HAL
open science

SmartAnvil: Open-Source Tool Suite for Smart Contract Analysis

Stéphane Ducasse, Henrique Rocha, Santiago Bragagnolo, Marcus Denker, Clément Francomme

To cite this version:

Stéphane Ducasse, Henrique Rocha, Santiago Bragagnolo, Marcus Denker, Clément Francomme. SmartAnvil: Open-Source Tool Suite for Smart Contract Analysis. Blockchain and Web 3.0: Social, economic, and technological challenges, Routledge, 2019. hal-01940287

HAL Id: hal-01940287

https://inria.hal.science/hal-01940287v1

Submitted on 30 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

1

SmartAnvil: Open-Source Tool Suite for Smart Contract Analysis

Stéphane Ducasse[1]
ORCID.org/0000-0001-6070-6599
Henrique Rocha
ORCID.org/0000-0002-9154-0277
Santiago Bragagnolo
ORCID.org/0000-0002-5863-2698
Marcus Denker
ORCID.org/0000-0003-2549-4222
Clément Francomme
ORCID.org/0000-0003-1269-0275

Abstract

Smart contracts are new computational units with special properties: they act as classes with aspectual concerns; their memory structure is more complex than mere objects; they are obscure in the sense that once deployed it is difficult to access their internal state; they reside in an append-only chain. There is a need to support the building of new generation tools to help developers. Such support should tackle several important aspects: (1) the static structure of the contract, (2) the object nature of published contracts, and (3) the overall data chain composed of blocks and transactions. In this chapter, we present SmartAnvil an open platform to build software analysis tools around smart contracts. We illustrate the general components and we focus on three important aspects: support for static analysis of Solidity smart contracts, deployed smart contract binary analysis through inspection, and blockchain

[1]Accepted to appear in "Blockchain and Web 3.0: Social, economic, and technological challenges"

navigation and querying. SmartAnvil is open-source and supports a bridge to the Moose data and software analysis platform.

1.1 Relevance: the need to support tool building

Solidity's smart contracts are new computational units with special properties [Eth18c]: they act as classes with aspectual concerns; they respond to message calls as a remote object; their memory structure is more complex than that of mere objects; they are obscure in the sense that once deployed it is difficult to access their internal state; and, they reside in an append-only chain.

There is a large spectrum of possible analyses that smart contracts could benefit from [DMO+18]. Some have already been proposed in the literature and others not yet. Here is an non exhaustive list: attack analysis [LCO+16, ABC17], general metric [TDMO18] and dedicated smart contract metrics analysis, gas consumption optimization [CLLZ17], gas cost prediction, dashboard [Few06], or contract visualisation [ABC+13] to name a few.

There is a need to support the building of new generation tools to help developers [DMO+18]. Such support should tackle several important aspects taking into account the specificities of Smart Contracts: (1) the general structure and semantics of contracts, (2) the object nature of published contracts, and (3) the overall chain composed of block and transactions. Several analysis tools exist such as Oyente [LCO+16], BlockSci [KGC+17], and others [BLPB17, BDLF+16, Eth18b] but to the best of our knowledge they focus on a single aspect of smart contracts either static analysis or collecting mere data.

In this chapter, we present SmartAnvil an open platform to build software analysis tools around various aspects of smart contracts. We illustrate the general components and we focus on three important aspects: support for static analysis of Solidity smart contracts, deployed smart contract binary analysis, and blockchain navigation and query. SmartAnvil is open-source and supports a bridge to the Moose data and software analysis platform.

The contributions of this article are the descriptions of several components that compose SmartAnvil open-source platform. The outline of the article is the following: Section 1.2 describes the general architecture and components of the platform. Section 1.3 describes the low-level components for static analysis. Section 1.4 describes the low-level components to interact and analyze deployed contracts. Section 1.5 describes blockchain navigation and query support. Section 1.6 discuss the future extensions on the platform and also presents the related work on blockchain analysis platforms. Finally, in Section 1.7, we make our final remarks and conclusions.

1.2 SmartAnvil: Open Platform Architecture

The SmartAnvil open platform is structured around several components to cover the different aspects of smart contract analysis. Figure 1.1 shows the key elements that we describe in the following. While SmartAnvil can work in isolation, SmartAnvil interacts with the Moose data and software analysis platform [NA05].

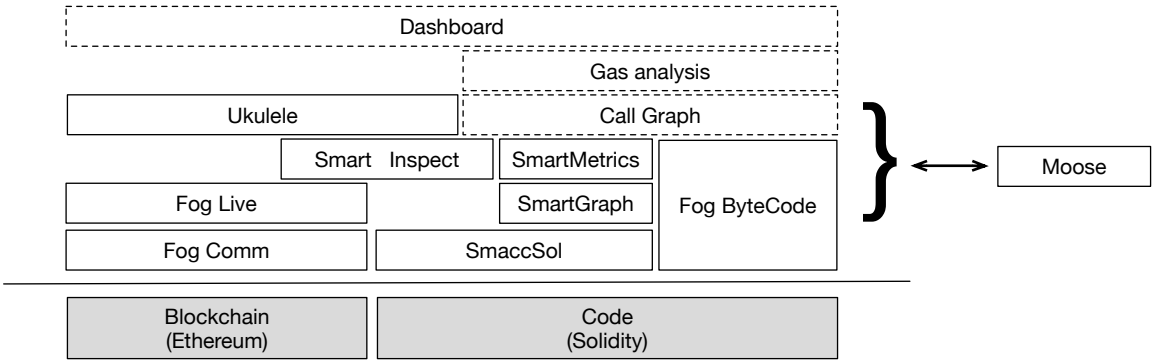

Figure 1.1: SmartAnvil components

1.2.1 Solidity Structural Components

Parser/AST (SmaccSol). The first basic but important component for static analysis is a parser/AST. The component SmaccSol offers a full parser and AST model for Solidity. It is based on the Smacc Compiler-Compiler framework [LBGD18, BGLD17]. This parser corrected early grammar errors of the Solidity language [RDDL17].

Graph (SmartGraph). Based on SmaccSol, the SmartGraph component builds a semantic tree, where it enforces the rules that cannot be defined in syntax level. It also defines scopes allowing one to easily lookup reified components, such as methods, types, parameters, arguments, contracts, etc. It supports powerful analyses and graph-like navigations by using simple queries over the tree. This component allows one to easily do static annotations or to build call-graphs.

Software Metrics (SmartMetrics). This component mixes the information provided by the previous components and the information provided by the Fog components supporting deployed contract analyses to offer a large set of metrics characterizing both the solidity code and the deployed contracts.

1.2.2 Deployed Contract Components

The other set of components provide libraries to reverse engineer deployed contracts.

Drivers and Access to Contracts (Fog Comm). This component supports the access to the GETH-RPC API, implementing the primitives and related type marshaling for accessing deployed contracts, transactions, blocks, and more [BRDD18b].

Blockchain element reification (Fog Live). This component extends FogComm by adding first citizen representations for blocks, transactions, external/internal accounts. It provides as well as contract mirrors and contract proxies – built by techniques of reverse engineering and SmaccSol's AST analysis –, for allowing a powerful connection to deployed contracts. It also provides the idea of Sessions, allowing cached and uncached interaction with the blockchain.

SmartInspect. This component leverages the implementation of contracts and mirrors in FogLive for gathering information. It exposes a contract's data in different formats to let the programmer access fine-grained information about the values of variables for a given deployed contract. It lets the developers inspect all the aspects of a deployed contract [BRDD18b].

Bytecode Reverse Engineering (Fog EVM ByteCode). This component extracts and interprets the stored hexadecimal code in deployed contracts building a representation of the instructions and the relative memory layout. It also allows one to stream instructions and cut the contents into smaller pieces as, lookup, methods, branching points etc. It complements the SmartGraph and SmaccSol AST. This low-level information is interesting for fine-grained low-level analyses, for example, gas consumption analysis.

