Smart-Graph: Graphical Representations for Smart Contract on the Ethereum Blockchain

Giuseppe Antonio Pierro

Smart-Graph: Graphical Representations for Smart Contract on the Ethereum Blockchain

Giuseppe Antonio Pierro
Centre Inria
Lille Nord Europe
Lille, France
giuseppe.pierro@inria.fr

***Abstract*—The Ethereum blockchain enables executing and recording smart contracts. The smart contracts can facilitate, verify, and implement the negotiation between multiple parties, also guaranteeing transactions without a traditional legal entity. Many tools supporting the smart contracts development in different areas are flourishing because in Ethereum blockchain valuable assets are often involved. Some of the tools help the developer to find security vulnerabilities via static and/or dynamic analysis or to reduce the Gas fees consumption. Despite the plethora of such tools, there is no tool supporting smart contracts evaluation and analysis via a graphical representation for expert developers.**

The paper embraces this way to facilitate the developers' analysis activity, by proposing a graphical representation model to visualize smart contract source code. The paper makes available a tool via a web interface, which accepts the smart contract address as an input and produces a graphical representation of the smart contract as an output. The graphical representation can help developers to better understand the structure of smart contracts and share it with other developers. Moreover, some metrics, such as the relations among smart contracts, are easier to be understood via "spatial" than "tabular" representation. Indeed, representing smart contracts' metrics via visual representation facilitates the developers, who are used to analyze the source code by directly inspecting it or using other tools that provide the metrics in a table format. Finally, the paper provides detailed data regarding a smart contract to the developers and proposes a graphical representation of the smart contracts without obscuration of details, also highlighting areas of the code that are possibly too big in size and/or too complex via a diagram displaying their connections.

***Keywords*—Smart contract, Ethereum blockchain, visual analysis, class diagram.**

I. Introduction

The use of blockchains is currently explored in a number of scientific fields due to their potential to radically change the ways of economical exchange and the hows of traditional legal entities [1]. For these reasons, in the Ethereum blockchain, where specific expertise in smart contracts' development is required, also other experts are required, i.e. experts in statistics, machine learning, economics, marketing and finance [2], [3]. In addition, the complexity of blockchain engineering hampers the understanding of the users, be them laypeople, experts in other fields and sometimes even software developers [4]. Previous studies showed that graphical representations of data can help the understanding of a subject, making it a faster and easier task [5].

In the paper, we review some visualization approaches and practical tools previously used to provide technical insights for experts and not-experts in the field of graphical representation of the static aspects of the software. After reviewing and discussing the main approaches, the paper proposes a web-based tool to graphically represent smart contracts. Agile methods promote "working software over comprehensive documentation" [6], [7]. Recent research has shown that agile teams use quite a number of artefacts [8]. One of these artefacts is Unified Modeling Language (UML) [9]. UML is a modelling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system [10]. UML offers a way to visualize a system's architectural blueprints in a diagram, including elements such as the individual components of the system; how the system runs; how entities interact with each other (components and interfaces) [11]. The paper aims to design and build a tool, "Smart Graph", to visualize the smart contracts entities and how they interact. Figure 1 shows an example of the graphical representation provided by the tool, "Smart Graph" [12].

In this paper, we survey both two-dimensional space (2D) and three-dimensional space (3D) visualization techniques, representing the static aspects of the software. Finally, the paper presents a web-based tool, "Smart Graph", to generate a 2D graphical representation of the smart contract's source code via a UML class diagram. The graphical representation provided by "Smart Graph" is an augmented version when compared to the previous traditional UML class diagram. Indeed, it allows giving information that is specific to Solidity programming language, such as "Function Modifer" and "Function Fallback" [13], [14].

Furthermore, the paper presents and discusses the general components of the web-based tool "Smart Graph". The advantage of web-based tools is that the Solidity developers do not need to install any software in their operating system. Solidity is a contract-oriented, high-level language for implementing smart contracts and is designed to target the Ethereum Virtual Machine (EVM) [15]. The tool presented in this paper aims to provide smart contracts programmers with a fast overall view of the smart contract structure and a useful insight into the source code they are developing.

