Smart Contract Locator (SCL) and Smart Contract Description Language (SCDL)

Andrea Lamparelli[1], Ghareeb Falazi[2][0000-0002-8064-9293], Uwe Breitenbücher[2][0000-0002-8816-5541], Florian Daniel[1][0000-0003-3004-8702], and Frank Leymann[2][0000-0002-9123-259X]

[1] Politecnico di Milano, Dipartimento di Elettronica Informazione e Bioingegneria
Via Ponzio 34/5, 20133 Milano, Italy
andrea.lamparelli@mail.polimi.it, florian.daniel@polimi.it

[2] Institute for Architecture of Application Systems, University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart, Germany
{falazi, breitenbuecher, leymann}@iaas.uni-stuttgart.de

Abstract. Today's blockchain technologies focus mostly on isolated, proprietary technologies, yet there are application scenarios that ask for interoperability, e.g., among blockchains themselves or with external applications. This paper proposes the Smart Contract Locator (SCL) for the unambiguous identification of smart contracts over the Internet and across blockchains, and the Smart Contract Description Language (SCDL) for the abstract description of the external interface of smart contracts. The paper derives a unified metamodel for blockchain smart contract description and equips it with a concrete, JSON-based description language for smart contract search and discovery. The goal of the proposal is to foster smart contract reuse both inside blockchains and through the integration of smart contracts inside enterprise applications. The idea is inspired by the Service-Oriented Architecture (SOA) and aims to provide a high-level, cross-blockchain interoperability layer.

Keywords: Blockchain · Smart Contracts · Description · SCDL · SCL

1 Introduction

A *blockchain* is a distributed ledger, that is, a log of transactions that provides for their persistency and verifiability [13]. *Transactions* are cryptographically signed instructions constructed by a user of the blockchain [15] and directed toward other parties in the blockchain network, for example the transfer of cryptocurrency from one account to another. A transaction typically contains a predefined set of metadata and an optional payload. Transactions are grouped into so-called *blocks*; blocks are concatenated chronologically. A new block is added to the blockchain using a hash computed over the last block as a connection link. A *consensus protocol* enables the nodes of the blockchain network to create trust in the state of the log and makes blockchains inherently resistant to tampering [9]. *Smart contracts* [14] extend a blockchain's functionality from storing

transactions to performing also computations, for example, to decide whether to release a given amount of cryptocurrency upon the satisfaction of a condition agreed on by multiple partners.

Blockchains can be broadly categorized into *permissionless* and *permissioned*. Early blockchain platforms, such as Bitcoin [13] and Ethereum [15], were permissionless in the sense that participating in the protocol with any role is open for everyone. These platforms favor absolute decentralization at the cost of having relatively weak privacy and performance capabilities. Therefore, permissioned blockchains, such as Hyperledger Fabric [1], Hyperledger Sawtooth [10] and Corda [3], were introduced as an alternative that guarantees data confidentiality and ensures better performance. However, these desirable properties come with the price of losing some degree of decentralization, since joining the network becomes restricted and under the control of a single entity.

Both kinds of blockchains have their use-cases that can sometimes coincide. For example, in a scenario that involves a consortium of enterprises partially trusting each other, one or more permissioned blockchain networks can be used to guarantee to all participants that the collaborative process itself is being conducted exactly as designed, while ensuring good performance and privacy. However, to provide a similar guarantee to external entities that do not trust the consortium as a whole, such as auditing authorities, it is not enough to use permissioned blockchains, since they favor privacy over transparency and cannot prove that some transactions were not removed from the ledger history due to a malicious agreement between the consortium members. In that case, the additional involvement of permissionless blockchains can provide the desired guarantees. Therefore, we see that there is no single blockchain technology that is capable of solving all potential use-cases, which means that existing and new variations of blockchains would continue to co-exist, and end-users would likely become involved in a mixture of them in relatively complex scenarios [6].

To integrate blockchains into existing processes, using, e.g., business process management systems [5,6], their smart contracts need to be used, since, from an external viewpoint, the public functions of smart contracts are the access-points at which blockchains can be utilized by other systems, i.e., they are the *integration points* of blockchains. However, as mentioned earlier, multiple permissioned and permissionless blockchain platforms might need to be integrated in the same use-case. The problem here is that smart contracts of different blockchains are invoked using different mechanisms, protocols, and data formats, which significantly raises the integration barrier for systems wishing to utilize them, since developers need to be aware of these variations making the integration process time-consuming and error-prone. Furthermore, the specific smart contracts relevant for a given use-case need to be identified, which is not a straightforward task, because information regarding existing smart contracts of various blockchains is not uniformly available for developers.

