Supporting Reuse of Smart Contracts through Service Orientation and Assisted Development

Luca Guida
Politecnico di Milano
Milan, Italy
luca.guida@mail.polimi.it

Florian Daniel
Politecnico di Milano
Milan, Italy
florian.daniel@polimi.it

***Abstract*—This paper focuses on two of the key challenges a developer encounters when reusing smart contracts: finding actionable information about existing smart contracts (descriptors) and writing the necessary integration logic to invoke selected contracts and implement missing functions. As for the former issue, the paper proposes a smart contract description format that allows the developer to search for publicly available contracts, understand which features a contract exposes and how to invoke them, according to a service-oriented approach. For the latter, the paper implements a simple, model-driven development environment consisting in a visual programming editor that provides a set of modeling constructs encoding specific, reuse-oriented code patterns. The approach is implemented and demonstrated in the context of the blockchain platform Ethereum and its programming language Solidity. The results obtained show that the proposed approach can be beneficial in the development of composite smart contracts and generic blockchain applications.**

I. Introduction

A *blockchain* is a shared, distributed ledger, that is, a log of transactions that provides for persistency and verifiability of transactions [1]. *Transactions* are cryptographically signed instructions constructed by a user of the blockchain [2], for example the transfer of cryptocurrency from one account to another. *Smart contracts*[1] [3] enable the blockchain to perform computations, for example to decide whether to release a given amount of cryptocurrency upon the satisfaction of a condition agreed on by two partners. If we move from one smart contract to multiple collaborating smart contracts, we turn the blockchain into a distributed computing platform [4]. As this was not the original intention of blockchain, we are however far from a mature support for distributed computing.

In this paper, we look at the problem of distributed computing in blockchains from a *service-oriented* perspective [5], in which smart contracts act as reusable services that can be integrated to develop new, value-adding contracts. However, while smart contracts present significant opportunities and also a need for reuse and contract interactions [6], some infrastructural elements that have proven useful in service-oriented computing do not yet have equivalents in state-of-the-art blockchain technology. For instance, the technology lacks (i) *abstract descriptors*, which allow developers to understand the functionality of a contract without having to read the actual code; (ii) a *descriptor registry*, which allows them to search for and discover contracts for reuse; and (iii) a dedicated, reuse-oriented *composition paradigm*, which provides them with contract-oriented programming constructs [6].

In service-oriented computing, a service descriptor is a standardized description of a service, a machine-readable set of metadata that identifies the endpoints of the service, the communication protocols, the operations, the message formats, the server methods, and the usage policies of the service. That is, it encapsulates all the information that a developer must know to access and use the service [7]. Typical description formats are the Web Services Description Language (WSDL, http://www.w3.org/2002/ws/desc/) or the Web Application Description Language (WADL, https://www.w3.org/Submission/wadl/). Descriptors can be made public for instance via registries following the Universal Description, Discovery and Integration (UDDI, http://www.uddi.org/pubs/uddi-v3.0.1-20031014.htm) specification, which provides a standard protocol for publishing and searching for services. The most prominent standard for the composition of web services is the Web Services Business Process Execution Language (WS-BPEL, http://www.oasis-open.org/committees/wsbpel/), which is also equipped with a variety of graphical editors for visual service composition.

In absence of similar infrastructure services for smart contracts, reuse in smart contract development remains underexplored. In addition, implementing correct contracts requires not only coding skills and proficiency in the programming languages of the chosen blockchain platform, but also vast knowledge about the functioning mechanisms of the specific blockchain network. This makes the creation of smart contracts complex and error prone. For instance, Atzei et al. [8] show that already today even simple smart contracts often suffer from a variety of security vulnerabilities; Nikolić et al. [9] show that several of the smart contracts deployed on Ethereum either "lock funds indefinitely, leak them carelessly to arbitrary users, or can be killed by anyone;" and Singh and Chopra [10] discuss socio-technical limitations of smart contracts: lack of control, lack of understanding and lack of social meaning.

With this paper, we aim to lay the foundation for a discussion on service-orientation for smart contracts by proposing:

- a smart contract *descriptor model and format* for the abstract description and publication of reusable contracts;
- a prototype of a *smart contract registry* for the search and discovery of smart contracts for both human users (Web user interface) and software agents (RESTful API);

[1]For simplicity, we use "smart contract" and "contract" as synonyms.

- a *visual development environment* for the model-driven development of smart contracts with a special focus on the integration of smart contracts and on code patterns – both aiming at easing reuse.

The paper specifically focuses on Ethereum, the most used blockchain platform for smart contracts, and on Solidity, Ethereum's programming language for smart contracts.

Next, we therefore provide some more details on Ethereum and then refine the goals of our work. Then, we discuss the description format (Section III) and the metamodel of the composition environment (Section IV). In Section V we implement the registry and graphical editor, then we describe a case study (Section VI). Before closing the paper, we discuss related works in Section VII.

II. Background and Objectives

A. Scenario

Let's consider the following scenario of the insurance sector inspired by the opportunities outlined by Gatteschi et al. [11]. The goal is to implement a so-called "parametric insurance" contract, *SmartTrainInsurance*, which reacts to a triggering event, i.e., the dissatisfaction of a minimum service level agreement by a train company. A customer interested in the insurance pays an extra premium when buying his/her monthly pass, and, if during the respective month the cumulative delays of the trains on the chosen route exceed a given limit, a compensation is automatically transferred to the customer. Service levels are checked monthly using a trustworthy API with train timetable information. Once triggered, the smart contract compensates all insured customers at once.

Railway companies have a punctuality commitment with their customers: in case of delayed trains, travellers are entitled to request a compensation. Today, the complexity of claim procedures often discourages passengers to submit a claim. This phenomenon is even more bothersome to commuters, who typically buy seasonal or monthly passes for which the indemnification process is usually more intricate than for individual tickets. A smart contract insurance may thus help lower complexity and automate the indemnification process.

In order to provide the service specified, the smart contract reuses three externals contracts and one library:

- *Ownable*, third-party contract inherited to reuse basic authorization control mechanisms, such as the ones that restrict access to some functions of the contract to the owner of the contract only, as well as to make possible ownership transfer operations;
- *Pausable*, third-party contract inherited to integrate a basic emergency stop mechanism that allows the owner of the contract to pause and unpause calls to critical functions, such as the registration of new policies;
- *usingOraclize*, contract inherited to access *Oraclize* (http://www.oraclize.it), an oracle service with access to external APIs and string and response processing functions;
- *SafeMath*, library called to make math operations safe, e.g., to prevent overflows or underflows with integer operations.