1.2.3 Transaction Navigation

Query and Navigation (Ukulele). This component offers a query language to navigate and access transactions and deployed contracts. To achieve this, it implements different indexing strategies based on map reduce technologies and it leverages the mirrors developed in FogLive. It has a small IDE to manipulate identified entities [BRDD18a].

1.3 Foundation for Smart Contract Static Analysis

Solidity [Eth18c] is a programming language used to specify smart contracts on the blockchain platforms and in particular Ethereum [Eth14]. Solidity was originally designed to be the primary smart contract language for Ethereum. Even though other contract languages have been created for Ethereum (e.g., Serpent, LLL), Solidity is still one of the major ones. Moreover, Solidity can also be used in other blockchain platforms such as Tendermint, Hyperledger Burrow, and Counterparty. Probably because of its popularity, a great amount of smart contract research deals with Solidity. Therefore, any proposed tool handling smart contracts should support Solidity to be well received.

Usually, static analysis is employed to examine programming language artifacts. Since Solidity is a programming language and smart contracts are its artifacts, we use static analysis techniques to implement tools for contract understanding and analyses [CCI90]. At their basis, most static analysis foundation techniques rely on ASTs (Abstract Syntax Tree). For instance, it is much easier to create code inspection tools on top of an AST than to rely on the purely textual contents of a contract. Therefore, to build static analyses for Solidity contracts, we need a way to create ASTs. Consequently, we require a parser to read Solidity source code and output a relevant AST representation of it.

In addition, we need a separate infrastructure from the Solidity compiler for the following reasons:

1. since the Parser and AST represent a foundational part of a strong analysis tool, we need to fully control the parser and its production to avoid unwanted side effect in analyses using the produced ASTs;

2. since the shape of the produced ASTs can severely impact the complexity of the analysis build on top, we need to control our ASTs.

Working with JSON AST produced by the Solidity compiler did not match the above requirements. In addition, we wanted to integrate with Moose (a data and software analysis platform) because it provides a rich and versatile set of tools [NA05] with a large ecosystems [Ber16, LKG07]. Therefore, developing a Solidity parser improves the support for better code analysis in Solidity [RDDL17]. To accomplish this, we used SmaCC (Smalltalk Compiler-Compiler) [BLGD17] relying on an adapted grammar specification of the Solidity language.

1.3.1 Challenges

The main challenge of designing a parser for Solidity is the inconsistencies among the documentation [Eth18c]. For example, in one part of the documentation it is stated that string literals are enclosed by either single or double-quotes (section "Solidity in Depth" - "Types" - "String Literal"). However, in another part (section "Miscellaneous" - "Language Grammar") only double-quotes are supported to define string literals. On the other hand, the official parser recognizes both single and double-quoted string literals. This is just one example of several inconsistencies we found [RDDL17]. We decided to follow the official parser, i.e., if something is parseable in the official parser then our parser should understand it as well. Even so, the inconsistencies in the documentation made the parser creation much more difficult because we lacked a reliable specification to follow.

Another challenge to create a parser is how often the Solidity language changes. The language specification is not stable and syntactic rules are added or changed to support new versions of Solidity. For example, in version 0.4.21 (and below), there was no syntactic rule to handle a constructor, they were just a function definition with the same name as the contract. But that changed in version 0.4.22 (and above), where the constructor has a distinct syntax different from function (by using the constructor keyword). Therefore, any parser for Solidity needs to adapt to changes in the syntax as the language is still under development.

We took into account these challenges when designing and developing our Solidity parser for Pharo.

1.3.2 Approach: SmaCC-Solidity

Creating a parser can be a difficult and time-consuming endeavor. Parser generators provide an easier way to tackle this problem [IM15]. Basically, we write the syntax specification using a formal grammar and the generator automatically builds the parser. For example, there are many parser generation tools available nowadays such as YACC [Mer93], ANTLR [Mil05], and SmaCC [BLGD17].

Generally, we can classify a parser as either top-down or bottom-up. A top-down parser starts its productions from the root element of the grammar working its way down to the bottom leaves. A bottom-up parser works vice-versa, starting from the leaves and working up to the root. Moreover, either type of parser usually works on a subclass of grammars. The most common subclasses are LL(k) for top-down, and LR(k) for bottom-up parsers [ALSU06].

When we consider the Pharo Smalltalk environment, there are two prominent parser generation tools: PetitParser [KLR13], and SmaCC [BLGD17]. Therefore, we had these two options to create our parser. We could also write the parser ourselves without relying on generators. We chose to develop our parser using SmaCC for the following reasons: (i) SmaCC requires a textual context-free grammar

as input that is similar to the grammar provided for Solidity; (ii) SmaCC generated parser can adapt more easily to future changes in the Solidity grammar; and (iii) SmaCC produces an LR parser, which has interesting advantages over other types of parsers [ALSU06]. Moreover, both PetitParser and a manually written parser would be more difficult to adapt and maintain than SmaCC. Therefore, we claim that SmaCC is the better option to create our parser.

The SmaCC-Solidity parser is publicly available in GitHub (*github.com/smartanvil/SmaCC-Solidity*).

1.3.3 Illustrative Example

Usually, someone uses a parser when he/she needs to perform static analysis on the source code. Let's suppose a developer who wants to create a Token contract. Roughly speaking, a token is a custom form of cryptocurrency managed by a contract. A token could represent loyalty points, bonds, game items, etc. In this scenario, the developer wants to first understand a well-used token before coding his own. In this example, the developer gets the GolemNetworkToken (address: 0xa74476443119A942dE498590Fe1f2454d7D4aC0d) contract code and decides to build a visual representation of it. By using our parser and some classes from the Roassal package [Ber16], a few lines of code (Listing 1.1) is all it needs to build a simple visualization.

Listing 1.1: Visualization using SmaCC-Solidity

```
1  ast := SolidityParser parse: sourcecode.
2  b := RTMondrian new.
3  b nodes: (ast sourceunits) forEach: [ :contract |
4      (contract className = 'SolContractDefinitionNode')
5      ifTrue: [ b shape box color: Color blue.
6          b nodes: (contract statements select:
7              [ :stt | stt className = 'SolStateVariableDefinition'] ).
8          b shape box color: Color gray; size: [ :f | f source lines size * 3].
9          b nodes: (contract statements select:
10             [ :stt | stt className = 'SolFunctionDefinitionNode']).
11         b layout flow ] ].
12 b.
```

In Listing 1.1, first we get the AST from parsing source code (line 1). Then, we create a new Roassal visualization [ABC+13] where each contract is a big gray box (lines 2-4). Inside each contract box, we get the state variable definitions (i.e., contract attributes) and create blue boxes representing it (lines 5-7). Moreover, for each function, we also create a gray box inside the contract, and the size of the function is related to its lines of code (lines 8-10). Finally, we define a standard layout for the visualization (line 11) and "return" the visualization object for show (line 12). The resulting visualization shows each contract on a big box, with smaller boxes inside representing attributes (in blue) and functions (Figure 1.2).