In this paper, we extend our previous approach [8], which introduced an Ethereum-specific smart contract description format, to a wider set of blockchain technologies. Here, we propose a Service-Oriented Architecture (SOA)-inspired

style of integration: We first analyze state-of-the-art blockchain platforms and derive cross-blockchain addressing and description requirements (Section 2). Then, we introduce a smart contract addressing format, the *Smart Contract Locator* (SCL), as a specialization of the generic URL scheme that facilitates the unambiguous identification of smart contract functions, both externally over the Internet and internally from within the blockchain network (Section 3). Then, we define a unified metamodel capable of describing the public interface of smart contracts of multiple permissioned and pemissionless blockchains; finally, we equip the metamodel with a JSON-based language called the *Smart Contract Description Language* (SCDL) for uniform smart contract descriptors that can be stored in a specialized registry to provide the functionality of smart contract search and discovery (Section 4). We close the paper with related works in Section 5 and a discussion of our proposal and future works in Section 6.

2 Analysis of Smart Contracts

In [4], we analyzed contract types, interaction styles, interaction protocols, data formats and blockchain-internal description formats of smart contracts, and demonstrated the suitability of smart contracts for the implementation of a smart contract-based, service-oriented architecture. Next, we study the specifics of smart contract interfaces for contract description.

2.1 Fundamentals of smart contracts

Most blockchain platforms today support different *programming languages* for the implementation of smart contracts, ranging from general-purpose languages like Java, C++, Python, JavaScript, Golang to platform-specific languages like Solidity for Ethereum or Bitcoin Script for Bitcoin [4]. Most of these languages are object-oriented and, hence, a smart contract can be seen as an object that has an identity, a behavior, a state, and events. Typically, smart contracts are executed using a blockchain-specific *virtual machine* that replicates the same "computer" on all nodes of the blockchain network. The most famous and used virtual machine today is the Ethereum Virtual Machine ("EVM", https://py-evm.readthedocs.io) developed by Ethereum and used by several other platforms for smart contract execution. For its execution, a smart contract must be *deployed* on the blockchain and *instantiated* in the virtual machine. This process creates an instance of the contract – along with a unique contract identifier – and initializes its state. After this initialization, the contract becomes accessible to possible clients who can invoke the contract according to its *external interface* (the functions made available) by submitting suitable *transactions* that carry the invocation in their body. Invocations may come from other smart contracts inside the same blockchain or from the outside, e.g., from enterprise applications. How exactly contracts are invoked is, again, platform dependent.

Bringing together the different models of smart contracts that have emerged so far, the most important characteristics can be summarized as follows (we analyze concrete technologies in the next subsection):

- ***Identity***: This is typically defined by a specific address that corresponds to the deployment location of the contract. Each platform has its own way to compute this address. In some blockchains contracts are treated like any other account, and the address is an *account identifier*; in other platforms they are considered immutable states (variables) identified by a virtual *memory address*. The address does not only distinguish different contracts from each other, but also different, independent instances of a same contract.
- ***State***: This refers to the properties (variables) internal to the contract that are *persistent* across multiple invocations. A contract can be *immutable*, where the state cannot be changed after its initialization, or *mutable*, where the state can be modified during the contract's life. Immutable contracts are typically used as transaction validators that check conditions only; mutable contracts can implement any kind of business logic.
- ***Functions***: These implement the operations a contract can perform and, thus, its behavior. A function usually has a *scope* that tells the visibility of the function (e.g., private vs. public or blockchain-internal vs. -external), a *name*, a number of *input* parameters, and optional *return* parameters. A function is called "pure" if return values depend only on input values and it does not produce any side effects on the state; it is called "view" function if it provides read-only access to state. Some blockchain platforms allow the *direct* invocation of functions using their name, others advocate the use of a single *dispatcher* function to forward input values to target functions.
- ***Events***: An event occurs when a contract sends a signal that an action or *state change* has taken place upon its invocation. Events allow external applications to monitor the state of the contract, while the blockchain platform allows applications to *subscribe* to or *unsubscribe* from events. Events usually have a *name* and a set of *parameters* that represent the payload of the event. Some platforms generate *system events*, others support developer-defined *custom events*. Custom events may require an explicit declaration of the event and its parameters (the event prototype) and can be launched programmatically; system events are launched automatically. Depending on the platforms *single* or *multiple* events may be launched at a time.
- ***Description***: For developers to understand the exact model of a given smart contract, since smart contracts are deployed on the blockchain, the developer could inspect the deployed code, but such is typically a compiled version and, hence, not useful to derive how to interact with it. Some platforms in addition generate descriptive *metadata* at compilation time that may provide both the actual source code and an abstract summary of the external interface of the contract, often called *Application Binary Interface* (ABI).

Ideally, for a given smart contract, all these aspects are specified in a proper descriptor and made accessible online (e.g., Ethereum proposes Swarm, https://ethersphere.github.io/swarm-home/, to host such metadata), yet as of today, there is no commonly used *registry* for storing and indexing metadata or descriptors for smart contracts of various blockchain platforms, let alone a uniform description language.