Ownable, *Pausable* and *SafeMath* can be retrieved from the *OpenZeppelin* library (https://openzeppelin.org/).

If the insurance contract is hand-coded, the reuse of these external contracts easily becomes time-consuming and error-prone. For example, in order to reuse a publicly deployed smart contract, a developer first of all requires information about the endpoint, the Ethereum address, of the contract. Then, for the developer to understand which specific functions of a contract to invoke or to inherit, he/she needs to have full access to the code of the respective contract and to manually go through it. Next we show that these tasks are everything but trivial.

B. Smart Contract Description and Discovery

A smart contract on Ethereum is equipped with *contract metadata*, a JSON file with information about the contract. It provides details about the compiler used, the interfaces exposed by the contract in terms of a so-called *Application Binary Interface* (ABI), and documentation about the contract in *Ethereum Natural Specification Format* [12]. To make contract metadata publicly accessible, it is typically published on *Swarm*, a component of Ethereum Web 3 which works as a redundant and decentralized store of Ethereum's public record.

Although these metadata provide some insight, several relevant details required for reuse are still missing, such as: (i) version of the contract; (ii) type of the contract (library, data, generic, oracle [6]); (iii) deployment information, if the contract was deployed on a public Ethereum network (address of the contract, ID of the network, ID of the chain)

The ID of the network identifies the Ethereum network where a contract is deployed. For instance, the Ethereum main network, often referred to as *Ethereum mainnet*, has network ID equal to "1"; the *Ropsten* test network is identified by a "3", while other test networks, *Rinkeby* and *Kovan*, by a "4" and a "42" respectively [13]. The ID of the chain distinguishes the Ethereum (ETH) and the Ethereum Classic (ETC) networks which both have "1" as their network ID: ETH has chain ID equal to "1", while ETC has chain ID "61" [14].

For the discovery of smart contracts, three categories of services offer basic repository or a service registry features:

- contract name resolution services, such as the *Ethereum Name Service*;
- package registration services, such as the *Ethereum Package Registry* (EthPM);
- Ethereum decentralized applications (DApps) directories.

The Ethereum Name Service has a similar function to the Internet's *Domain Name Service* (DNS). It uses dot-separated, hierarchical names known as *domains* to *resolve* human-readable names of contracts, like "contract.eth", into machine-readable identifiers, such as the Ethereum address or the Swarm hash of the contract [15].

The Ethereum Package Registry is an index for Ethereum smart contract packages based on the ERC190 Smart Contract Packaging Specification [16]. It provides the means to publish and consume smart contract packages in a convenient way, as it is supported by many development tools, such as *Truffle*, the most popular development framework for Ethereum.

A smart contract is typically not used "as is," but as a component of a larger software solution. A common use case is the integration of a smart contract in a decentralized application, a *DApp*. DApps may be distributed through specific DApps directories and marketplaces, in a similar way to mobile apps which are distributed through app marketplaces. Some of the most popular DApps directories are *State of the DApps* (http://www.stateofthedapps.com) and *Modex* (http://market.modex.tech).

C. Smart Contract Reuse

The most common way of code reuse in Solidity, the programming language for writing Ethereum smart contracts, is *inheritance* from one or more given contracts. When inheriting from a contract, that contract's source code is copied into the code of the new contract; polymorphism is supported [17]. Inheritance typically happens from *abstract contracts*, for which at least one function lacks an implementation. *Interfaces* are contracts that do not have any implemented function, that cannot inherit from other contracts or interfaces, and that do not define variables, structs, enums or constructors: they only include function signatures to be refined [17].

Another type of reuse is represented by *library function calls*. *Libraries* are contracts that do not maintain any state and that are deployed only once at a specific address. Library function calls (using the `DELEGATECALL` operation of the Ethereum Virtual Machine) execute the code of the called function in the context of the calling contract, making the storage and state variables of the calling contract accessible to the called function. In other terms, the calling contract loads code at runtime from the target contract. This may expand the attack surface of the calling contract as it allows the called contract to access and potentially tamper with the calling contract's storage [18].

D. Objectives

The objective of the research described in this paper is to overcome some of the aforementioned limitations to the description, discovery and integration of smart contracts. In particular, we aim to develop an *abstract description format* to support:

- the differentiation of contract types using specific fields of the descriptor;
- the encapsulation of endpoint information, in order to specify how a public smart contract can be invoked.

We further aim to accompany the description format with a dedicated *registry* to store smart contract descriptors, enabling effective search and discovery by allowing developers to navigate through available documentation and metadata.

Finally, we propose a *smart contract editor* for the creation of composite smart contracts based on the principles of *visual programming* [19] that:

- is integrated with the smart contract registry and fosters reuse; and
- eases the process of writing new code for contract integration and for the implementation of additional functions.

III. Smart Contract Description

A smart contract descriptor should encapsulate all the information required to effectively access an Ethereum smart contract. We identify the following five main requirements:

1) It should accommodate all the *default metadata* already produced by the Ethereum compiler during compilation, such as the specification of the interfaces of functions, constructors and events defined in the contract, documentation for users and developers, technical details about the compiler and the libraries used in the contract.
2) It should contemplate *generic descriptive information*, such as the name of the contract, the author, the programming language used, the type of contract, which are needed to support search and discovery.
3) It should further contemplate all the *technical details* needed to actually connect to and interact with the contract, such as endpoint addresses, Ethereum network IDs and chain IDs.
4) The descriptor should be able to describe both *deployed and non-deployed* Ethereum contracts, meaning that it should provide details about the endpoint only for contracts which were deployed on a Ethereum-based blockchain network.
5) The smart contract descriptor should be encoded in a *lightweight data-interchange format*, easy for machines to process and for developers to understand.

The information above provides the means to successfully call a contract or library, but it does not make it possible to inherit the contract, because inheritance in Solidity requires access to the complete source code of the contract to be inherited. For this reason, when available, the source code may be provided together with the descriptor in order to enable the implementation of inheritance scenarios.

A. Descriptor Model and Format

Given the previous requirements, the *model* (schema) we propose for smart contract description is illustrated in Figure 1. The proposal re-arranges metadata provided by the Solidity compiler and extends them with new constructs (in bold).