1.3.4 Evaluation

For comparison, we verified if our parser could recognize the same Solidity contracts as the official parser. It is reasonable to assume that the official parser is the correct implementation. Therefore, any contract parsed by the official should also be supported by our implementation. We used Etherscan to

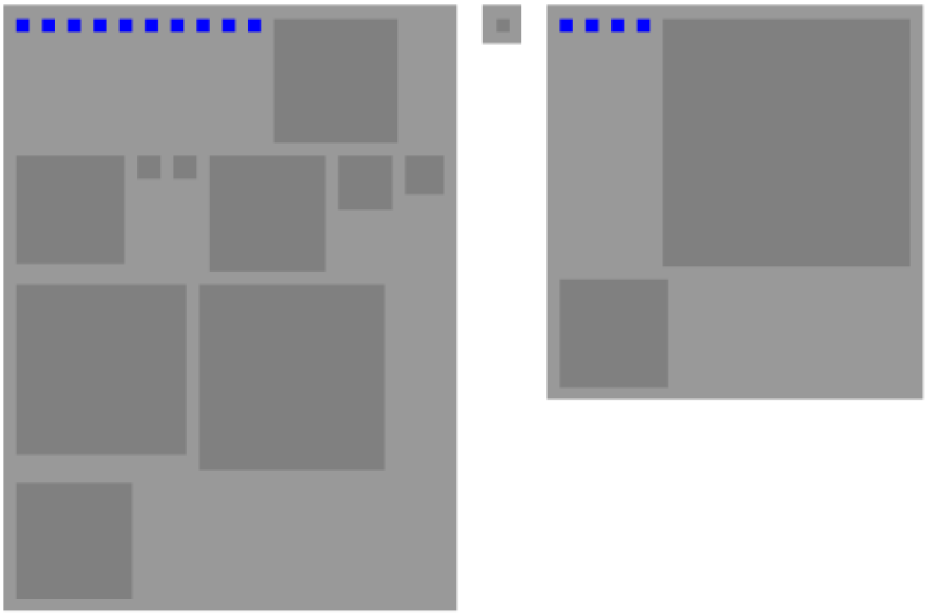

Figure 1.2: Contract Functions using SmaCC-Solidity

access Solidity contracts that were publicly available with its source code. All such contracts were correctly recognized by the official parser. Therefore, our evaluation was to verify if our implementation could parse them as well.

At the time of the experiment (August 2017), Etherscan had approximately 2.7K verified contracts in its database (in August 2018, the number of contracts was more than 40K). From those 2.7K, we selected only unique contracts from Solidity version 0.4.x for a total of 1,175 contracts. Our implementation successfully parsed all 1,175 contracts. Consequently, we can claim our parser recognizes the same language artifacts as the official one.

1.3.5 Related Work

We selected related work from two main topics: parsing conflicts and parser generators.

Isradisaikul and Myers [IM15] describe the difficulty in solving grammar conflicts with LALR parser generators. They propose an algorithm that generates better counterexamples for LALR parsers. A counterexample shows a parsing scenario that causes an ambiguity conflict in the LR method. Such scenarios help developers diagnose the conflict's source and identify problems in the grammar specification. The authors implemented their algorithm in Java as an extension to the CUP parser generator and they also evaluated their approach in 20 grammars against a state-of-the-art ambiguity detector. Their algorithm usually finds more counterexamples in less time than the compared techniques.

Passos et al. [PBB07] proposes a methodology to resolve conflicts in the LALR parsing method. The authors describe the challenges for handling conflicts without changing the defined language. They used YACC to generate a parser for the Notus language to illustrate the difficulty to resolve conflicts for LALR(1). The authors created a tool, called SAIDE, based on their methodology which is an improvement over the regular methods available to handle conflicts in LALR parsers.

We are going to give a brief analysis of some well-known parser generation tools. SmaCC [BLGD17] is the parser generation tool we chose for this research. SmaCC can generate either LR(1) or LALR(1) bottom-up parsers. YACC [Mer93] is also a bottom-up LALR(1) parser generator. The original YACC was developed in C for Unix systems, but now we have implementations of it on several other languages. ANTLR [Mil05] differs from the previous tools because it is a top-down parser generator. ANTLR creates predictive LL(k) parsers, which are less sensitive to grammar irregularities than LR(1) and LL(1) parsers. There are implementations of ANTLR for languages such as Java, C++, C#, JavaScript, Python, and others. Unfortunately, as far as we know, there is no implementation of ANTLR for Pharo. Finally, PetitParser [KLR13] is a scannerless parser generator that relies on parsing expression grammars. PetitParser is integrated with Moose, which facilitates its use on Pharo. The main problem with PetitParser is it does not read a grammar specification, and we must write the grammatic expressions using the PetitParser language.

1.4 Deployed Smart Contract Analysis: Inspection

Opaqueness is the major issue to analyze contract data [BRDD18b]. Smart contracts are opaque in the sense that once deployed in the blockchain it is very difficult to verify their internal state at run-time (i.e., the state stored in the contract defined by its internal attributes). By contrast, nowadays any programming language offers simpler ways to inspect object data. Developers use such inspection to access run-time data during development or maintenance activities. Similarly, developers could benefit from smart contract run-time inspection to verify the currently stored data. Moreover, from a business perspective, companies could use contract inspection to help clients verify and understand the information that is actually stored in the contract.

The difficulty to inspect contract data is not a widely known problem [BRDD18b]. Nowadays, we have few tools to inspect contract information. Alternatively, developers resort to other practices to acquire a contract's internal state such as creating getter functions to access data. For this reason, it is important to be able to develop tools that can be used for inspecting internal attributes. Moreover, the general concern is that a developer should be able to see the values stored in his own contracts.

Therefore, in this context, inspecting contract attributes is not only important but necessary for developers working with smart contracts.

1.4.1 Challenges

There are many challenges to pierce through the opaqueness problem and reveal contract information.

- **Binary and incomplete specification.** From the technical aspects, we only have the Ethereum API to access a binary representation of the contract. The first challenge we faced is an *incomplete* specification of the contract encoding performed by the Solidity compiler [Eth18c].

- **Inconsistent specification of hash computation.** Another challenge related to the specification is the hash computation for dynamic types. Static types use text as input for the hash calculation. However, dynamic types follow a different standard that it is not clearly specified in the documentation. For dynamic types, it is necessary to use binary data packed specifically for

each type. For example, to access an array it is necessary to pack the index number and the array position (offset) into a binary representation to obtain the correct hash. Other dynamic types would require a different input to get its hash.

- **Packed and ordered data.** We also highlight the challenges on decoding types, as the compiler packs as much data as possible into contiguous memory. Therefore, we need to know the specific types in the correct order to acquire the contract data, and that is not an easy task when we have an incomplete specification.

We acknowledge as a problem the challenges and difficulties of analyzing our own contracts deployed in the Ethereum blockchain database. To solve this problem we propose an inspector that allows the user to perceive a clean representation of the instance's data attributes. Such inspector is built using several components of the platform: FogComm, FogLive.

1.4.2 Approach: SmartInspect

SmartInspect [BRDD18a] is an internal state inspector for Solidity contracts based on pluggable mirror-based reflection system. The goal of this tool is to allow the inspection of known contract instances based on their source code. The binary structure of the deployed contract is decompiled using a memory layout reification built from its source code. Therefore, only people with access to the source code and its deployed binary representation will be able to inspect its internal attributes. Moreover, SmartInspect approach can introspect the current state of a smart contract instance without the need to redeploy it nor develop additional code for decoding. SmartInspect is implemented in Pharo and it is publicly available in GitHub as part of the SmartAnvil tool suite (*github.com/RMODINRIA-Blockchain/SmartShackle*).

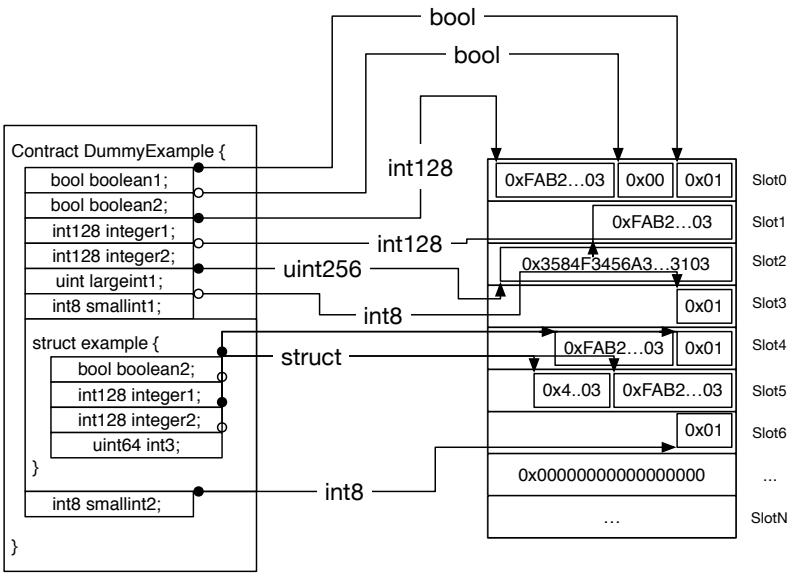

Figure 1.3: Memory Layout Representation

The general idea of the Smart Inspector is to decompile the storage layout encoded by the Ethereum API. By using the contract source code as reference, it is possible to know where the attribute values are stored in the encoded binary structure (Figure 1.3). It was a challenge to decompile the memory layout because the Solidity documentation [Eth18c] is incomplete regarding how some types are encoded. Therefore, we had to reverse engineer some of the encoding performed by the compiler ourselves.