2.2 Comparison of blockchain platforms

In order to understand the state of the art of smart contract support by blockchain platforms, we have selected platforms for comparison from the two major blockchain families, permissionless and permissioned. As mentioned earlier, permissionless blockchains allow anyone to participate and access information stored in the network, whereas permissioned blockchains allow only invited nodes to participate and access data. The selected platforms are:

- *Bitcoin* (https://bitcoin.org), the first permissionless blockchain platform introduced with limited support for smart contracts. Contracts are used as validators, have an immutable state and are used to lock/unlock values only.
- *Ethereum* (https://www.ethereum.org), the permissionless platform that first introduced Turing-complete smart contracts that, in principle, allow the implementation of arbitrary application logic.
- *Hyperledger Fabric* (https://www.hyperledger.org/projects/fabric), a permissioned blockchain platform developed by The Linux Foundation that leverages on container technology to host smart contracts called "chaincode".
- *Neo* (https://neo.org), also known as the "Ethereum of China," with support for multiple digital assets and smart contracts; Neo is permissionless.
- *EOSIO* (https://eos.io), a more recent permissioned/permissionless platform with a special focus on transaction throughput for businesses.
- *Hyperledger Sawtooth* (https://sawtooth.hyperledger.org), another permissioned blockchain platform from the The Linux Foundation that is highly modular and configurable. It introduces *transaction families*, which are pluggable, user-defined components, as the way to define smart contracts.

Moreover, Ethereum is the "father" of many other blockchain platforms, such as Qtum (https://qtum.org), Ubiq (https://ubiqsmart.com), Rootstock (https://www.rsk.co) and others. We omit them from the comparison, as they all comply with Ethereum's smart contract model and use the EVM.

Table 1 summarizes how the chosen platforms implement smart contracts. For the comparison, we use the smart contract characteristics described earlier; we do not consider aspects like access policy, consensus protocol, performance or similar, as these do not affect smart contracts' external interfaces. The analysis aims to provide a picture that abstracts away from implementation languages and instead emphasizes the addressing and functional interface perspective.

Addressing: Looking at how smart contracts are identified (first dimension), it is evident how contracts are referenced differently across different platforms. While there may be platform-specific reasons for this (e.g., Bitcoin does not have the concept of accounts), conceptually – from an external point of view – it must be possible to do so in an abstract, uniform manner.

Interface: State, if not immutable, is manipulated through functions, which are only visible to consumers if they are public; Ethereum and EOSIO further distinguish between functions that are internal to the blockchain (invocable only by contracts of the same blockchain) and functions that are external (invocable also by agents outside the blockchain). Most of the platforms support launching

Table 1. Comparison of smart contract support by most representative blockchain platforms from an external perspective.

Platform	Identity	State	Functions	Events	Description
Bitcoin	contracts specify how *unspent transaction outputs* (UTXO) can be used; identified by UTXO address	set when instantiating contract; immutable	public; can be invoked directly	–	–
Ethereum	contracts implement generic application logic; have own accounts	stored in contracts; modified using functions	public/private and blockchain-internal/-external; invoked directly	multiple custom events possible; explicit declaration of event prototype	contract meta-data and *Application Binary Interface* (ABI)
Hyper-ledger Fabric	contracts (*chaincode*) implement generic application logic and are addressed using an ID	stored in contracts; modified using functions	public/private; invoked using dispatcher function	max one custom event per invocation; no explicit event prototype needed	*Chaincode Interface* (CCI) for language-neutral description
Neo	contracts implement generic application logic; have own accounts	stored in contracts; modified using functions	public/private; invoked either directly or via a dispatcher (recommended)	multiple custom events possible; no declaration needed	contract metadata and *Neo ABI*
EOSIO	generic; hosted by EOSIO accounts (1-to-1 relationship) and identified by human-readable unique string	stored in the contract, modified using functions (*actions*)	public/private and blockchain-internal/-external; invoked using dispatcher func.	multiple system events possible; no custom events	contract metadata and *EOSIO ABI*
Hyper-ledge Sawtooth	contracts (*transaction families*) implement generic application logic; addressed using a 35-byte hex hash of transaction family name	stored in transaction families, modified using functions	public; invoked only from external apps via a REST call to custom transaction family processor	multiple custom events possible; explicit declaration of event prototype	public interface of a transaction family defined by the developer via a set of *protobuf** message types

* https://developers.google.com/protocol-buffers/

custom events to communicate with external agents; only Bitcoin and EOSIO support either no events or only system events. From a description point of view, it is interesting to note that most platforms are able to generate some descriptive metadata at compile time, along with an ABI that provides a summary of function prototypes – both however providing different kinds of information and focusing on blockchain-specific aspects. Yet, as the table also shows, there are significant similarities across platforms, which hints at the possibility to abstract external interfaces and uniformly describe them for uniform access.