The model consists of information oriented toward users (`Userdoc`, `Descriptor`, `Endpoint`) and developers (`Dev`), the former interested in reusing the contract, the latter interested in extending its implementation. The property `Descriptor.author` may be empty if there is no tag `@author` in the contract's NatSpec documentation. The entity `Descriptor.ABI` may be empty if no methods, constructors or events are defined in the contract. The entities `Userdoc.Methods` and `Dev.Devdoc.Methods` may be omitted if no documentation for methods or constructors is defined in the contract. For deployed contracts, `Endpoint` contains the `address`, `network_id` and `chain_id`; if the contract is not deployed, `Endpoint` is not provided. The fields `Devdoc.author` and `Devdoc.title` may not be defined if there are no tags `@title` and `@author` in the contract NatSpec documentation. The entity `Libraries` may be empty if no libraries were used in the contract.

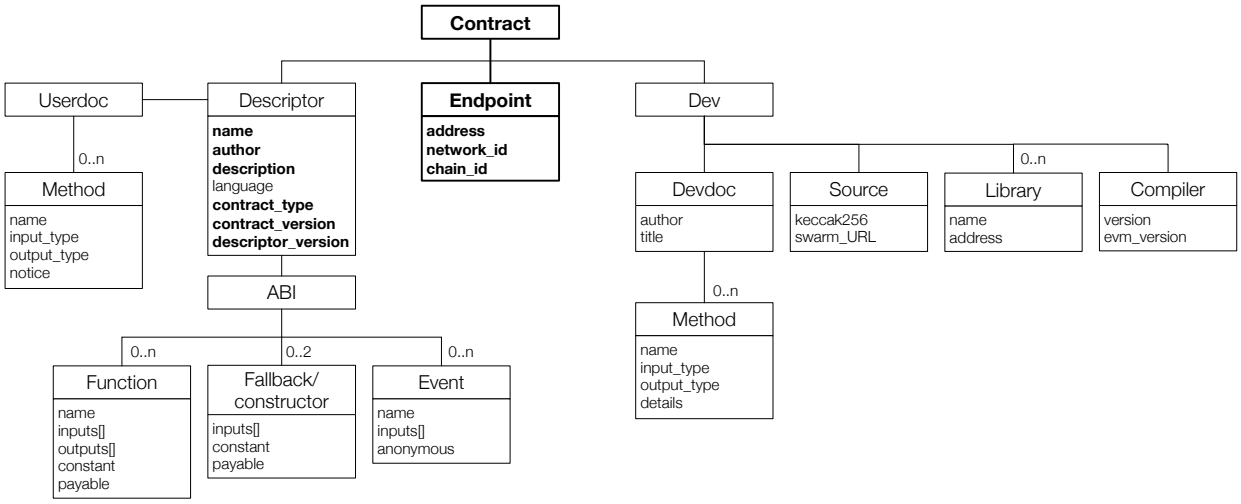

Fig. 1. Smart contract descriptor model. In bold newly defined constructs compared to the metadata automatically generated by the Solidity compiler.

As for the *format*, we propose the use of simple JSON. A possible alternative to JSON would have been XML, a widely-used standard markup language adopted for WSDL files. The main reason for the choice of JSON is that part of the descriptor is created by extracting information from the Solidity metadata file, which is already encoded in JSON.

Listing 1 shows an *example* of the JSON descriptor of the *Ownable* smart contract, one of the contracts which are reused in the context of the case study.

```
{
 "contract": {
   "descriptor": {
     "name": "Ownable",
     "author": "",
     "description": "The Ownable smart contract
         provides basic authorization control
         functions, thus simplifying the implementation
         of user permissions.",
     "language": "Solidity",
     "contract_type": "generic_contract",
     "contract_version": "1.0",
     "descriptor_version": "1.0",
     "abi": [
       {
         "constant": false,
         "inputs": [],
         "name": "renounceOwnership",
         "outputs": [],
         "payable": false,
         "stateMutability": "nonpayable",
         "type": "function"
       },
       {
         "inputs": [],
         "payable": false,
         "stateMutability": "nonpayable",
         "type": "constructor"
       },
       {
         "anonymous": false,
         "inputs": [
           {
             "indexed": true,
             "name": "previousOwner",
             "type": "address"
           }
         ],
         "name": "OwnershipRenounced",
         "type": "event"
       }
     ],
     "userdoc": {
       "methods": {
         "renounceOwnership()": {
           "notice": "Renouncing to ownership will
               leave the contract without an owner."
         }
       }
     }
   },
   "endpoint": ,
   "dev": {
     "devdoc": {
       "methods": {
         "renounceOwnership()": {
           "details": "Allows the current owner to
               relinquish control of the contract."
         }
       },
       "title": "Ownable"
     },
     "sources": {
       "keccak256": "0
           x84c7090c27cf3657b73d9e26b6b316975fa0
           bd233b8169f254de0c3b3acfaefc",
       "swarm_URL": "bzzr://
           b983355647976c1daa5de581a1b6a41
           be9c5adc17cce257b8679649db78f8a11"
     },
     "libraries": {},
     "compiler": {
       "version": "0.4.24+commit.e67f0147",
       "evmVersion": "byzantium"
     }
   }
 }
}
```

Listing 1. Fragment of the smart contract descriptor of the *Ownable* smart contract. In bold newly defined JSON objects compared to the metadata automatically generated by the Solidity compiler.

From the descriptor, a developer can infer that the `Ownable` smart contract defines a non-payable function named `renounceOwnership`, a constructor, and an event named `OwnershipRenounced` with a `previousOwner` parameter of type address. Additional details come in the form of contract documentation and technical specifications.

IV. Composite Smart Contract Development

The development of composite smart contracts may be supported by a visual editor integrated with a smart contract registry. We identify the following requirements that affect the respective development paradigm:

- It should allow the user to use a collection of *visual constructs* that represent code constructs like variables, function definitions and calls, logical expressions, events, modifiers and other relevant Solidity statements.
- Visual constructs should be able to represent both standard *Solidity instructions* and more complex *Solidity patterns*; the latter aim to provide high value with only few clicks.
- It should enable the user to create smart contracts by choosing visual constructs from a palette of constructs and dragging and dropping them onto a *modeling workspace*. Visual contract should be configurable and allow one to enter relevant information, e.g., the name of variables, function arguments, comments, ...
- It should support the *automatic generation* of the Solidity code corresponding to the visual constructs selected by the user.
- It should support the *integration with external oracle services* for Ethereum in order to allow users to access external data feeds in the context of their smart contracts.

In the following, we address these requirements.