The rest of the paper is structured as follows: Section II presents some work on the graphical representation of data

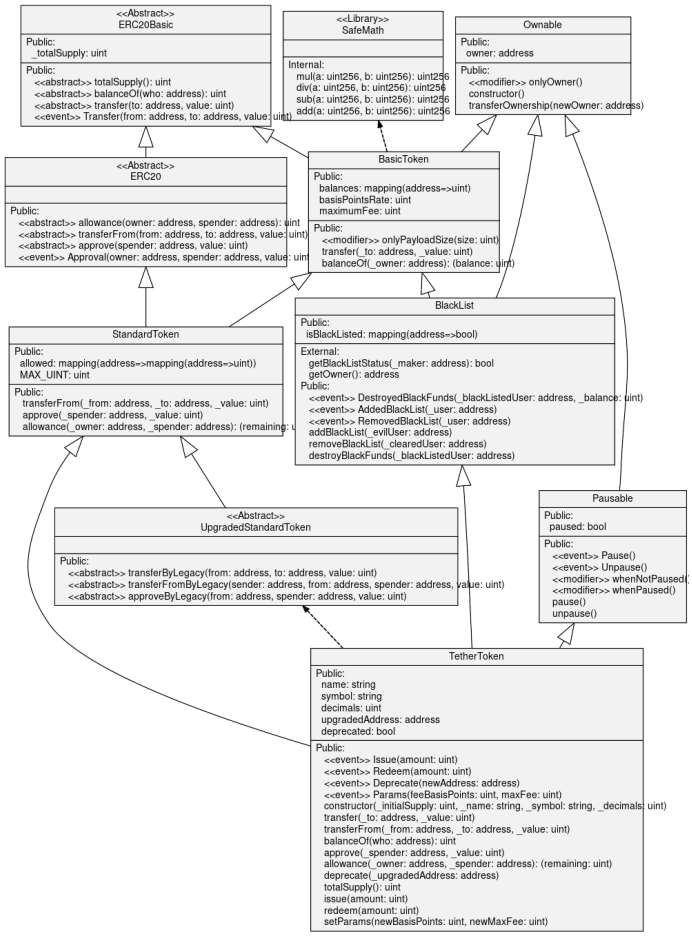

Fig. 1. UML class diagrams of Tether smart contract.

software. Section III describes the methodology that was used to design and build the tool. Section V presents the conclusion of our research.

II. Background and related work

The design, development, implementation and review of a program might be a challenging task even for expert users [16]. Often it is also a time-consuming activity mainly because of the software complexity to manage [17]. Graphical representations of programs have the potential to provide a better and faster understanding of its design and functionality, saving time and giving valuable information to improve its quality [18]. Therefore, the use of visualization techniques might speed the understanding of source code programs [19]. It can also help in eliminating bugs due to a faulty understanding and in spending less time in reading the source code because the software developer does not need to go through each line of the source code [20]. Indeed, by using a graphical representation, the software developer can have a quick glance on the program implementation [21]. Over the past few years, researchers have proposed many software visualization techniques for static program analysis and various taxonomies have been provided [22], [23]. The techniques can be divided into techniques for 2D and 3D representations [23].

Among the 2D software visualization techniques, UML class diagrams are probably the most popular graph-based software visualization techniques [24]. Their purpose is to display inter-class relations, such as inheritance, associations, aggregations and composition [25]. Like other graphical representations, when UML class diagrams grow, they gradually become visually complex and prone to information overload. Other software visualization techniques for static program analysis employ graphs or diagrams such as tree-map, matrix, Kiviat/radar chart. 3D representations can represent real-world metaphors [26].

Other visual representations make no use of graphs and diagrams, which are more suited for expert users. In the paper, we will focus on graphical representations that can provide useful insights for experts, i.e. smart contracts developers. However, other work focused on visual representations that could be more useful for non-experts in software development, i.e. experts in other domains or stakeholder. For instance, visualizing source code programs using a real-world metaphor consists in representing the programs within a familiar context, by using graphic conventions the user immediately understands. Some visual representations are based on metaphors, which is a useful device to let people understand complex and abstract conceptual domains in terms of more concrete conceptual domains [27]. To make software structure more intelligible, some concrete metaphors, such as the city metaphor, have been selected, as they rely on the human natural understanding of the physical world, including spatial factors and relations in perception and navigation [28]. Such metaphors can facilitate the understanding of software structures and relations among its components [29].