After we decoded the memory layout, we still needed to apply our solution to any contract for a general reusable solution. We employ a mirror-based architecture[BU04] that mimics the structure of any contract for us to access the memory layout that we can decode. A mirror works like an independent meta-programming layer which splits the concern of reflection capabilities into a mirror object. First, we require the contract source code as input to start building the mirror. Then, we parse the source code (using our Solidity Parser described earlier) to create an AST (Abstract Syntax Tree). By interpreting the AST, we are able to discover every type declared in the contract in the correct order. We require this information to decoded the memory layout and access the contract data. Aiming at a general solution, we model configurable mirror objects that allow us to interact with deployed contract instances of the same configuration (usually meaning the same contract deployed in the blockchain).

Figure 1.4 shows a diagram of the whole process to inspect a contract, the pluggable reflective architecture generates a mirror for a given contract's source code by using its AST (Abstract Syntax Tree). Then, we use this mirror to extract information from a remote contract instance deployed in the blockchain (which is encoded as a binary memory layout). The contract data we gathered is exposed in four different formats: (i) data proxy object (REST), (ii) Pharo widget user interface, (iii) JSON, and (iv) HTML.

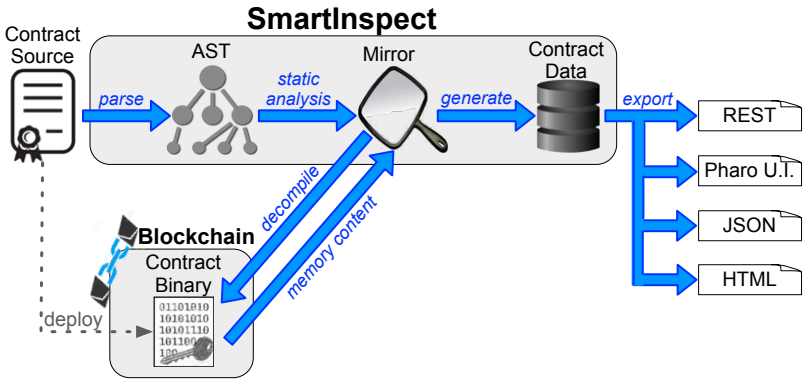

Figure 1.4: SmartInspect building process.

This approach allows us to access remote structureless information in a structured way. Our solution meets most of the desirable properties that are important for remote inspection namely: interactiveness and distribution [PBD+15].

1.4.3 Illustrative Example

We now present an example of a smart contract written in Solidity and SmartInspect introspecting its contents. Let's suppose a scenario where a supervisor manages a poll that users are allowed to vote one single time. The poll is coded into a smart contract because it is used for management decisions that rely on the veracity and immutability of the information. Only the contract's supervisor is allowed to modify the list of voters. The following listing highlights the most important code parts for this contract:

Listing 1.2: Solidity Poll Contract Example

```
1 pragma solidity ^0.4.24;
2 /** @title Poll Contract Example
3   * @author RMoD-Blockchain Team
4   */
5 contract RmodPoll {
6    /* Type Definition */
7     enum Choice { POSITIVE, NEGATIVE, NEUTRAL }
8     struct PollEntry { address user; Choice choice; bool hasVoted; }
9
10   /* Attributes / Internal State */
11    PollEntry[] pollTable;
12    address private supervisor;
13
14    /* Constructor */
15    constructor() public{ supervisor = msg.sender; }
16
17    /* Functions */
18    // Add the account as a voter in this poll
19    function addVoter(address account) public returns(uint){ ... }
20    // Verifies if the account is already registered in this poll
21    function isRegistered(address account) public view returns(bool){ ... }
22    // Returns the Voter's index in the pollTable
23    function voterIndex(address voterAccount) private view returns(int){ ... }
24    // Function called by the voter to assert his vote
25    function vote(Choice myChoice) public { ... }
26    // Verifies if everyone registered for this poll have voted
27    function everyoneHaveVoted() public view returns (bool){ ... }
28    // Return the amount of votes for Positive, Negative, and Neutral
29    function countVotes() public view returns (uint,uint,uint){ ... }
30 }
```

This contract defines two custom types: Choice (line 5) and PollEntry (line 6). A Choice models the answers to the poll (whether the vote was positive, negative, or neutral). A PollEntry is a record representing a vote, i.e., the voting user, the selected option, and if he/she has voted or not. Note that to refer to the voter we need an account address (using the primitive type address) that refers to an Ethereum account. The contract stores internally a poll table (an array of PollEntry, line 9) and an address to the poll's supervisor account (line 10). The poll table is an empty array where the supervisor will eventually store the poll information (i.e., the array will have an entry for each user that is allowed to vote). The supervisor's address is used for security checks, i.e., to ensure that only

the supervisor can call some specific functions (e.g., addVoter function). It is not necessary to define and explain the remaining functions for this illustrative example.

Let's suppose that the poll needs to be closed soon and not everyone has voted. Therefore, the poll's supervisor will need to send reminders to the users who haven't voted yet. However, the contract's attributes were not defined as public. Moreover, we did not create any function to check for that information before deploying the contract. Since we cannot update the source code of a deployed contract instance, inspecting its internal state is the only way to know the accounts that have not voted. Because the supervisor has access to both the contract's source code and its deployed instance, he/she can execute SmartInspect to see the contract's data (Figure 1.5). Finally, the supervisor can see which users did not vote for this current poll instance.

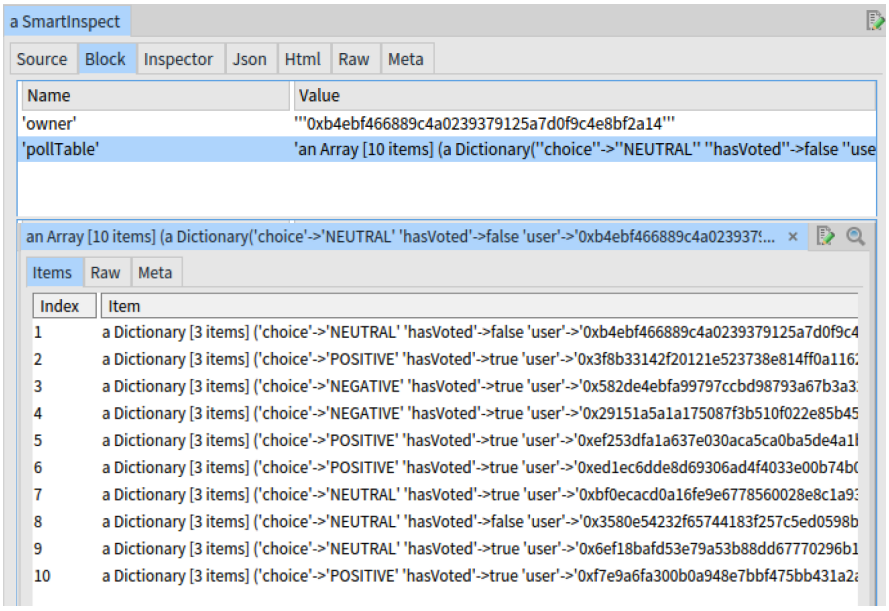

Figure 1.5: SmartInspect Pharo User Interface Screenshot

1.4.4 Evaluation

We investigate whether SmartInspect implements the necessary and desirable features when compared to other inspectors. We identified six characteristics that are important for a blockchain remote inspector: Privacy, Extendability, Pluggability, Consistency, Reusability, and Instrumentation. We detail the characteristics as follows:

- **Privacy:** inspection should not breach or compromise data privacy by exposing data to unauthorized users. When considering smart contracts, a lack of privacy is dangerous as it could be exploited by malicious users to acquire illicit advantages and resources.