A. Compositon Metamodel

Visual programming paradigms commonly leverage on model-driven development techniques [20], which may present different granularities of abstraction – from constructs representing low-level, syntactic elements, such as variables or assignment operators, to constructs representing full-fledged code templates that only need to be configured. We propose an intermediate level of abstraction to represent both low-level Solidity instructions and aggregate combinations of instructions, i.e., patterns, that developers can use to code smart constructs without any restriction. A modeling paradigm that suits these needs is block-based modeling (blocks are similar to pieces of a puzzle), as supported by Google's Blockly framework (http://developers.google.com/blockly).

As shown in literature [21], compared to text-based programming, blocks leverage on recognition instead of recall: the user just has to recognize the block he/she needs among the ones available in the visual editor without having to retrieve from his/her memory all the constructs that he may use. In other terms, instead of requiring the developer to remember the full vocabulary of Solidity, he/she can benefit from a user-friendly palette that is persistent on the screen and self-explaining. Figure 2 illustrates the metamodel of the proposed, block-based modeling paradigm for Solidity smart contracts.

A contract is represented as a set of nested blocks with properties and respective values for their configuration. Which block fits into which other block depends on the shapes of the blocks. The shapes assure that constructed constructs are syntactically correct. One starting block determines the overall shape of a smart contract. In addition to their shape, blocks may have an associated color to further highlight their purpose and respective descriptive labels. For example, blocks representing values are gray and have a nub, variable assignment blocks are blue and have a notch. Together, these features make block-structured code self-explanatory.

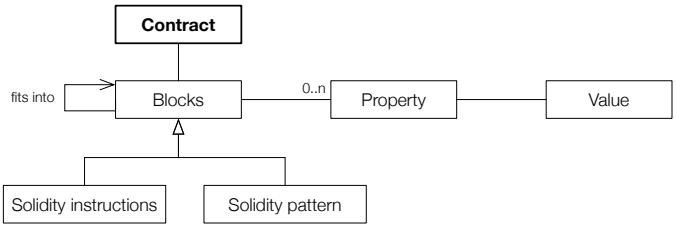

Fig. 2. Metamodel of the block-based composition language for composite smart contracts. In bold the Solidity patterns identified and supported.

We distinguish two distinct categories of blocks, *Solidity instructions* and *Solidity patterns*, which specialize the described metamodel.

B. Solidity Instructions

The Solidity instructions class contains all the native constructs of the Solidity language, e.g. contract documentation, definition and usage of variables, modifiers, events, functions. Figure 3 illustrates the metamodel of the supported instructions and how they can are aggregated using blocks. There are blocks or configurations for all Solidity instructions (as of October 2018).

C. Solidity Patterns

The availability of blocks implementing typical patterns as prefabricated aggregations of instructions enables the user to quickly implement some frequently-used operations. For instance, the invocation of *Oraclize* and the definition of the related `callback` function, which typically consist of at least 10 lines of code, can be condensed into a single block, requiring the developer to fill in only few configuration parameters.

Is illustrated in Figure 4, we distinguish two types of patterns: reuse patterns and shortcut patterns.

1) Reuse patterns: These specifically enable and promote the reuse of existing smart contracts and libraries; specifically, the reuse patterns support:

- invocation of functions belonging to external smart contracts; for instance, calls to mathematical functions belonging to the *SafeMath* external library contract, as required in our case study;
- binding of the functions defined in an external smart contract to a specific variable type of the contract being edited (also known as `using... for...` construct); for example, the developer may want to use a specific external contract when operating with integer variables;
- inheritance of functions, events, modifiers and state variables from an external smart contract; for example the *SmartTrainInsurance* contract inherits basic authorization control and emergency stop mechanisms from the *Ownable* and *Pausable* contracts;
- invocation of an oracle service to query an external API for immediate feedback;

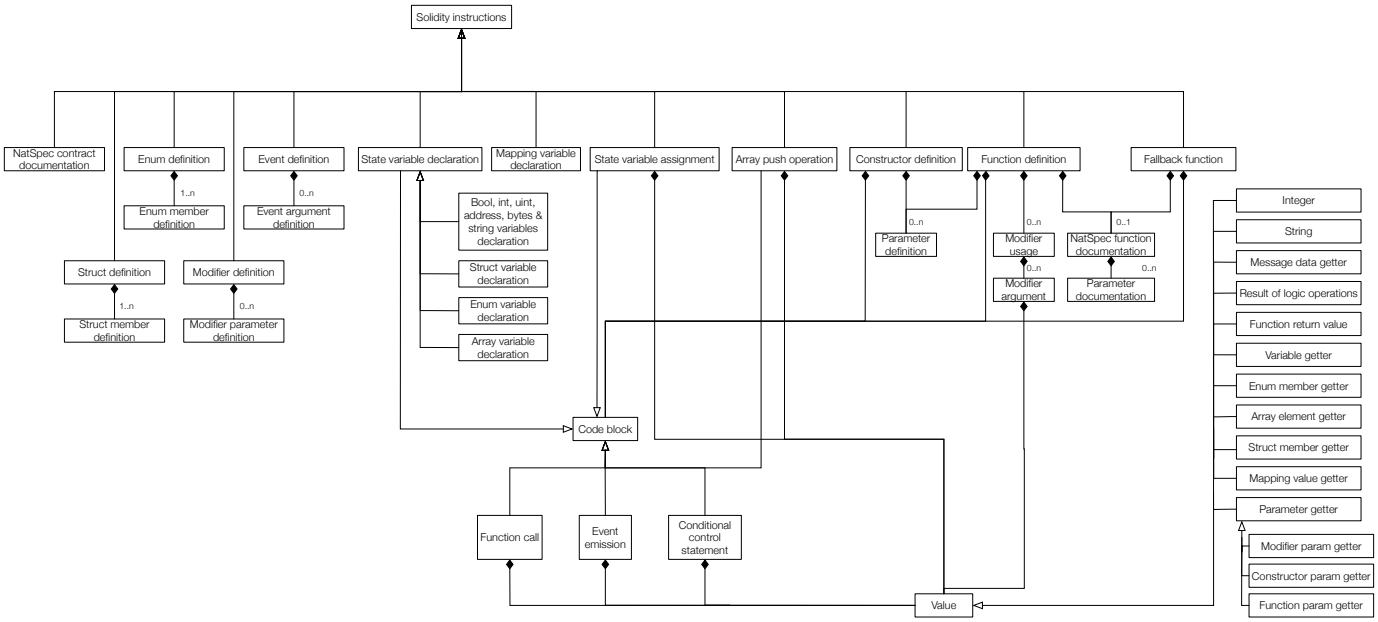

Fig. 3. Metamodel of supported basic Solidity instructions specializing the generic metamodel in Figure 2