For instance, Wettel and Lanza [30] developed the "CodeCity" visualization tool that is based on the city metaphor. CodeCity displays classes as buildings and packages as city districts. Classes within the same package are grouped using a tree-map layout to slice the city into districts. The visual representation based on the city metaphor is as close as possible to real cities. By doing this, the authors intend to enable a very intuitive understanding of a software system. The user is free to zoom and navigate through the city representing the software.

Graham et al. proposed another metaphor to represent source code software: the Solar systems [31]. Their visual representation presents the source code software as a virtual galaxy made of many solar systems. Each solar system represents an entire package. Its central star is an icon that represents the package itself, while planets, in orbit around it, represent classes within the package. The orbit level indicates the depth of a class in the inheritance tree: the farther from the star, the deeper in the inheritance tree. Blue planets represent classes and light blue planets interfaces. Solar systems are shown as a circular formation to improve readability, but the position of planets and solar system can be moved by the user, which is a very interesting feature that few visualizations offer.

The visualization techniques listed above have been applied to several projects written with different programming languages, especially programming languages that support object-oriented programming paradigms (OOP), such as C++, C# and Java. Unfortunately, there is no example for projects such as the decentralized application (DApp) written with the Solidity programming language, because Solidity is a fairly new language compared to the others. Anyway, if we limit the study to static program analysis, the Solidity programming language can be compared with the most popular OOP languages. Indeed, smart contract acts as a class. This means that smart contracts, like the classes of other OOP programming languages, can inherit from other smart contracts a common interface for implementing a specific set of functions, for instance, ERC20 tokens or NFTs.

However, Solidity presents specific features and constructs, such as "Receive Ether Function", "Function Modifer" and "Function Fallback", which are not present in other OOP programming languages [32]. A "Receive Ether Function" is a function that can receive Ether, the cryptocurrency of the Ethereum Network. A "Function Modifer" is a Solidity construct which is used as a pattern to change the behavior of some functions, and in many cases, to restrict them. A "Function Fallback" is a Solidity construct which is executed when a "Function Identifier" does not match any of the available functions in a smart contract. The current UML definition does not provide a way to display this kind of information.

At the date hereof, a web-based tool to produce UML diagrams from smart contract source code does not yet exist. For other object-oriented programming languages, such as C++, C# and Java, there are instead a lot of tools to produce UML diagrams [33].

III. Research Methodology

The tool presented in the paper has been designed to make the access easy for the users as it does not require any configuration and installation in the users' computers. The users can indeed access the tool from any web browser. Through the tool's web interface, the user can give a smart contract address as an input, and get a augmented UML class diagram of the smart contract address as an output. The tool is available via a web browser and can be tested at the following link: https://aphd.github.io/smart-graph/. The backend produces the graphical representation of the smart contract, i.e. the augmented UML class diagrams.

A. Research Questions

The paper aims to discuss the strengths and limitations of the UML class diagrams to visually represent the smart contracts' source code. The paper considers and analyzes a source code coming from a DApp composed of many smart contracts, and it proposes a web-based tool to overcome the limitations of the UML class diagrams. The paper aims to answer the following research questions:

- Q1 Are the UML Class diagrams used by the most popular OOP languages, such as C++ and Java, sufficient to visually represent the smart contracts' source code?
- Q2 Is there a better way to visually represent the smart contracts' source code?

To answer the research questions, the following hypotheses are proposed:

- H1 The current UML class diagrams used by some OOP languages, such as C++ and Java, are not sufficient to visually represent the information of a smart contract. For instance, in the current UML class diagrams' specification, there is no way to visually represent features of the source code that are unique in Solidity, and that could be very important for a smart contract developer, such as the "Receive Ether Function" and the "Fallback Function".
- H2 Given the complexity of some DApps, which are often composed of many smart contracts, a static representation of the source code is not apt to represent the most important information of the source code in an accessible way. An interactive environment for the smart contracts' developers can be useful to better represent the smart contracts' source code, for instance by allowing to visualize the smart contracts' source code through actions, such as drill down or zoom in/out in the diagram.

B. System Architecture

The system is composed of two entities: a back-end and a front-end, which are loosely coupled by an API. Both back-end and front-end use open source technologies. The main components of "Smart Graph" are listed below:

- Express is a "Node.js" web application framework that provides network routing.
- Webpack is an open-source JavaScript module bundler. A bundler takes the dependencies of the software source code and generates a dependency graph.

