/1/6>
- [13] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.
- [14] A. Ampatzoglou, S. Bibi, A. Chatzigeorgiou, P. Avgeriou, and I. Stamelos, "Reusability index: A measure for assessing software assets reusability," 05 2018.
- [15] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.
- [16] A. Ampatzoglou, I. Stamelos, A. Gkortzis, and I. Deligiannis, "A methodology on extracting reusable software candidate components from open source games," in *Proceeding of the 16th International Academic MindTrek Conference*. ACM, 2012, pp. 93–100.