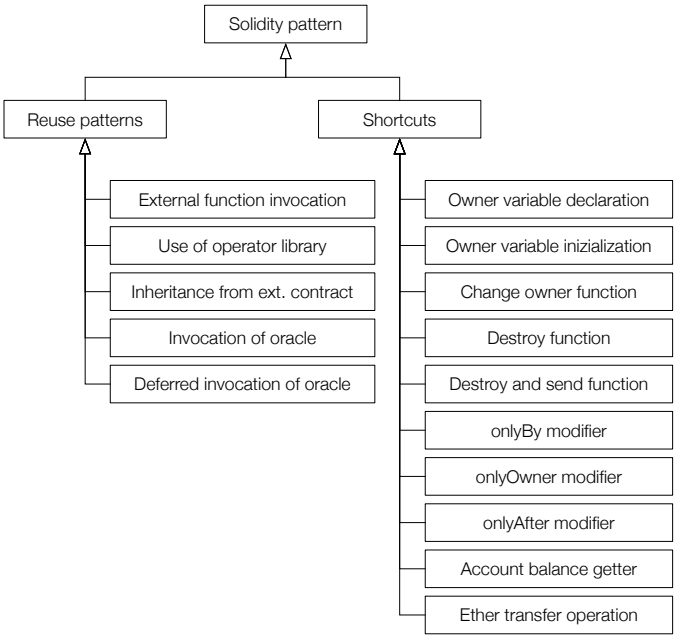

Fig. 4. Metamodel of supported, high-level code patterns specializing the generic metamodel in Figure 2

- invocation of an oracle service to schedule a query to be performed at a specific time in the future, as required by the *SmartTrainInsurance* contract to query the train punctuality API right after the expiration date of the policy.

2) Shortcut patterns: These represent generic, commonly used patterns of the Solidity language inspired by the list of *Solidity Common Patterns* [22] and by the *OpenZeppelin* library [23]. The supported shortcuts are:

- `owner` variable declaration, used to specify who is the owner of a smart contract;
- initialization of the previously declared `owner` variable to value `msg.sender`;
- `changeOwner` function definition, used to change the owner of a smart contract;
- `destroy` function definition, used to permanently stop an existing smart contract and send the remaining Ether to the owner of the contract;
- `destroyAndSend` function definition, used to permanently stop an existing smart contract and send the remaining Ether to a recipient to be specified as a parameter;
- `onlyBy` modifier definition, used to restrict the usage of a function only to a specified account;
- `onlyOwner` modifier definition, used to restrict the usage of a function only to the owner of the contract;
- `onlyAfter` modifier definition, used to restrict the usage of a function only after a specified amount of time has passed;
- Ethereum account balance getter, used to retrieve the balance of an Ethereum account;
- Ether transfer operation, used to transfer Ether to a specified account.

V. Implementation

The current proof-of-concept prototype comprises a smart contract registry and a web-based visual editor.

A. Smart Contract Registry Prototype

The Ethereum smart contract registry, hereafter called *SolidityRegistry*[2] is made of three main modules: a registry website

[2]Source code available at http://github.com/LucaGuida/SolidityRegistry

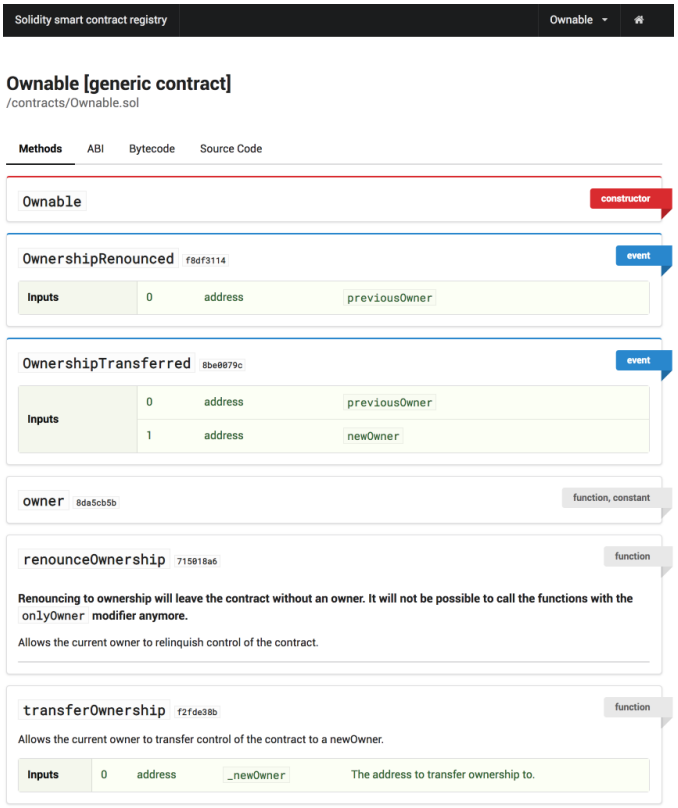

Fig. 5. Screenshot of the SolidityRegistry responsive website

generator, a smart contract descriptor generator, and a registry API. The registry comes in the form of a responsive website that makes the smart contract descriptors browsable by users and an API that allows programmatic interactions. In its first version, the registry is automatically generated from a set of given smart contracts.

The *website generator* is in charge of compiling the smart contracts and generating the respective HTML and CSS code. It is implemented starting from an existing documentation generator for Solidity called *Doxity* [24]. As shown in Figure 5 for the *Ownable* smart contract, each web page displays a sidebar on the left from which the user can navigate to other contracts, and a central area containing information about the selected contract. The central frame displays name and type of the contract, Ethereum endpoint (for deployed contracts), and four tabs: *Methods*, *ABI*, *Bytecode* and *Source Code*.

The *smart contract descriptor generator*, the one in charge of generating a descriptor for each smart contract stored in the registry, is implemented from scratch as a Python script using libraries belonging to the *Python Standard Library* (`os`, `json`, `shutil` and `distutils`). The module parses the metadata files generated by the Solidity compiler and creates JSON descriptors compliant with the format defined before.

The *registry API* is designed in order to enable any external application compatible with the HTTP protocol to access both source code and descriptors stored in the registry. It is implemented as a common REST API.