1) Frontend: The front-end is the platform component visible and accessible to the users, stakeholders included.

The front-end was implemented in a web-based environment. This is an advantage, as the front-end component is platform-independent. This means that the users will only need a web browser to access the platform. Moreover, it will be possible to benefit from cutting-edge web technologies to produce graphical representations and visualize them.

Figure 2 shows the "Smart Graph" GUI accessible through a web browser. The GUI is divided into two sections: the "Smart Graph Form", on the top of the page, and the "Smart Graph Diagram Container", in the middle of the page. The "Smart Graph Form" has a text field as an input and a button named "Generate the Diagram" to get the output. The input text field allows the user to write a smart contract's address. When the user clicks on the button, the backend generates the corresponding UML class diagram, which is shown below the form. The "Smart Graph Diagram Container" is the area where both the classical UML class diagrams and the augmented UML class diagrams are displayed.

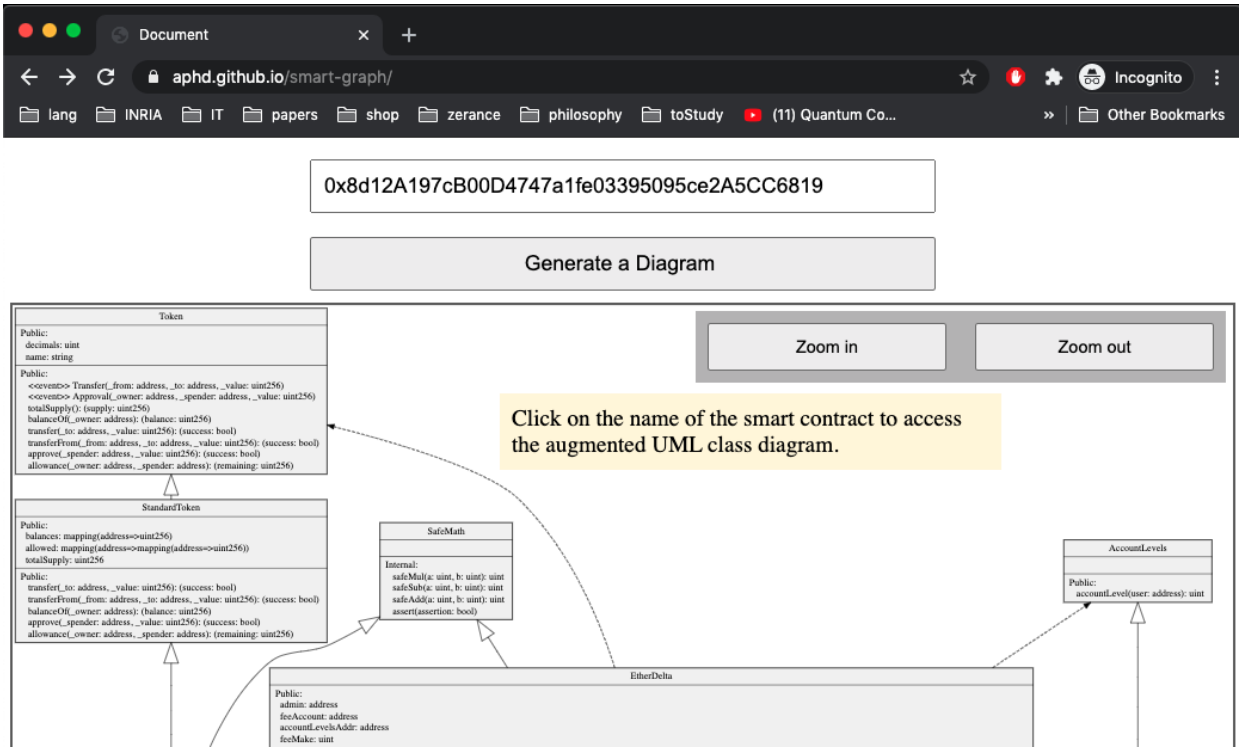

Fig. 2. The "Smart Graph" GUI accessible through a web browser over the internet.