- **Extendability:** the inspector can be extended for other technologies. Ideally, a debugger or inspector should rely on a middleware that is extensible. For smart contracts, the inspector should be extensible over different smart contract languages and blockchain technologies.

- **Pluggability:** the inspector can be used on existing objects without the need to re-instantiate the objects or the system. For contracts, this means we can inspect existing deployed contracts without any dependency on the contract side, or the need to redeploy the contract. An unpluggable approach has the disadvantage of requiring the redeployment of a contract, which has non-trivial transactional costs.

- **Consistency:** the representation used by the inspector must reveal the information in a consistent manner, i.e., the inspection must reflect the current state of the deployed contract.

- **Reusability:** the inspector can be reused for different contracts. Lack of reusability would require a developer to spend time redefining the inspection for each individual contract.

- **Instrumentation:** the inspector can alter the semantics of a process to assist in debugging. Basically, this is the mechanism to halt the process and inspect it at that point (e.g., breakpoints and watchpoints). This characteristic is not possible in blockchain platforms, as we cannot modify the deployed contract code to halt a function in the middle of its execution in the blockchain.

We analyzed SmartInspect and two other inspectors (Remix and Etherscan) according to the aforementioned characteristics for inspectors. We also compared two practices for accessing contract data: ad-hoc Decoder, and Getter methods (Table 1.1).

Table 1.1: Related techniques comparison

Characteristic	SmartInspect	Remix	Etherscan	Decoder	Getter
Privacy	Yes	Yes	No	Yes	No
Extendability	Yes	No	No	No	No
Pluggability	Yes	Yes	Yes	Yes	No
Consistency	Yes	Yes	Yes	Yes	Yes
Reusability	Yes	Yes	Yes	No	No
Instrumentation	No	No	No	No	No

As we can see from Table 1.1, SmartInspect's only characteristic flaw is related to *instrumentation* for remote debugging. However, this is a limitation imposed by the Ethereum blockchain technology rather than a design flaw in our approach.

Remix [Eth18b] is a suite of tools designed by the Ethereum Foundation for Solidity developers. Remix IDE is probably the most used tool in its suite. In the Remix tool suite, there is a full transaction debugger with inspection capabilities. Remix inspector only falls short in extendability and instrumentation. It was designed specifically for Solidity and Ethereum, and as such, it is not extensible to other platforms. As we previously explained, instrumentation is not possible due to the characteristics of the Ethereum platform.

Etherscan is an analytics platform for Ethereum. It provides live information from Ethereum on blocks, transactions, contracts, etc. If a developer publishes his/her contract source code on Etherscan, then it will be possible to inspect the contract's data. However, by doing so the developer will expose his contract source code and internal data to anyone, in a very easy way to access it. That is the reason

why Etherscan inspection lacks privacy (unlike SmartInspect and Remix which can expose the data to the developer only). Etherscan provides an API to access its services, but it is not extensible. Just like SmartInspect and Remix, Etherscan's inspector cannot support instrumentation.

An ad hoc Decoder uses the Ethereum API on the memory slots to manually fetch the data. This is a complex task since it demands a deep understanding of the memory layout of each contract a developer plans to inspect. It also requires a developer to know the type of each attribute and code the ad-hoc decoder accordingly. Its advantages are that it allows access to data without loss of privacy and without the need to redeploy the contract. In fact, SmartInspect uses this concept of decoding memory layouts as a part of its inspection process.

Getter methods are a simple solution since they are cheap to implement and easy to test. The developer does not need to know the memory layout of a contract to create getter methods. However, if the developer forgets to make a getter for a given attribute, he/she will need to re-deploy the contract and, most often, lose the data from the previous instance. Solidity does not support the return of many complex types (e.g, structs, mappings) on its functions. Therefore, a developer might need to adapt his/her data or function to provide access to a complex type. Moreover, the easy access to the data may cause a loss of privacy, since getter methods are a public part of the contract binary encoding.

1.4.5 Related Work

Chis et al. [CNSG15] performed an exploratory research to better understand what developers expect from object inspectors, and based on that feedback they propose a novel inspector model. The authors interviewed 16 developers for a qualitative study, and a quantitative study conducting an online survey where 62 people responded. Both studies were used to identify four requirements needed in an inspector. Then they propose the Moldable Inspector, which indicates a new model to address multiple types of inspection needs. We followed the lessons taken by the Moldable Inspector when creating SmartInspect. We deem noteworthy the multiple views aspect, as SmartInspect can present its inspected data in four different views (REST, Pharo U.I., JSON, and HTML).

Papoulias [PBD+15] gives a deep analysis of remote debugging. As discussed by the author, remote debugging is especially important for devices that cannot support local development tools. The author identifies four important characteristics for remote debugging: interactiveness, instrumentation, distribution, and security. Based on the identified properties, Papoulias proposed a remote debugging model, called Mercury. Mercury employs a mirror-based approach and an adaptable middleware. We used Papoulias research as an inspiration to create SmartInspect, especially relying on a mirror for remote inspection.

Salvaneschi and Mezini [SM16] propose a methodology, called RP Debugging, to debug reactive programs more effectively. The authors discuss that reactive programming is more customizable and easier to understand than its alternative the observer design pattern. The authors also present the main problems and challenges to debug reactive programs, and the main design decisions when creating their methodology. Although our inspector is from a different application domain, the RP Debugging design served as an inspiration to plan our own inspecting approach.

Srinivasan and Reps [SR14] developed a reverse engineering tool to recover class hierarchical information from a binary program file. Their tool also extracts composition relationships as well. They use dynamic analysis to obtain object traces and then they identify the inheritance and composition

information among classes on those traces. The authors' experiments show that their recovered information is accurate according to their metrics. The author's tool contrasts with SmartInspect as we use static analysis and they use dynamic analysis.

1.5 Chain Navigation and Query Language

Ethereum and other blockchains store a massive amount of heterogeneous data. In early 2017, Ethereum was estimated to have approximately 300GB of data [BLPB17]. Retrieving information from this massive data is not an easy task. Moreover, the Ethereum platform only allows direct access to its *first-class* data elements, which includes blocks, transactions, accounts, and contracts [Eth14]. Therefore, if we search for a particular information inside a data element, we would need a unique identifier (i.e., either the number or its hash) to access the block containing such information. Another alternative would be to direct access one block and sequentially access its parents to search for a data. Moreover, the information returned by the Ethereum platform when we access its blocks is encoded into a JSON-like structure that we need to interpret to acquire a specific item. Therefore, the Ethereum platform does not provide a semantic way to search for information and neither an easy form to present such information.

In a classical database, when we need to search for a particular information, we usually write a query to fetch, present, filter, and format such information. Database query languages like SQL provide a rich syntax to describe the data we want to acquire from the database. Since blockchain can be considered a database, it would be better if we could use a similar way to fetch information inside the blocks. In this context, users could benefit from a query language providing an easier way to fetch, format, and present the information extract from the blockchain platform.

1.5.1 Challenges

In this section, we describe in more detail the challenges when trying to acquire data from a blockchain. Although our research is focused on Ethereum, such problems are also present on other blockchain platforms as well.

- **Massive Data.** Blockchain databases already possess a massive amount of data. In December 2017, Ethereum processed, on average, 876K transactions per day [BRDD18a]. Since the data in the blockchain cannot be deleted, the number of recorded transactions will only grow in time. Consequently, older information could get overwhelmed by new transactions and become "lost in time".