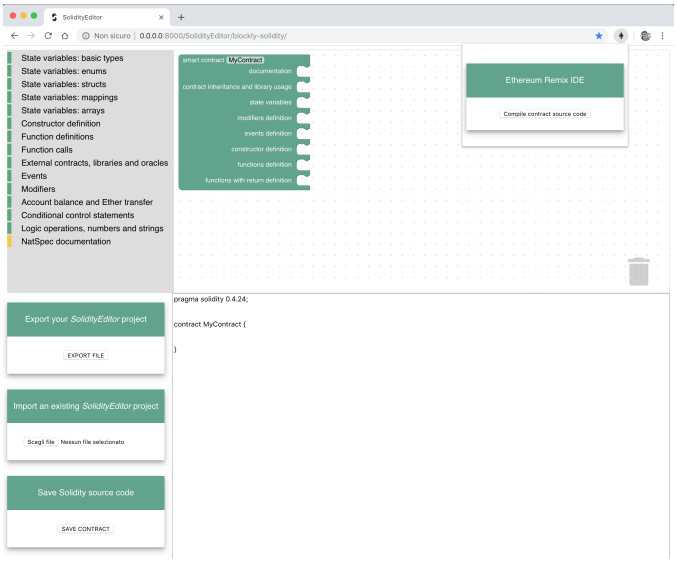

Fig. 6. Screenshot of SolidityEditor after loading it

B. Visual Programming Environment

The implementation of the programming environment is more sophisticated and subject to a set of crucial functional requirements. Next to supporting the block-based modeling paradigm described in Section IV, the goal of the implementation was to develop an editor that:

- is able to *retrieve information* from the smart contract registry about third-party contracts or libraries that users may want to re-use with *calls* or with the `using...for...` construct in their smart contracts;
- allows the user to *export* the generated Solidity code to an external IDE, such as *Ethereum Remix* (http://remix.ethereum.org), in order to compile it, test it and submit it for static code analysis (compiler warnings);
- enables the user to *store* the Solidity code or a snapshot of the current state of the workspace (constructs with their configurations and connections) on the local machine;
- enables the user to *load* a previously saved workspace snapshot from a local file.

The design of the visual editor, hereafter called *SolidityEditor*,[3] is based on *Google Blockly* (http://developers.google.com/blockly), a framework for the creation of visual, block-based programming editors. Blockly allows developers to define new custom blocks representing code constructs and to write a code generator for each them: as a result, end-users can benefit of an environment in which code is generated automatically while they connect blocks in the workspace.

Alternatives to Google Blockly we considered are *OpenBlocks* (http://web.mit.edu/mitstep/openblocks.html), a Java library for creating blocks-based programming environments, and *Droplet* (http://droplet-editor.github.io) [21], a peculiar toolkit for block-based editors that enables users to interchangeably use blocks and textual code. We excluded the

[3]Source code available at http://github.com/LucaGuida/SolidityEditor.

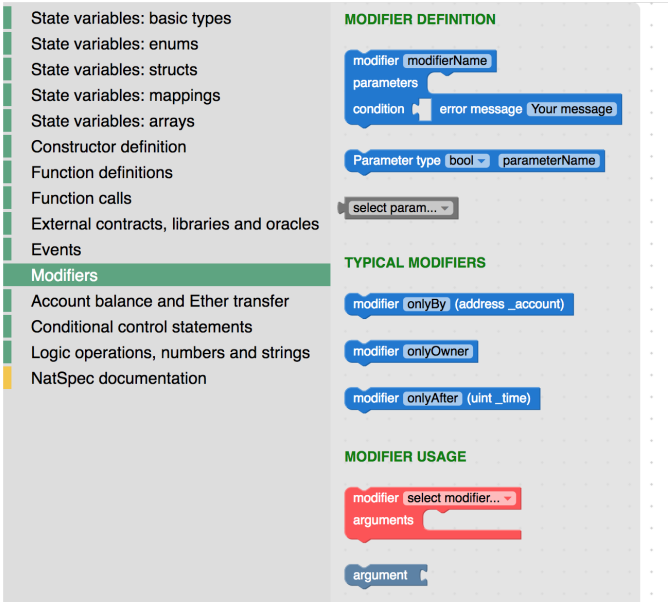

Fig. 7. Screenshot of the *Modifiers* category of SolidityEditor showing blocks representing *Solidity instructions* (modifier definition, modifier parameter definition, modifier parameter getter, modifier usage, modifier argument) and *Shortcut patterns* (onlyBy, onlyOwner, onlyAfter).

former as it outdated and no longer under development, while the latter is even less mature and adopted compared to Blockly.

The integration with *Ethereum Remix IDE* is implemented as a Google Chrome browser extension, as it requires the interchange of data among different browser tabs (editor tab and Ethereum Remix IDE tab), so it could not be implemented without a specific mechanism for enabling inter-application communication inside the browser.

As illustrated in Figure 6, SolidityEditor is composed of a main screen, a HTML web page made of five components:

- in the center of the page, the *Blocks workspace*, a white area containing the blocks the user has selected for his contract;
- on the left, a *sidebar* allows the user to navigate through the available blocks and to drag-and-drop them into the workspace;
- under the blocks workspace, a text area contains the Solidity source code of the contract generated from the blocks;
- on the bottom left, another *sidebar* allows the user to locally store the workspace, to load a previous snapshot of the workspace (feature still under development in the current version of the prototype), or to save the Solidity source code;
- on the top right, a button in the browser UI launches the Chrome extension, which provides the user with an easy way to compile the smart contract with one single click.

In order to support all Solidity instructions and patterns discussed in Section IV in SolidityEditor, we implemented 70 blocks. In addition, the editor comes with a small collection of 9 Google Blockly pre-defined blocks (if-else control state-

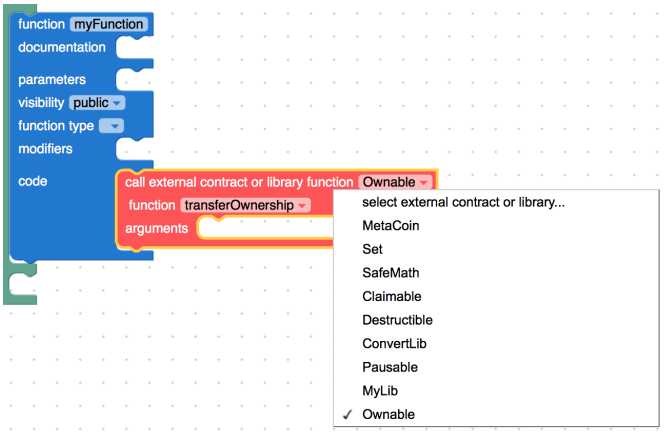

Fig. 8. Screenshot of the external contract and function selector based on the integration with SolidityRegistry