In designing the front-end part, a challenge is representing DApps that are composed of many smart contracts. In these cases, the UML class diagrams are visually complex and prone to be information overloaded. To visualize many UML class diagrams, the "Smart Graph Diagram Container" is interactive, as the user can perform different actions within the class diagram. For instance, the GUI presents two buttons, which are accessible in the upper right part of the interface 2 and make it possible to zoom in or zoom out a certain part of the UML diagrams. The zoom-in action allows getting details about specific features of the smart contract, by selecting the most important parts of the visual representation and hiding the less relevant parts. Moreover, the user can see the augmented version of the UML diagram, by clicking on the smart contract's name displayed on the top of the UML class diagram, as shown in Figure 3.

Figure 3 shows the details about the method named "transferOwnership", which belongs to the smart contract "Ownable" displayed in Figure 1. The implementation of the "Ownable" smart contract made by the OpenZeppelin developers is available at the following link: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol. The function "transferOwnership" allows transferring the owner's account, i.e. the account that deployed the smart contract, to a new owner.

Figure 4 shows the pipeline of the operations executed by the backend when a user requests the UML class diagrams by providing a specific smart contract address.

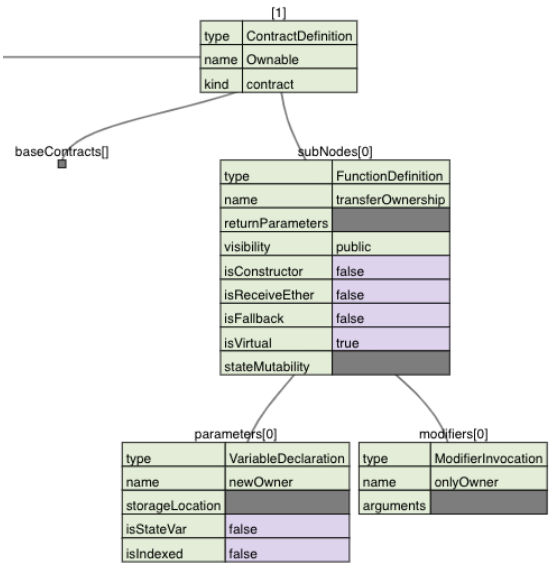

Fig. 3. trasnferOwnership method visualized as a tree diagram.

2) Backend: The backend code is written in javascript because it supports synchronous operations like Promise. The Promise's feature is very important in the backend-part because it allows handling potential blocking operations, such as fetching resources from a server. An instance of potential blocking operation occurs when the user submits the request to the backend of the tool. When this happens, the backend makes

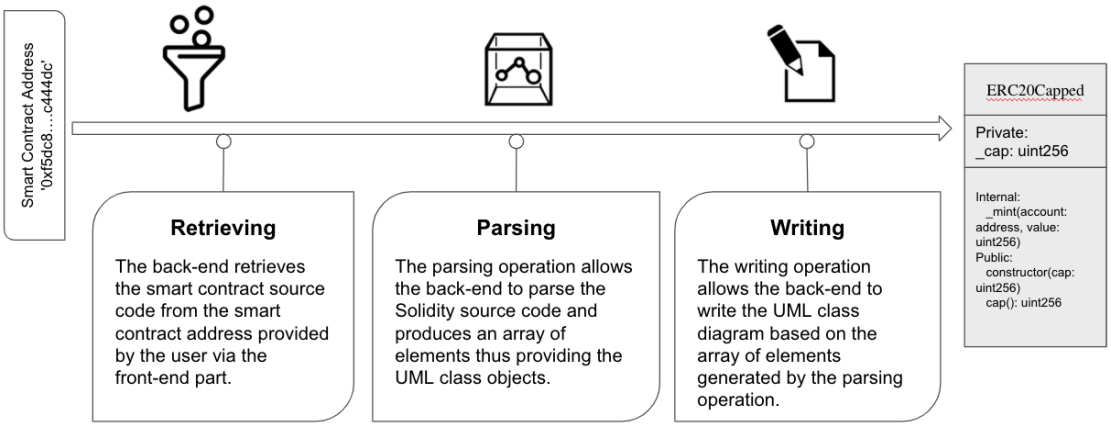

Fig. 4. Pipeline of the operations executed by the backend when a user requests the UML class diagrams of a specified smart contract's address.