- **Heterogeneous Data.** Blockchain not only stores a great amount of data but also manages a mixture of *first-class* elements such as transactions, blocks, accounts, and smart contracts. All of the *first-class* elements are different but interrelated by hashes. Even though Ethereum allows access to any of its *first-class* elements, the heterogeneity of the elements (each one with different meaning and high-level representation) complicates the acquisition of information.

- **Data Opaqueness.** For flexibility reasons, Ethereum stores its information using a generic representation. Therefore, the stored data is opaque, since there is no meta-data describing the

information and neither a simple way to know what was recorded. This opaqueness is useful and even necessary from the Ethereum standpoint because it allows the storage of arbitrary structures and behaviors. On the other hand, the opaqueness over-complicates searching for information, since a user needs to access generic representations without any knowledge of its content.

- **Direct Access.** In general, blockchain only allows direct access to its elements by using a unique identifier. This identifier is a hash number that is generated for every *first-class* element stored in the blockchain [Bit18]. The Ethereum platform, in particular, can also use a unique number (related to the order of the element) to access blocks and transactions [Eth18a]. The direct access complicates the user task in finding information because he will either need to: (i) store the identifiers for later usage, or (ii) perform a sequential access to gather the data.

To address these challenges, we proposed a query language that enables its users to acquire information more easily. We highlight the following benefits of using a query language to fetch information from blockchain: (i) describe structural and semantic filters to query for information; (ii) reformat and transform the acquired data; (iii) order the query results; (iv) limit the number of results returned.

1.5.2 Approach: Ukulele

In this section, we present the Ukulele version 0.9, a query language designed to acquire information from the blockchain. Ukulele is the improvement of our previous work called Ethereum Query Language (EQL) [BRDD18a]. Ukulele is publicly available at GitHub as part of the SmartAnvil suite (https://github.com/smartanvil/UQLL). Ukulele syntax is based on the SQL language. The idea is to allow users to write queries as close to SQL as possible to facilitate the adoption of Ukulele since SQL is a very popular language [SKS11, EN10].

Listing 1.3 shows the main elements of the Ukulele syntax. The syntax is described using EBNF (Extended Backus-Naur Form), we did not format the terminals *identifier* and *number* in double-quotes to highlight that they are not literals. We omitted the Expression rule to not clutter the specification, but it follows a similar structure of SQL expressions.

Listing 1.3: Ukulele Grammar in EBNF format

```
1 <SelectStatement> ::= <SelectClause>  <FromClause>
2              [<WhereClause>]  [<OrderByClause>]  [<LimitClause>]
3 <SelectClause> ::= "select" <Expression> {"," <Expression> }
4 <FromClause> ::= "from" <SourceBind> {"," <SourceBind> }
5 <WhereClause> ::= "where"  <Expression>
6 <OrderByClause> ::= "order" "by" <Expression> [ "asc" | "desc" ]
7 <LimitClause> ::= "limit" number
8 <SourceBind> ::= identifier "as" identifier
9 <Expression> ::= ...
```

As we can see from Listing 1.3, the syntax for Ukulele queries is very similar to the "Select" statement from the SQL language. The Ukulele "Select" statement consists of the following clauses: *select*, *from*, *where*, *order by*, and *limit*. Only the *select* and *from* clauses are required, the others are optional clauses. We also like to highlight that Ukulele similarly to SQL is also case-insensitive.

Unlike SQL, the Ukulele "Select" statement does not have a *group by* clause. Although we have plans to support a *group by* clause in the future, the current version does not allow the usage of such clause.

In the *from* clause of Ukulele, we need to use a collection. In Ukulele, a collection is a semantic representation of queried data. The Ukulele language have four predefined collections: *ethereum.blocks*, *ethereum.transactions*, *ethereum.accounts*, and *ethereum.contracts*. Each one of those collections represents all first-class elements from a particular type (blocks, transactions, accounts, or contracts). We can use 'eth' as an alias for 'ethereum'. It is possible to create custom collections from a subset of another one by creating "views" similar to SQL (Listing 1.4). Just like SQL that views can be used in place of a table in the *from* clause, so does Ukulele views and collections.

Listing 1.4: Ukulele View Grammar in EBNF format

```
1 <View> ::= "create" ( "view" | "collection" ) identifier "as"
2            "(" <SelectStatement> ")"
```

Ukulele already has a predefined structure of objects to act as a container for collection items. For example, when we retrieve information from a collection of blocks, the result will be presented as block objects. Therefore, each object type has attributes related to their storage structure (i.e., a block object will have different attributes than a transaction object). The attributes available for each object type are the following:

- **Block:** number, hash, parentHash, parent, nonce, timestamp, size, miner, difficulty, totalDifficulty, gasLimit, gasUsed, extraData, transactionsRoot, transactionsHashes, transactions, amountOfTransactions, uncleHashes, uncles.
- **Transaction:** hash, nonce, blockHash, blockNumber, block, transactionIndex, fromAddress, from, toAddress, to, value, gasPrice, gas, input, timestamp.
- **Account:** address, name, balance.
- **Contract:** descriptions, instances.
  - **Contract.Instance:** address, name, balance, binaryHash, bytecode.

For a more detailed description of each attribute, see our first publication on EQL [BRDD18a]. One major improvement over EQL is handling contracts. For example, we can use contracts' internal attributes or functions in Ukulele queries. Moreover, we can handle the general structure of a contract (by using the contract object) or the deployed instances (the instances attribute of a contract object).

Internally, our implementation of Ukulele relies on indexes to increase its performance when querying data. Since each access to the blockchain to fetch data is a remote call, any form of optimization can improve the time required to acquire and present data. An index is a summarization of data stored into a structure that improves the performance of retrieval operations. The basic idea is to allow a more efficient search into the database. In our particular case, we index blockchain data (e.g., blocks, transactions) with its related hash to speed up fetching data.

For Ukulele, we implemented a Binary Search Tree (BST) to serve as the index structure. The BST employs a two-dimensional array where the first dimension of each entry is used for storing the property value, and the second dimension is used for storing a set of hashes to the elements

that correspond to this specific value. We chose this implementation because of its simplicity for selecting an interval of indexes in any comparison operation, such as "greater than", "lesser than". We acknowledge that BST has a high storage requirement. However, we wanted a simple and fast solution to our first implementation of Ukulele. Moreover, Ukulele builds these internal indexes automatically, without the need for user interaction. On the other hand, Ukulele also supports that users create their own custom indexes, to speed up queries. The syntax to create custom indexes is similar to SQL (Listing 1.5).

Listing 1.5: Ukulele Custom Index Grammar in EBNF format

```
<Index> ::= "create" ["unique"] "index" identifier "on" identifier "(" identifier
    ")"
```

1.5.3 Illustrative Example

For this example, we will show some of the new features of Ukulele (i.e., contract querying). Let's suppose a smart contract that handles the selling of a product. The following listing shows the essential code parts for the product contract.

Listing 1.6: Solidity Product Contract Example

```
pragma solidity ^0.4.24;
/** @title Product Contract Example
  * @author RMoD-Blockchain Team
  */
contract RmodProduct {
   /* Type Definition */
    enum State {ON_SALE, WAITING_SEND, RECEIVED, FINISH }

    /* Attributes */
    State private state = State.ON_SALE;
    address private owner;
    address private buyer;
    uint private price;
    string private itemName;

    /* Constructor */
    constructor(string _name, uint _price) public {
        owner = msg.sender;  itemName = _name;  price = _price;
    }

    /* Functions */
    // Returns the this product information (name,price)
    function getInfo() public view returns(string,uint) { ... }
    // Buys this product
    function buy() payable public { ... }
    // Inform the seller/owner the product was properly received
    function received() public { ... }
    // Completes the sale, the owner terminate the contract and get the money
    function terminate() public { ... }
}
```

As we can see, the above contract (Listing 1.6) can be used to sell any product. In our example, the user only wants to query only on these product contracts. Therefore, he/she creates a view for this task (Listing 1.7). The "binaryHash" attribute of an instance is an MD5 hash of the contract's binary structured. Thus, by using this attribute someone can get all contracts with the same internal structure. In this case, the view query (Listing 1.7) is getting only RmodProduct contracts (Listing 1.6).