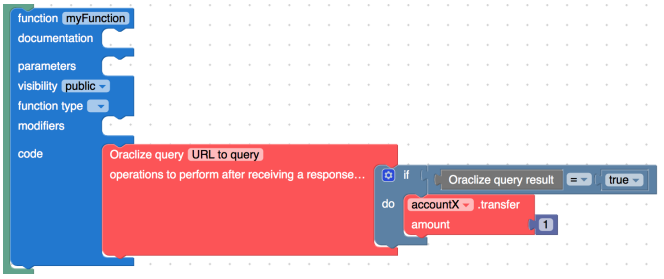

Fig. 9. Screenshot of the query and query result blocks which may be used for connecting with Oraclize

ments, logic operations, number and string input blocks). A large majority of native Solidity constructs is now available as blocks in SolidityEditor. Among the 70 custom blocks, 15 of them represent the *Reuse patterns* and *Shortcut patterns* introduced in Section IV-C, as illustrated in Figure 7.

By leveraging the integration with SolidityRegistry, users can access the registry from the editor, and then add to their workspace blocks that provide the means to call external contracts or libraries available in the registry. Figure 8 shows how to invoke the registry from the editor.

Similarly, inheritance from external contracts and the construct `using...for...` are implemented as two specific blocks that make it possible for a contract to inherit functions, events and modifiers, or to bind external libraries from SolidityRegistry.

Moreover, the developer has access to specific blocks for querying an oracle service, namely *Oraclize*: as shown in Figure 9, the *query* block makes it possible to enter the URL of an endpoint to be queried, for instance a web service providing weather data or currency conversation rates, while the *query result* block enables the use of the response.

The manipulation and composition of blocks is constrained by some basic verification mechanisms. For example, drop-down menus embedded into blocks allow the developer to select only the variables and parameters in the scope of that specific block, thus preventing out-of-scope errors or mistyped

identifiers. Similarly, when emitting an event, using a modifier, invoking a function or assigning a value to the member of a `struct` variable, dynamic drop-down menus make it easy to pick the desired element among the available ones. Further automated checks may be implemented in order to anticipate validations that typically happen at compile time.

VI. Case Study

As an informal validation of the conceived development paradigm and the respective modeling tool, we implemented the SmartTrainInsurance smart contract described in Section II-A. We recall that the contract requires reusing four third-party smart contracts: Ownable, Pausable, SafeMath and usingOraclize. In the implementation we made the assumption that the fee charged by Oraclize for the monthly query operation to the external API has a minimal impact on the balance of the contract, thus not interfering with the compensation mechanism.

For a preliminary assessment of the benefits of SolidityEditor and SolidityRegistry, we implemented the smart contract in two different ways: first, by hand; then, using SolidityEditor[4]. The manual implementation did not use the Ownable, Pausableor, and SafeMath smart contracts and instead implemented the respective functionality from scratch. Figure 10 illustrates an excerpt of the block model of the contract during development; more specifically, it represents the blocks used to inherit external contracts, and to define state variables, events and the constructor function.

SolidityEditor substantially reduced the risk of syntax errors (shorter time needed for debugging) and lowered the minimum knowledge of the programming language needed to implement a new contract (no need to consult the Solidity documentation). Also the code produced by SolidityEditor was more compact than the manually implemented one: after removing empty lines, the manually developed code counted 152 LOC, while the code produced by SolidityEditor counted 111 LOC – a 27% reduction. This result is a consequence of the increased code reuse in SolidityEditor thanks to the usage of SolidityRegistry: instead of defining typical Solidity events, modifiers and functions from scratch, the inheritance of pre-existing contracts (*Ownable* and *Pausable*) and the usage of a library (*SafeMath*) made it possible to obtain the same functionalities with less source code.

From a qualitative point of view, we also observe that the formatting style and indentation of the automatically generated code are consistent throughout the contract. Also, the usage of blocks with specific shapes and connectors enforces the generation of more structured code: similar constructs result to be clearly grouped in sections, so all the modifiers are defined together, followed by event definitions, then constructor and function definitions, and so on according to a disciplined paradigm.

[4]The Solidity code of the two versions can be inspected at https://github.com/LucaGuida/SmartTrainInsurance

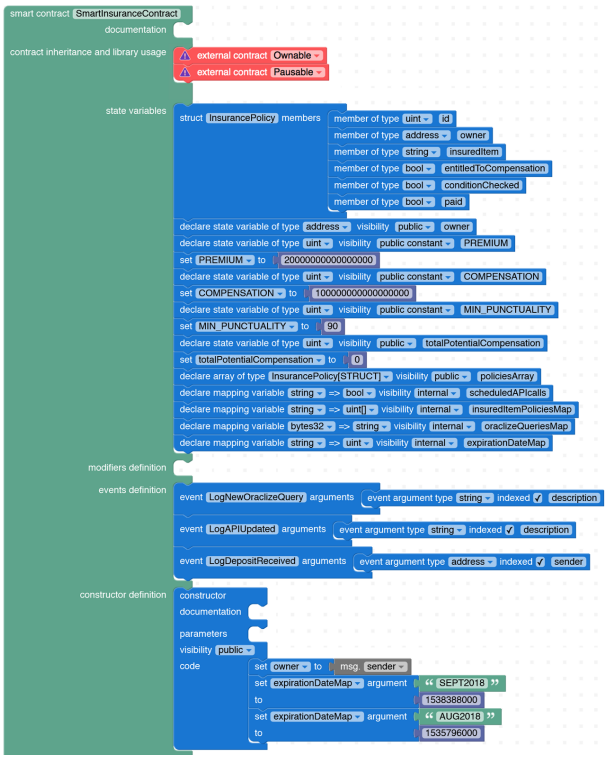

Fig. 10. Screenshot of the development process of the *SmartTrainInsurance* contract in *SolidityEditor*

VII. Related Work

A possible way to enable the creation of smart contracts by non-developers may consist in designing smart contract templates and enabling users to customize their behaviour by entering information or configuring options in a user-friendly User Interface (UI). However, as of the time of writing template-based smart contract creation tools do not support the composition of multiple smart contracts, thus jeopardize the integration and reuse of existing smart contracts. Another shortcoming of these solutions is their lack of flexibility: the user has to stick to a limited set of templates without any opportunity to make changes or to add further features. For example, *Dealmate* (http://dealmate.io) is a template-based contract builder that enables users to create basic Ethereum smart contracts between two parties, for instance a buyer and a seller, with the contract acting as an escrow agent. Other examples are *TokenGen* (https://tokengen.io) and *BlockCAT* (https://blockcat.io), tools that make it possible for anyone to create an ERC20 token smart contract and to launch a token sale. When using these platforms, the user is expected to customize the contract only by filling in some input fields: the smart contract cannot be tailored to the actual requirements of the user.