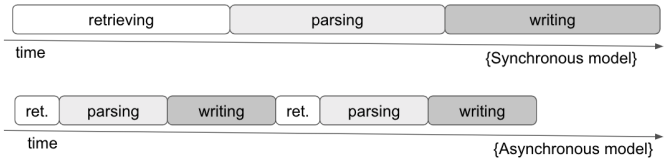

Fig. 5. Synchronous Model vs Asynchronous Model.

a request to Etherscan to get the source code corresponding to the smart contract address specified by the user. Etherscan is an Ethereum block explorer which allows to explore and search the Ethereum blockchain for smart contracts source code [34].

The operation of fetching a smart contract source code from the network could be time-consuming, as it can take around 5 seconds. Therefore, we use a feature of JavaScript language named "Promise", in order to timely make the system responsive to other requests. Essentially, a Promise is an object that represents an intermediate state of an operation. There is no guarantee of the precise moment when the operation will be completed and the result will be returned, but there is the guarantee that, when the result is available or when the promise fails, the code provided to the Promise will be executed in order to do something else with a successful result or to gracefully handle a failure case. This is useful to set up a sequence of async operations to correctly work. Figure 5 shows the asynchronous operation that can occur during the generation of the augmented UML class diagrams.

The backend supports the following main operations: data retrieving III-C, data parsing III-D and data writing III-E.

C. Data Retrieving

The retrieving operation allows the backend part to retrieve the smart contract source code. The retrieve method takes one argument, the path to the resource to fetch which is made from the following two parts: the Etherscan address and the smart contract address. For instance, a valid resource path is https://etherscan.io/address/0x00000000219ab540356cbb839cbe05303d7705fa#code. The retrieve method returns a Promise that resolves to the Response object to that request, whether it is successful or not. Once a Response object is retrieved, the HTML code contained in the Response object is processed to remove the HTML Tags to obtain just the Solidity source code of the smart-contract. The Solidity source code is processed by the backend Parser.

D. Data Parsing

The parsing operation allows the back-end to parse the Solidity source code and produces a tree data structure in which each node stores an object of key-value pairs. For instance, the smart contract "renounceOwnership" method has the following nodes of key-value pairs.

```
{
  "type":"FunctionDefinition",
  "name":"renounceOwnership",
  "visibility":"public",
  "isConstructor":false,
  "isReceiveEther":false,
  "isFallback":false,
  "isVirtual":true,
  "stateMutability":null
}
```

E. Data Writing

The writing operation allows the back-end to write two versions of UML class diagrams. The first version is similar to the UML class diagram of OOP languages, such as Java and C++. This version of diagram contains information such as the smart contract's attributes, the smart contract's functions, and the relationship type among different smart contracts. The second version is the augmented UML diagram. It displays additional information compared to the first-version diagram, such as information about the presence of "Function Modifer" and "Function Fallback" which are not part of the current unified modeling language specification.

Based on previous studies on graphical representations in different fields [35], multiple representations of the same smart contract source code have been preferred over the traditional UML class diagram. This technique allows to quickly visualize complex systems and guarantees better readability. Based on the users' actions, the backend will generate the corresponding augmented version of the UML class diagram.

IV. Results and Discussion

Figure 1 displays the UML class diagram of a smart contract, following the current UML specification whose details are publicly available at (https://www.omg.org/spec/UML/About-UML/). There are two limitations of this representation type. First, it is very difficult to give a meaningful and comprehensive representation of the DApp in a screen, because is made of many smart contracts having different relationships among each other. Indeed, the UML diagram should display many classes, each one corresponding to different smart contracts. A recent study[36] shows that many DApps are made of a number of smart contracts greater than 10, which makes the graphic representation difficult. Moreover, the diagram generated via the UML specification hides some important details which are typical of smart contracts. Indeed, the Solidity programming language, used to write the smart contracts, owns specificities that no other OOP language owns. The current UML class diagram specification does not include specific features and constructs, such as "Receive Ether Function", "Function Modifer", "Function Fallback" and "Pure Function". These specific features could be very important for smart contracts' developers to have a quick insight on the smart contract's specific features. So, as to what concerns Q1, "Are the UML Class diagrams used by the most popular OOP languages sufficient to visually represent the smart contracts' source code?", the answer is negative.