3 Smart Contract Locator (SCL)

Internally, all platforms provide for smart contract addressing or identification; so, there is no need for intervention. Instead, Figure 1 (solid, black components on the top) illustrates our minimal, architectural assumptions for the specification of the Smart Contract Locator (SCL), which is our proposal for uniformly addressing smart contracts from the outside of their blockchains: an *external*

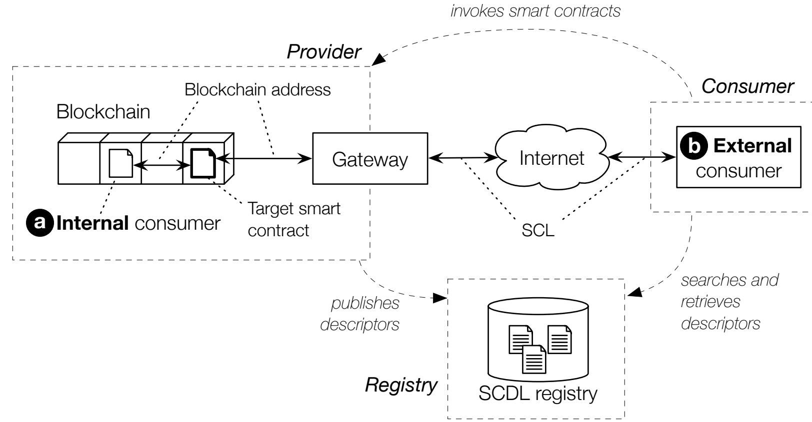

Fig. 1. Conceptual components for smart contract addressing (solid lines) for (a) blockchain-internal consumers and (b) blockchain-external consumers with service-oriented architecture for smart contracts (thin, dashed lines).

consumer (e.g., an enterprise application) that wants to invoke a *target smart contract* (e.g., a currency exchange app) deployed inside a *blockchain network* (e.g., Ethereum) to which it does not have own access (it does not own any node of the network) may have to cross the Internet to reach a so-called *gateway*, a web-accessible agent that is able to mediate between the external consumer and the target smart contract. SCL tells the external consumer how to reach that gateway and how to identify the target smart contract.

We intentionally limit the use of SCL to smart contract addressing only; the identification of the functions to be invoked and the passing of suitable parameter values will be done using the payload of the messages exchanged between consumer and smart contract (e.g., using http POST messages). We assume that the communication channel from the external consumer to the gateway is properly secured using state-of-the-art security mechanisms like https, access control, and encryption.

Now, given the IETF specification of the generic URL format [2]:

```
URL = scheme:[userinfo@]host[:port]path[?query][#fragment]
```

and the preliminary proposal for smart contract addressing in [6] (see Section 5), we define an SCL as a specialization of a URL composed of a standard URL (up to the `path` element included), which identifies the gateway, and of an SCL query, which identifies the target smart contract inside the blockchain network:

```
SCL = scheme:[userinfo@]host[:port]path"?"scl_query
scl_query = "blockchain="bc"&blockchain-id="id"&address="addr

bc = "ethereum" | "bitcoin" | "fabric" | "eosio" | ...

id = NetworkIdentifier                    // not further detailed here
```

```
addr = eth_addr | bit_addr | fab_addr | eos_addr | ...
eth_addr = 40ByteHexString              // not further detailed here
bit_addr = Bech32Address                // not further detailed here
fab_addr = PathString                   // not further detailed here
eos_addr = 12CharacterString            // not further detailed here
```

The SCL extension of URLs thus specifies (i) which type of blockchain is addressed, (ii) which exact blockchain network (there may be more networks accessible through a given gateway), and (iii) the blockchain-internal smart contract address or identifier.

In the following, we list example SCL addresses for a set of the supported blockchains that are accessed using the `https` scheme via a hypothetical gateway hosted at `mygateway.com`:

```
* Ethereum:
https://mygateway.com?blockchain=ethereum&blockchain-id=eth-mainnet
    &address=0xa0b73e1ff0b80914ab6fe0444e65848c4c34450b
* Bitcoin:
https://mygateway.com?blockchain=bitcoin&blockchain-id=btc-mainnet
    &address=1Mbk53DzVKCz6MHiBd8ZHkPhsZETo7PtZR
* Hyperledger Fabric:
https://mygateway.com?blockchain=fabric&blockchain-id=part-vendors
    &address=channel1%2Fchaincode1%2Fsmartcontract1
* EOSIO:
https://mygateway.com?blockchain=eos&blockchain-id=eos-mainnet
    &address=myfancyacc05
```

4 Smart Contract Description Language (SCDL)

Looking at the dashed annotations in Figure 1, we can identify the typical roles of the service-oriented architecture (SOA): a provider, a consumer and a registry [11]. We assume that:

– The ***consumer*** is represented either by a blockchain-internal entity (a smart contract) or a blockchain-external entity (a software application) – both of them interested in reusing a given target smart contract, e.g., to inherit application logic or to integrate blockchain capabilities into enterprise applications. In order to do so, it is crucial that developers be able to find suitable smart contract descriptions that tell them all they need to know in order to invoke the contract from the inside/outside.
– The ***provider*** is represented by the operator of the blockchain, who is interested in opening its smart contracts to external entities. The practice is commonly known as Blockchain-as-a-Service (BCaaS [12]) and is pushed by vendors like Amazon (https://aws.amazon.com/managed-blockchain), Upvest (https://upvest.co) or Kaleido (https://kaleido.io). In order to allow external consumers to connect to a hosted blockchain, the provider publishes suitable descriptors and a gateway.