Listing 1.7: Ukulele Create View Example

```
1 CREATE VIEW RmodProductContracts (
2     SELECT instance
3     FROM eth.contract.instances AS instance
4     WHERE instance.binaryHash = '4d3c0777436de51258c7ba6c235a2e52'
5 );
```

Now, the user can employ the view to more easily query only product contracts. In this example (Listing 1.8), the user wants cheap products that are still up for sale.

Listing 1.8: Ukulele Contract Query Example

```
1 SELECT instance.itemName, instance.price, instance.state
2 FROM RmodProductContracts AS instance
3 WHERE  instance.price < 40 AND instance.state = 'ON_SALE';
```

Figure 1.6 shows the results from the above query.

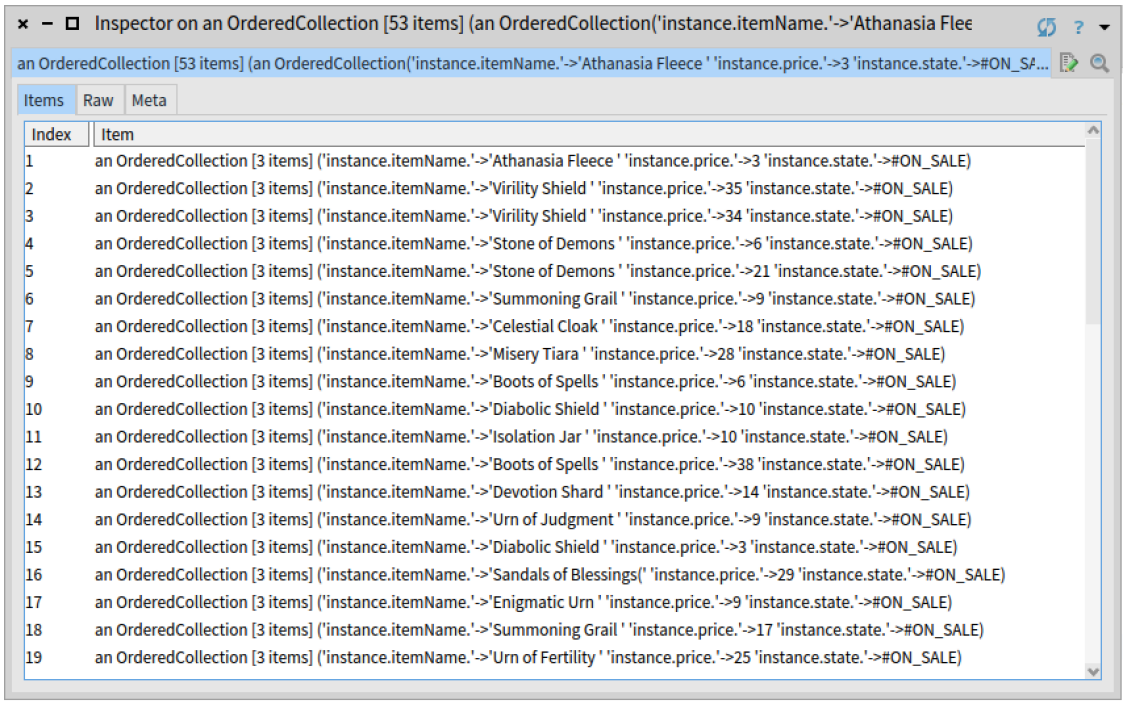

Figure 1.6: Ukulele Contract Query Example Results

Ukulele can also execute contract functions. Besides using functions for querying values, this feature can be used to execute a function that changes the contract state in a batch. For example, let's

suppose a user who wants to collect the sales on his products. Without Ukulele, the user would have to execute the function 'terminate' on each contract manually and one at the time. Using Ukulele, the same user can just write a simple query (Listing 1.9).

Listing 1.9: Ukulele Block Query Example

```
SELECT instance.itemName, instance.terminate()
FROM RmodProductContracts AS instance
WHERE instance.owner = '0xca35b7d915458ef540ade6068dfe2f44e8fa733c'
    AND instance.state = 'RECEIVED';
```

1.5.4 Evaluation

We compared Ukulele against Presto and Web3, on the supported features designed for acquiring data in the blockchain. The features we analyzed were the following:

- **Query Language:** the implementation offers a query language to specify the data you want to fetch. Lack of a query language can make it more difficult to acquire complex information.
- **Group Functions:** the implementation supports the use of aggregate (group by) functions. Lack of group functions limits transformation on the data and what it is possible to acquire.
- **Join Data:** checks whether it allows different data sources to be joined. Ideally, it should allow merging data from different platforms as well (e.g., blockchain and No-SQL).
- **Views:** indicate if the implementation supports the definition of views or some similar feature. Views allow the user to save his queried data for later usage.
- **Tool Support:** see if there is a tool support to specify, check, and show the acquired data as well as the commands used to fetch such data. Lack of tool support can hinder the adoption of the implementation.
- **Data Source:** checks if the implementation can fetch any type of first-class elements stored in the blockchain.

Table 1.2: Related techniques comparison

Feature	Ukulele	Presto	Web3
Query Language	Yes	Yes	No
Group Functions	No	Yes	No
Join Data	Partial	Yes	No
Views	Yes	No	No
Tool Support	No	No	No
Data Source	All	Partial	All

Web3 is the official Ethereum API coded in Javascript that implements an interface to the Remote Procedure Call (RPC) protocol [Eth18a]. Web3 is not a query language, all data is gathered by using

its API interface. Even though Web3 can access any data stored in Ethereum, it can only fetch by direct and sequential access. Therefore, the user must search the information manually among the collected data. Although not a very polished solution to gather data, Web3 and its underlying protocol is the basis to access Ethereum data. Both Ukulele and Presto use Web3 (or another related API) to access the blockchain platform and perform its search.

Presto is a querying engine to run SQL on other platforms, including Ethereum blockchain. Moreover, Presto supports full SQL syntax for querying such as joins, group functions, etc. However, it does not support another auxiliary syntax such as views. There is no tool support to edit queries or view the results. Another flaw in Presto is that it can only acquire Ethereum data related to Blocks and Transactions. Therefore, Presto cannot show information on contracts or accounts.

Ukulele is a query language implemented to automate searching for information in Ethereum. As we previously mentioned, Ukulele is still under development and the current version does not support group functions. Moreover, the current version of Ukulele is able to join multiple data sources from Ethereum but it is not possible to merge such data with different platforms (such as NoSQL). There is no tool support for Ukulele, although one is currently under development. The major advantage the Ukulele has over Presto is that Ukulele can fetch any data from Ethereum first-class elements (blocks, transactions, contracts, and accounts).

1.5.5 Related Work

In this section, we discuss some related work on blockchain research focused on search and navigation. Tools and implementations (e.g., Presto) that were not originated by research are not presented here.

Morishima and Matsutani [MM18] propose a new way to search for blockchain information by using GPUs. They argue that blockchain full nodes could turn into a bottleneck for the platform if the search performance is not improved. The authors evaluate the performance of their approach against the existing practice and other methods; and conclude that their proposal throughput is 3.4. times higher than the second highest approach. Although the work of Morishima and Matsutani is about searching blockchain data, it has a different purpose than Ukulele. Ukulele was designed to help users search for complex data. On the other hand, Morishima and Matsutani designed their approach to improve the performance of searching for data in blockchain nodes. The similarity is that both research deals with the internal structure of the blockchain and searching for data in that platform.

Ruta et al. [RSI+17] discuss a new semantic-enhanced layer for blockchain platforms focusing on the supply chain application domain. They propose a framework for on-line object discovery to implement their layer on a Hyperledger blockchain. Their approach adopts a semantic matching between queries and objects, which allows users to find supply chain information more easily. Unlike Ukulele, there is no query language. However, it does use semantic information to fetch blockchain information. Ukulele provides a rich syntax for querying information, but it does not use semantic-enhanced objects and information to improve the search.