A study regarding different methods to visually represent the source code has been investigated. Some of them involve metaphors, such as the metaphor of the city and the metaphor of the planet solar. This type of visualization could better fit the structure of DApps, because most of them are made of different smart contracts: a recent study[37] shows that 20% of the DApps deployed in the Ethereum blockchain are made of 10 or more smart contracts. In the study an interactive visualization has been proposed to improve the existing web-based tools that provide a static representation of the smart contracts' source code. In this kind of existing tools, the developer must use the horizontal and vertical scroll bar to see the diagram from the left to the right and from up to down, and viceversa. Moreover, the user needs to check the source code to understand the details of the implementation.

As to what concerns Q2, "Is there a better way to visually represent the smart contracts' source code?", the paper argued that the novelty of the tool "Smart Graph" is precisely the fact that it allows to visualize additional information, which was not provided by traditional UML class diagrams [38]. Indeed, in "Smart Graph" the user can drill down into a specific function, to look for information specific to the Solidity programming language. For instance, Figure 3 shows a tree diagram displaying all information related to a particular smart contract's function. In this way the developer does not need to inspect the source code, but s/he can get all the information s/he needs just by interacting with the visual representation.

V. Conclusion

The paper presented a tool, "Smart-Graph", that produces UML diagrams from source code in Solidity. "Smart-Graph" is a target-oriented tool for experts in the blockchain domain and it provides graphical representations which were already tested in other contexts, i.e. the UML diagram classes. However, when compared to previous UML diagrams, Smart-Graph provides an augmented version of the graphical representation. Indeed, "Smart-Graph" allows the user to inspect the class diagram to look for further details, such as "Fallback Function" or "Function Modifier", which are not accessible in the traditional UML class diagrams. Future research should be dedicated to the development of the tool, which could also provide visual representations of smart contracts based on metaphors, thus also targeting people who are not experts in the software development domain.

References

[1] C. Shen and F. Pena-Mora. Blockchain for cities—a systematic literature review. *IEEE Access*, 6:76787–76819, 2018.

[2] A. de Villiers and P. Cuffe. A three-tier framework for understanding disruption trajectories for blockchain in the electricity industry. *IEEE Access*, 8:65670–65682, 2020.

[3] G. A. Pierro and Henrique Rocha. The influence factors on ethereum transaction fees. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 24–31. IEEE, 2019.

[4] Stefan Schmidt, Marten Jung, Thomas Schmidt, Ingo Sterzinger, Günter Schmidt, Moritz Gomm, Klaus Tschirschke, Tapio Reisinger, Fabian Schlarb, Daniel Benkenstein, et al. Unibright-the unified framework for blockchain based business integration. *White paper, April*, 2018.

[5] Scott Tilley and Shihong Huang. A qualitative assessment of the efficacy of uml diagrams as a form of graphical documentation in aiding program understanding. In *Proceedings of the 21st Annual International Conference on Documentation*, SIGDOC '03, page 184–191, New York, NY, USA, 2003. Association for Computing Machinery.

[6] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, DocEng '02, page 26–33, New York, NY, USA, 2002. Association for Computing Machinery.

[7] Scott R. Tilley, Hausi A. Müller, and Mehmet A. Orgun. Documenting software systems with views. In *Proceedings of the 10th Annual International Conference on Systems Documentation*, SIGDOC '92, page 211–219, New York, NY, USA, 1992. Association for Computing Machinery.

[8] Cris Kobryn. Will uml 2.0 be agile or awkward? *Commun. ACM*, 45(1):107–110, January 2002.

[9] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a formalization of the unified modeling language. In *European Conference on Object-Oriented Programming*, pages 344–366. Springer, 1997.

[10] Jan Hendrik Hausmann and Stuart Kent. Visualizing model mappings in uml. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, page 169–178, New York, NY, USA, 2003. Association for Computing Machinery.

[11] Kenneth Baclawski, Mieczyslaw K Kokar, Paul A Kogut, Lewis Hart, Jeffrey Smith, Jerzy Letkowski, and Pat Emery. Extending the unified modeling language for ontology development. *Software and Systems Modeling*, 1(2):142–156, 2002.

[12] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding asynchronous interactions in full-stack javascript. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1169–1180. IEEE, 2016.