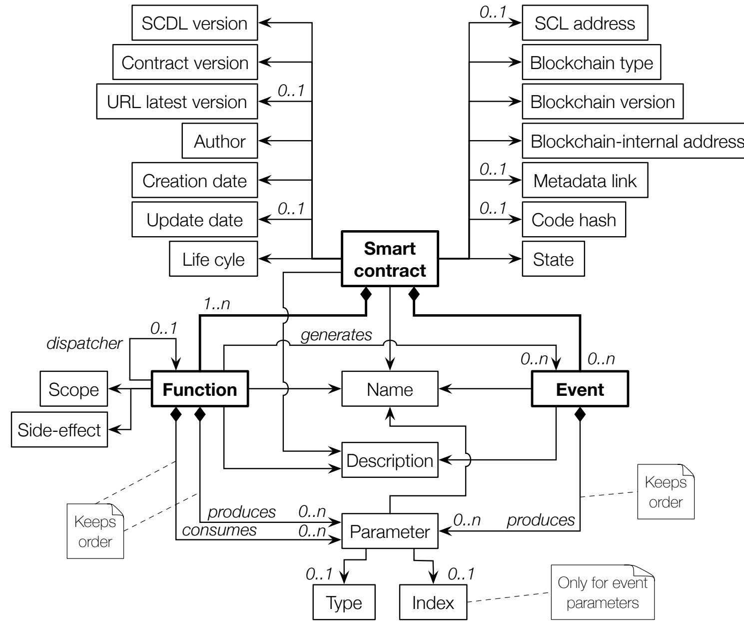

Fig. 2. Metamodel of Smart Contract Description Language (SCDL), version 1.0.

– The ***registry*** hosts smart contract descriptors and provides consumers with search and retrieval capabilities. The design of this registry is out of the scope of this paper and part of our future work.

The goal of the Smart Contract Description Language (SCDL) is now to enable the *abstract, blockchain-independent description of the external interfaces of smart contracts* and to cater to both *internal and external consumers*. The language should further provide for the *extensibility* to allow developers to include blockchain-, contract- or application-specific metadata if needed.

4.1 Language metamodel

Given these requirements and the results of Section 2, which analyzed state-of-the-art support for smart contracts, Figure 2 illustrates the metamodel of SCDL; furthermore, the left half of Table 2 explains each of the entities in the metamodel. According to the metamodel, a smart contract can be seen as a blockchain- or web-accessible entity that is characterized by a set of descriptive metadata elements, a set of functions and a set of events.

Typical *metadata* are generic attributes like contract name, description, author and version, but also access-oriented attributes like the SCL address for external consumers and the blockchain type, version and internal address for internal consumers. Where available (e.g., for Ethereum smart contracts) publicly accessible metadata can be linked and a hash of the contract's code can

be added to allow developers to check if a descriptor is up to date. *Functions*, too, have a name and a description and are characterized by the set of input and return parameters they consume/produce; parameter lists are ordered (the order is needed for some platforms to be able to properly invoke functions). Functions may further produce events, e.g., for the implementation of asynchronous communication with consumers, have a scope (e.g., public vs. private), produce or not side-effects (change or not the state), and specify a dispatcher function for those platforms where functions are not invoked directly (e.g., Hyperledger Fabric). *Events* have a name, a description and an ordered list of output parameters. *Parameters* have a name, an abstract data type (external consumers) that allows the derivation of a blockchain-specific, native data type (internal consumers) and may be indexed to enable consumers to query events on the blockchain.

The metamodel does not explicitly provide any extensibility points. It represents the minimum set of properties that allow a provider to describe any of the smart contracts studied in Section 2. For Bitcoin scripts, we can interpret clauses as functions and describe how to trigger them by means of the parameters needed to make them true. If additional properties are needed, these can simply be added as properties to the composite objects of the language, i.e., smart contract, function, event, parameter. For instance, if a provider wants to explicitly mention the programming language of a given smart contract, this could be achieved by adding a language property to the smart contract object.

For simplicity, in this paper we assume that there exists a suitable agreement between the provider and the consumer regarding the *costs* the provider may incur when executing smart contracts on behalf of the external consumer (internal consumers are charged directly by the blockchain platform).

4.2 SCDL JSON syntax

We propose to equip the metamodel with a concrete syntax based on JSON, which is supported by multiple blockchain platforms (e.g., Ethereum, Hyperledger Fabric) and, hence, maintains consistency with existing conventions.

The translation of the metamodel to a concrete syntax follows few simple rules: *entities with associated properties* produce JSON objects with properties; *composition relationships* are translated to JSON arrays; the *order* of parameters of functions or events is expressed by their order inside their respective arrays; *abstract data types* of parameters are expressed using JSON Schema (https://json-schema.org)[3]. The right half of Table 2 defines each individual language construct in detail and equips it with a respective domain of possible values. The general structure of a SCDL descriptor is organized as shown in Figure 3.

Next to JSON, also formats like XML, YAML or similar are compatible with the metamodel. We propose the use of JSON Schema to express abstract data types in order to enable external consumers (e.g., a business process engine connected to a blockchain via a gateway) to understand basic data types without the need for blockchain-specific knowledge.