As an alternative to template-based solutions, many researchers are investigating the possibility of adopting existing modelling languages, such as the *Business Process Model and Notation* (BPMN), to support the composition of smart contracts.

For example, Falazi et al. [25] propose a methodology to integrate blockchain-based operations within *Business Process Management Systems* (BPMSs), software solutions that support the creation of instances of BPMN process models, thus automatizing the execution of tasks and allowing process managers to monitor the state of processes. More specifically, they extend the standard BPMN in order to support read/write operations of blockchain transactions: their goal is to allow any existing BPMS solution compliant with BPMN 2.0 to interact with blockchain platforms during the execution of business processes. This is achieved with the *Blockchain-aware Modeling and Execution* (BlockME) method, which extends BPMN 2.0 with blockchain-aware constructs, enables the transformation of BlockME models into standard BPMN 2.0 models and finally introduces a middleware component called *Blockchain Access Layer* (BAL) which allows business process engines to access blockchain operations using asynchronous *Application Programming Interfaces* (APIs).

However, BPMN-based solutions use a notation and a methodology that was designed specifically for modeling business processes, and which may not be appealing to developer who are not familiar with that kind of approach.

VIII. Conclusion and Future Work

This paper contributes to the state of the art with two novelties in the blockchain landscape: an abstract description format for Ethereum smart contracts that provides all necessary information to search for and reuse smart contracts, and a visual programming environment for assisted development of smart contracts. The description format is equipped with a proof-of-concept implementation of a smart contract registry, i.e., SolidityRegistry. The programming environment, SolidityEditor, features a block-based modeling formalism equipped with constructs for basic Solidity instructions and more complex, aggregated patterns of instructions. The goal of these latter is to specifically foster reuse among smart contracts. The approach takes inspiration from web services (smart contracts) and service-oriented computing (registry and composition paradigm), and shows that smart contracts indeed present similar reuse opportunities as web services.

The case study implemented both manually and using SolidityEditor describes a first hands-on experience of the benefits of coding with blocks. Of course, the development of reliable smart contracts still requires intimate knowledge of the functioning mechanisms of the underlying blockchain platforms and specialized coding skills. It is not possible to completely eliminate the innate complexity of today's blockchain technology using a visual programming paradigm. Further abstractions and simplifications will be needed to make the technology accessible also to non-experts. Some of these simplifications may be achieved, for example, by refining the proposed visual programming paradigm; others will require simplifications of the underlying technology.

In our future work, we will further refine the proposed description format, so as to include also information about usage policies and pricing (which we omitted in this paper for simplicity). We also consider integrating SolidityRegistry with tools such as *Etherscan* (http://etherscan.io) to enable users to obtain more information about deployed contracts and their actual usage, and to publish the registry online. Then, we also plan to conduct proper user studies with SolidityEditor, in order to identify strengths and weaknesses and to improve its effectiveness.

Finally, as an additional contribution, both SolidityRegistry and SolidityEditor are open source and free for use.

References

[1] N. Satoshi, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[3] N. Szabo, "Smart contracts: Building blocks for digital markets," *EXTROPY: The Journal of Transhumanist Thought,(16)*, 1996.

[4] B. Dickson, "How blockchain can create the world's biggest supercomputer," *TechCrunch*, December 2016.

[5] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, "Web services," in *Web Services*. Springer, 2004, pp. 123–149.

[6] F. Daniel and L. Guida, "A service-oriented perspective on blockchain smart contracts," *IEEE Internet Computing, in print*, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8598947

[7] R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional, 2011.

[8] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (SoK)," in *Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[9] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," *arXiv preprint arXiv:1802.06038*, 2018.

[10] M. P. Singh and A. K. Chopra, "Violable contracts and governance for blockchain applications," *arXiv preprint arXiv:1801.02672*, 2018.

[11] V. Gatteschi, F. Lamberti, C. Demartini, C. Pranteda, and V. Santamaría, "Blockchain and smart contracts for insurance: Is the technology mature enough?" *Future Internet*, vol. 10, no. 2, 2018.

[12] Ethereum, "Ethereum Natural Specification Format," http://github.com/ethereum/wiki/wiki/Ethereum-Natural-Specification-Format.

[13] J. Rojas, "Suggestion of list of known Ethereum network ids," https://github.com/ethereumbook/ethereumbook/issues/110, 2018.

[14] V. Buterin, "Ethereum EIP 155," https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md, 2016.

[15] ENS, "Introduction to the Ethereum Name Service," http://ens.readthedocs.io/en/latest/introduction.html.

[16] EthPM, "Integrating your tool with the Ethereum Package Registry," https://www.ethpm.com/docs/integration-guide.

[17] Solidity, "Contracts," https://solidity.readthedocs.io/en/latest/contracts.html.

[18] M. Zupan, "Interactions between smart contracts with Solidity," https://zupzup.org/smart-contract-interaction/.

[19] B. A. Myers, "Taxonomies of visual programming and program visualization," *Journal of Visual Languages and Computing*, vol. 1, no. 1, pp. 97 – 123, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1045926X05800369

[20] S. Beydeda, M. Book, V. Gruhn *et al.*, *Model-driven software development*. Springer, 2005, vol. 15.

[21] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable programming: Blocks and beyond," *Communications of the ACM*, vol. 60, no. 6, pp. 72–80, 2017.

[22] Solidity, "Common Patterns," https://solidity.readthedocs.io/en/latest/common-patterns.html.

[23] Zeppelin, "OpenZeppelin," https://github.com/OpenZeppelin/openzeppelin-solidity.

[24] DigixGlobal, "Doxity - Documentation Generator for Solidity," https://github.com/DigixGlobal/doxity.

[25] G. Falazi, M. Hahn, U. Breitenbcher, and F. Leymann, "Modeling and Execution of Blockchain-aware Business Processes," *Proceedings of the 12th Advanced Summer School on Service Oriented Computing*, 2018.