[13] Silvia Crafa, Matteo Di Pirro, and Elena Zucca. Is solidity solid enough? In *International Conference on Financial Cryptography and Data Security*, pages 138–153. Springer, 2019.

[14] Giuseppe A. Pierro, Henrique Rocha, Roberto Tonelli, and Stéphane Ducasse. Are the gas prices oracle reliable? a case study using the ethgasstation. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 1–8. IEEE, 2020.

[15] Chris Dannen. *Introducing Ethereum and solidity*, volume 1. Springer, 2017.

[16] Juha Sorva, Ville Karavirta, and Lauri Malmi. A review of generic program visualization systems for introductory programming education. *ACM Trans. Comput. Educ.*, 13(4), November 2013.

[17] Rajiv D Banker, Gordon B Davis, and Sandra A Slaughter. Software development practices, software complexity, and software maintenance performance: A field study. *Management science*, 44(4):433–450, 1998.

[18] Bernard Charlin, Stuart Lubarsky, Bernard Millette, Françoise Crevier, Marie-Claude Audétat, Anne Charbonneau, Nathalie Caire Fon, Lea Hoff, and Christian Bourdy. Clinical reasoning processes: unravelling complexity through graphical representation. *Medical education*, 46(5):454–463, 2012.

[19] S. Ducasse and M. Lanza. The class blueprint: visually supporting the understanding of glasses. *IEEE Transactions on Software Engineering*, 31(1):75–90, 2005.

[20] M. Mirakhorli and J. Cleland-Huang. Detecting, tracing, and monitoring architectural tactics in code. *IEEE Transactions on Software Engineering*, 42(3):205–220, 2016.

[21] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 167–176, 2005.

[22] J. I. Maletic, J. Leigh, A. Marcus, and G. Dunlap. Visualizing object-oriented software in virtual reality. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pages 26–35, 2001.

[23] P. Caserta and O. Zendra. Visualization of the static aspects of software: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7):913–933, 2011.

[34] Gustavo A Oliva, Ahmed E Hassan, and Zhen Ming Jack Jiang. An exploratory study of smart contracts in the ethereum blockchain platform. *Empirical Software Engineering*, pages 1–41, 2020.

[24] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on uml class diagrams. *Artificial intelligence*, 168(1-2):70–118, 2005.

[25] Marcela Genero, Mario Piattini, and Coral Calero. A survey of metrics for uml class diagrams. *Journal of object technology*, 4(9):59–92, 2005.

[26] R. Wettel and M. Lanza. Visualizing software systems as cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99, 2007.

[27] George Lakoff. The contemporary theory of metaphor. 1993.

[28] Barbara Tversky. Spatial schemas in depictions. In *Spatial schemas and abstract thought*, pages 79–111, 2001.

[29] Kari Smolander. Four metaphors of architecture in software organizations: finding out the meaning of architecture in practice. In *Proceedings International Symposium on Empirical Software Engineering*, pages 211–221. IEEE, 2002.

[30] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 551–560. ACM, 2011.

[31] H. Graham, H. Y. Yang, and R. Berrigan. A solar system metaphor for 3d visualisation of object oriented software metrics. In *InVis.au*, 2004.

[32] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96, 2016.

[33] Hans-Erik Eriksson, Magnus Penker, Brian Lyons, and David Fado. *UML 2 Toolkit, CafeScribe*, volume 26. John Wiley & Sons, 2003.

[35] G.A. Pierro, Alexandre Bergel, Roberto Tonelli, and Stéphane Ducasse. An interdisciplinary model for graphical representation. In *International workshop on Cognition: Interdisciplinary Foundations, Models and Applications*, 2020.

[36] M. Marchesi et al. An organized repository of ethereum smart contracts' source codes and metrics. *Future Internet*, 12(11):197, Nov 2020.

[37] G.A. Pierro, Roberto Tonelli, and Michele Marchesi. Smart-corpus: an organized repository of ethereum smart contracts source code and metrics, 2020.

[38] G. A. Pierro, A. Bergel, R. Tonelli, and S. Ducasse. An Interdisciplinary Model for Graphical Representation. In *CIFMA 2020 - 2nd International Workshop on Cognition: Interdisciplinary Foundations, Models and Applications*, Amsterdam / Virtual, Netherlands, October 2020.