[3] For mappings see https://github.com/floriandanielit/scdl#data-encoding.

Table 2. SCDL 1.0 constructs with concrete syntax and domains of values. Mandatory elements are the minimum information needed to uniquely characterize smart contracts.

Construct	Description	Syntax element	Type of $value
Smart contract			
✘ SCDL version	Version of SCDL used; in this paper the version is 1.0	`scdl_version : $value`	`String`
✘ Name	Expressive name of contract	`name : $value`	`String`
✘ Contract version	Version of smart contract	`version : $value`	`String`
☐ URL latest version	Optional URL to latest descriptor	`latest_URL: $value`	`URL`
☐ Description	Free text description of contract	`description : $value`	`String`
✘ Author	Developer name of contract	`author : $value`	`String`
✘ Creation date	Date the descriptor was created	`created_on : $value`	`Date`
☐ Update date	Date the descriptor was updated	`updated_on : $value`	`Date`
☐ Lifecycle	Lifecycle state of contract	`lifecycle : $value`	`"ready" | "destroyed"`
☐ SCL address	SCL address of possible smart contract gateway	`scl : $value`	`SCL address, see Section 3`
✘ Blockchain type	Name of the blockchain platform	`blockchain_type : $value`	`"ethereum" | "bitcoin" | "fabric" | "neo"`
✘ Blockchain version	Version of the blockchain platform the contract runs on	`blockchain_version : $value`	`String`
✘ Blockchain-internal address	Blockchain-internal address of smart contract, e.g., Ethereum accout	`internal_address : $value`	`String`
☐ Metadata link	Link to external metadata of contract, if available	`metadata : $value`	`URL`
☐ Code hash	SHA256 hash of contract code	`hash : $value`	`String`
✘ State	Tells if the contract maintains internal state of not	`is_stateful : $value`	`Boolean`
✘ Functions	List of functions provided by contract	`functions : [Function, Function,...]`	`Array of Function`
☐ Events	List of events generated by contract	`events : [Event, Event,...]`	`Array of Event`
Function			
✘ Name	Contract-wide unique name of function	`name : $value`	`String`
☐ Description	Free text description of function	`description : $value`	`String`
✘ Scope	Visibility of function	`scope : $value`	`"public" | "private" | "internal" | "external"`
✘ Side-effect	Tells whether the function as side-effects on state or not	`has_side_effects : $value`	`Boolean`
☐ Dispatcher	Name of dispatcher function to use for function invocation	`dispatcher : $value`	`String`
✘ Input parameters	List of input parameters	`inputs : [Parameter, Parameter,...]`	`Array of Parameter`
✘ Output parameters	List of output parameters	`outputs : [Parameter, Parameter,...]`	`Array of Parameter`
☐ Events generated	List of names of events generated by function	`events : [String, String,...]`	`Array of String`
Event			
✘ Name	Contract-wide unique event name	`name : $value`	`String`
☐ Description	Free text description of event	`description : $value`	`String`
✘ Output parameters	List of output parameters	`outputs : [Parameter, Parameter,...]`	`Array of Parameter`
Parameter			
✘ Name	Unique parameter name	`name : $value`	`String`
✘ Type	Abstract, blockchain-indepdendent data type of parameter	`type : $value`	`String`
☐ Index	Tells if parameter is indexed and therefore searchable	`is_indexed : $value`	`Boolean`

✘ Mandatory element ☐ Optional element

```
{"scdl_version" : "1.0.0",          // generic smart contract properties
 "name" : "TokenConversion", ...
 "functions" : [
   { "name" : "convert", ...                        // function properties
     "inputs" : [
       { "name"   : "amount",
         "type"   : "number"
       }, ...                                        // list of parameters
     ],
     "outputs" : [...],                              // list of parameters
     "events"  : [...]                               // list of parameters
   }, ...                                            //  list of functions
 ],
 "events" : [
   { "name"    : "...", ...                          //   event properties
     "outputs" : [...],                              // list of parameters
   }, ...                                            //     list of events
 ]
}
```

Fig. 3. General structure of SCDL descriptor

4.3 Example: ZilliqaToken contract

As an example, let's consider the ZilliqaToken contract deployed on Ethereum by the Zilliqa Team; the deployed contract and its code can be inspected at https://bit.ly/2GBajXC. The contract follows the ERC20 standard (https://theethereum.wiki/w/index.php/ERC20_Token_Standard) for the implementation of the ZIL token in Ethereum. The contract allows its users to check their token balance, transfer tokens among accounts, approve others to spend tokens, etc.

Figure 4 provides an excerpt from a possible SCDL descriptor of the contract (core metadata, one function and one correlated event). Next to the name and a short description, the descriptor provides the external consumer with the `SCL` address of the contract and the internal consumers with the `internal_address`. As we chose the latest version of the contract, there is no link to any newer version of the contract, and the source code is linked using the `metadata` link. The contract is stateful, as it tracks token balances. The function `transfer` allows the user to transfer a given `_value` to a receiver `_to`. The function can be invoked directly using its name and generates the event `Transfer` with parameters `from`, `to`, `value` upon completion of the transfer. The parameters `from` and `to` are indexed and can thus be used for fast search of token transfers among accounts. The description of the complete contract is linked in the caption of Figure 4.

5 Related Work

The problem of describing the external interface of software components is not new and has gained particular attention with the advent of the service-oriented architecture. Two core service models have emerged: SOAP web services [11] and RESTful APIs [7], the former equipped with description lan-

```
{ "scdl_version" : "1.0",
  "name" : "ZilliqaToken",
  "version" : "^0.4.18",
  "latest_url" : null,
  "author" : "0xBfE4aA5c37D223EEBe0A1F7111556Ae49bE0dcD2",
  "description" : "Contract token implementation following the ERC20 standard, the new created
   token is called ZIL",
  "created_on" : "Jan-12-2018 09:44:42 AM +UTC",
  "updated_on" : "Jan-12-2018 09:44:42 AM +UTC",
  “scl" : "https://mygateway.com?blockchain=ethereum&blockchain-id=eth-
          mainnet&address=0x05f4a42e251f2d52b8ed15E9FEdAacFcEF1FAD27",
  "internal_address" : "0x05f4a42e251f2d52b8ed15E9FEdAacFcEF1FAD27",
  "blockchain_type" : "ethereum",
  "blockchain_version" : "v0.4.18+commit.9cf6e910",
  "metadata" : "https://etherscan.io/address/0x05f4a42e251f2d52b8ed15e9fedaacfcef1fad27#code",
  "hash" : "b311edaec5a164050cede3219bf28cc6ce4c0ca43b8bf34d6fd309fb60c4d1d8  -",
  "is_stateful" : true,
  "lifecycle" : "ready",
  "functions" : [
      { "name" : "transfer",
        "description" : "* @dev transfer
         token for a specified address.
         @param _to The address to transfer
         to. @param _value Amount to be transf."
        "scope" : "public",
        "has_side_effects" : true,
        "inputs" : [
           {    "name" : "_to",
                "type" : "string"
                "pattern" : "^0x[a-fA-F0-9]{40}$"

           {    "name" : "_value",
                "type" : "number"
                "minimum" : "0"
                "maximum" : "2^256-1"
           }
        ],
        "outputs" : [
           {    "name" : null,
                "type" : "boolean"
           }
        ],
        "events" : ["Transfer"],
        "dispatcher" : null
      }, ...
  ],
```


continues


```
  "events" : [
      {    "name" : "Transfer",
           "description" : "Triggered when
            tokens are transferred",
           "outputs" : [
               {    "name" : "from",
                    "type" : "string",
                     "pattern" :
                        "^0x[a-fA-F0-9]{40}$"
                    "is_indexed" : true
               },
               {    "name" : "to",
                    "type" : “string”,
                     "pattern" :
                        "^0x[a-fA-F0-9]{40}$"
                    "is_indexed" : true
               },
               {    "name" : "value",
                     "type" : "number"
                     "minimum" : "0"
                     "maximum" : "2^256-1"
                    "is_indexed" : false
               }
           ]
      }, ...
  ]
```


Fig. 4. JSON-based SCDL descriptor of ZilliqaToken smart contract with hypothetical SCL addresss. For brevity, we report here only one function and one connected event; the full descriptor can be inspected online via https://bit.ly/2LRy9Tb.

guages like WSDL (https://www.w3.org/TR/2007/REC-wsdl20-20070626) and WSDL-S (https://www.w3.org/Submission/WSDL-S), the latter with languages like WADL (https://www.w3.org/Submission/wadl) and Swagger / OpenAPI (https://swagger.io). WADL and Swagger / OpenAPI are oriented toward stateless resources and are, hence, out of scope. The metamodels of WSDL and WSDL-S are generic, that of SCDL is smart contract specific (e.g., it expresses relationships between functions and events and identifies indexed parameters).

The first approach to describing smart contracts in a blockchain-familiar fashion (JSON) is introduced in [8], where we suggested a SOA-based approach that allows one to uniformly describe Ethereum smart contracts and to store the resulting descriptions in a specialized registry that facilitates reuse. Compared to that work, the SCDL we propose here goes beyond Ethereum to a wider set of permissioned and permissionless blockchains. Furthermore, we also target developers of external applications by differentiating between internal,

blockchain-specific smart contract addresses, and external, uniform addresses, i.e., SCLs, which can be used over the Internet.

In previous work [6], we instead focused on the process-based composition of heterogeneous smart contracts. The approach uses an extension of BPMN that allows invocations to permissioned and permissionless smart contract functions from standard business processes that can be executed by regular process engines. To allow for technology-agnostic models, the process engine utilizes an extensible middleware component called Blockchain Access Layer (BAL), which translates the calls it receives from external applications, e.g., the process engine, into blockchain-specific invocations. To identify the smart contract function that needs to be invoked, the BAL used a non URL-compatible URI scheme.

The SCL addressing scheme presented in this paper allows external applications to address heterogeneous smart contract functions across the Internet by utilizing the concept of a gateway that provides access to one or more blockchain platforms. This decouples the external consumers from the middleware that facilitates the communication with blockchain platforms.

6 Discussion and Outlook

This paper advances the state of the art in blockchain technology with two proposals of abstraction, i.e., the Smart Contract Locator (SCL) for cross-blockchain addressing of smart contracts and the Smart Contract Description Language (SCDL) for the abstract description of smart contracts. We consider both as founding ingredients for the development of a service-oriented architecture that is based on smart contracts and enables a service-like integration of blockchains into generic software applications. Commercial Blockchain-as-a-Service providers like Amazon, Upvest and Kaleido are evidence that the market is ready, yet this paper claims that suitable abstractions and middleware support are still missing. In this respect, SCL and SCDL do not just want to advance that state of the art but they also want to stimulate the discussion.

The proposal of SCL is compliant with standard URLs, which makes it natively ready for the Internet. The examples in this paper use a scheme binding of `"http"` or `"https"`, but nothing prohibits the use of SMTP or any other transport protocol. Similarly, SCDL is proposed with a JSON binding for serialization. This choice was driven by the observation that most blockchain platforms analyzed already make large use of JSON, e.g., for the invocation of functions, and hence aims to keep consistency. However, given the metadmodel of SCDL, alternative bindings can be defined for XML, YAML, WSDL or others.

The next step of our work will concentrate on the specification of a smart contract invocation protocol to rule the communication between external consumers and gateways, as well as on the implementation of a reference architecture for gateways able to provide access to different blockchain technologies. In terms of SCDL, the next version of the language will provide for the description of non-functional aspects like service-level agreements and payments – one feature where smart contracts excel compared to SOAP/REST services. SCDL will also

be equipped with a suitable, open registry able to host descriptors and to provide for search and retrieval of smart contracts.

We intend to use GitHub to evolve the proposals of SCL (https://github.com/ghareeb-falazi/scl) and SCDL (https://github.com/floriandanielit/scdl) with help from the community.

Acknowledgements. This work was supported by the European Union's Horizon 2020 research and innovation programme, project DITAS, grant agreement RIA 731945.

References

1. Androulaki, E., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Barger, A., Cocco, S.W., Yellick, J., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G.: Hyperledger Fabric. In: EuroSys 2018. pp. 1–15. ACM Press (2018). https://doi.org/10.1145/3190508.3190538
2. Berners-Lee, T., Masinter, L., McCahill, M.: Uniform Resource Locators (URL) (1994), https://www.ietf.org/rfc/rfc1738.txt
3. Brown, R.G.: The Corda Platform: An Introduction. Corda Platform Whitepaper pp. 1–21 (2018), https://www.corda.net/content/corda-platform-whitepaper.pdf
4. Daniel, F., Guida, L.: A Service-Oriented Perspective on Blockchain Smart Contracts. IEEE Internet Computing **23**(1), 46–53 (Jan 2019)
5. Falazi, G., Hahn, M., Breitenbücher, U., Leymann, F.: Modeling and execution of blockchain-aware business processes. SICS Software-Intensive Cyber-Physical Systems **34**(2), 105–116 (Jun 2019). https://doi.org/10.1007/s00450-019-00399-5
6. Falazi, G., Hahn, M., Breitenbücher, U., Leymann, F.: Process-Based Composition of Permissioned and Permissionless Blockchain Smart Contracts. In: EDOC 2019 (to appear)
7. Fielding, R.: Representational state transfer. Architectural Styles and the Design of Network-based Software Architecture pp. 76–85 (2000)
8. Guida, L., Daniel, F.: Supporting Reuse of Smart Contracts through Service Orientation and Assisted Development. In: IEEE DappCon 2019. pp. 59–68 (2019)
9. Mingxiao, D., Xiaofeng, M., Zhe, Z., Xiangwei, W., Qijun, C.: A review on consensus algorithm of blockchain. In: SMC 2017. pp. 2567–2572. IEEE (2017)
10. Olson, K., Bowman, M., Mitchell, J., Amundson, S., Middleton, D., Montgomery, C.: Sawtooth: An introduction. Hyperledger Sawtooth Whitepaper pp. 1–7 (2018), https://www.hyperledger.org/wp-content/uploads/2018/01/Hyperledger_Sawtooth_WhitePaper.pdf
11. Papazoglou, M.P., Georgakopoulos, D.: Service-oriented computing. Communications of the ACM **46**(10), 25–28 (2003)
12. Samaniego, M., Deters, R.: Blockchain as a service for iot. In: 2016 IEEE iThings / GreenCom / CPSCom / SmartData. pp. 433–436. IEEE (2016)
13. Satoshi, N.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008), https://bitcoin.org/bitcoin.pdf
14. Szabo, N.: Smart contracts: building blocks for digital markets. EXTROPY: The Journal of Transhumanist Thought,(16) (1996)
15. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)