1.6 Discussion and Related Work

In this section, we discuss our future research and tools and then we present related work on blockchain analysis platforms and tool suites.

1.6.1 Future research and tool development

As a platform SmartAnvil should support a large range of analysis. We are focusing on the foundations of the tools. We see the following topics as the next components around SmartAnvil.

Metrics. It is not possible to control what you do not measure; and such statement is the basic wisdom on why we need to use metrics [Hev97]. Metrics are useful to assess the internal quality of a software as well as the productivity of the development team. Since the 90s, there has been a plethora of proposed software metrics [LH93, CK94, LK94, HM95, BMB96, Hev97, TNM08]. The reason for so many different metrics is because specific domains or characteristics require measurements tailored for its particular needs. Usually, applying a metric designed for one domain to another leads to mismatch or incompatibility. For instance, the domain mismatch led to procedural programming metrics to be adapted for the object-oriented paradigm.

Due to the extremely fast growing pace of smart contract usage, in this new software paradigm measuring code quality is becoming as essential as in out-of-chain software development. However, traditional software metrics do not capture the specific aspects of smart contracts. Therefore, we need metrics designed for smart contracts.

Gas Estimation. In the Ethereum platform, any transaction that changes the internal state of a contract is charged a special resource called Gas [Eth18c]. The name was inspired by the view that this resource is the fuel for running contracts [Eth16]. Therefore, Gas units correspond to the execution of computation instructions. The idea is to avoid infinite loops (and the halting problem), encourage efficient code, and make users pay for the computation resource they used.

Even though the unused gas is refunded back to the user, optimize this resource is the key to have a transaction approved quickly. Since the Miner is awarded the used gas cost, he/she will prefer to mine transactions that optimize his/her gains. In this context, a better estimation of the gas by the user may encourage Miners to process such user's transaction.

Estimate how many gas units a contract function is going to require is not a simple task. The Remix IDE gives an estimate if the function does not contain loops or external calls. Consequently, Remix is not reliable to estimate gas on any function. Therefore, we require better estimation tools to assess the amount of gas consumption.

Security Recommendations. The number of smart contracts is growing at a fast rate, and developers are not yet accustomed to code in this new environment. Bad programming practices in a smart contract can turn into terrible security flaws [DMO+18]. For example, the infamous "DAO attack" was caused by insecure coding practices on the contract that allowed an attacker to drain approximately 50 million dollars worth of cyptocurreny [LCO+16, TVI+18]. As an analogy, we can think

of web applications that had SQL-injections flaws, which would be simple to avoid if developers designed and adopted a more secure coding practice.

Most security flaws in smart contracts can be traced to poor coding that could be avoided if the developer had a better expertise with the environment and the programming language. A simple and elegant way to give such expertise to developers is to create a recommendation system that warns developers of problematic code and suggests a better way to accomplish the same program. In this context, we claim it is important to give security recommendations for contract developers.

Dashboard. To make all the information gathered by future tools useful in practice, care has to be taken to create presentations that target both developers and even other stakeholders like management and customers. We plan to explore how Dashboards can support both viewing data and defining conditions that should automatically notify developers in case of problems [Few06].

1.6.2 Related work

In this section, we discuss related work on blockchain analysis platforms and tools.

Porru et al. [PPMT17] acknowledge the need to develop specialized tools and techniques for blockchain-oriented software (BOS). The authors define the term BOS as a software that interacts with blockchain. The authors describe issues and challenges faced when working with blockchain from a software engineering point-of-view: new professional roles, security and reliability, software architecture and metamodels, modeling languages, and metrics. They also analyze 1,184 GitHub projects using blockchain, and they propose new ideas and practices to improve the state-of-art on BOS. The authors present interesting research possibilities that inspired several components on the SmartAnvil platform.

Bartoletti et al. [BLPB17] propose a framework for blockchain analytics coded as a Scala library. Their framework works on both Ethereum and BitCoin platform and it employs a general-purpose abstraction layer to promote reuse. One great feature of the authors' framework is the ability to integrate data from other sources besides blockchain, such as a NoSQL database. The authors contrast their framework features against five other tools, but they do not conduct a performance comparison. This work is interesting because the authors combine blockchain data with a secondary database. Unlike SmartAnvil, their framework does not support querying or analysis of contracts, only blocks, and transactions. Moreover, the query language is a specific syntax based on a Scala API and it is not as popular as SQL-based queries. Finally, their framework focus only on gathering data and does not support other tools such as inspection or static analysis.

Kalodner et al. [KGC+17] implement an open-source blockchain analysis platform, called BlockSci. They designed their platform with a specific concern for performance by using an in-memory database and actual pointers instead of hash linkage. For instance, the authors' claim it is 15 to 600 times faster than other tools. They support the following blockchains: BitCoin, LiteCoin, Namecoin, and Zcash. The authors focus their analysis on financial transactions. Therefore their analysis is not well polished for blocks and it specifically does not support smart contracts. This contrast with SmartAnvil platform, which was designed to help smart contract developers as well as providing tools for other first-class elements (such as transactions and blocks).

Etherscan (*https://etherscan.io*) is also an analytics platform connected to the Ethereum platform and providing on-the-fly blockchain information on accounts, contracts, transactions, blocks, etc. Similarly to SmartAnvil, Etherscan also has an inspector and other tools to interact with Ethereum. Etherscan does not have a query language but it supports simple search on blockchain elements. Although Etherscan is free to use, it is not open source. Therefore, it is not possible to extend any Etherscan features or components. Unlike SmartAnvil which is open source to encourage developers to contribute to improving our tools.

Remix [Eth18b] is an open source suite of tools developed by the Ehtereum Foundation. Although Remix is mostly recognized for its IDE component, it does provide many other tools such as inspection, transaction debugger, gas estimation, etc. Since it is developed by the Ethereum Foundation, Remix is always integrated with the latest Ethereum APIs. Remix is coded in JavaScript and it is under the MIT open source license. SmartAnvil shares some similar tools with Remix (e.g., inspector, parser). However, both suites have distinguished tools that are absent on the other. For instance, SmartAnvil have a query language, a smart graph, and integration with Moose. On the other hand, Remix has a transactional debugger and a full IDE.

There are tools designed to improve the security of smart contracts. Oyente [LCO+16] is a tool that flags potential security flaws on smart contracts. Oyente uses static analysis on the contract binary code to make security recommendations. SmartCheck [TVI+18] is another tool using static analysis that identifies unsecured patterns on contracts source code and warns the user with better practices. Bhargavan et al. [BDLF+16] proposed an unnamed framework to convert smart contract to their own functional language F*, which was design to better verify the correctness and security of smart contracts. Although SmartAnvil does not support security verification or recommendation of smart contracts, this is important for smart contract developers and we plan to add a tool for such tasks in the future.

1.7 Conclusion

In this chapter, we presented a set of analysis tools showcasing our SmartAnvil platform to cover different aspects of smart contracts analysis. For each tool, we showed the importance of each, and the main challenges when developing them.

First, we presented the basic tool to support static code representation of Solidity smart contracts, our parser SmaCC-Solidity. By having full control over our parser and its representation allows us to avoid any problems by future changes on the Solidity language (which is not stable). The parser is used on other SmartAnvil tools such as SmartInspect, and SmartMetrics. Developers can use this parser to increase the integration between Solidity and Pharo by implementing more tool support; or build their own static analysis tools on the resulting AST.

Second, we showed SmartInspect, our internal state inspector for Solidity smart contracts. As we explained, contracts are opaque because once deployed it is very difficult to see their runtime state defined by their internal attributes. SmartInspect can pierce through the opaqueness and show to developers the live state of their own contracts while assuring a level of privacy that few other tools support. The inspector can see the contents of any contract instance of the given source code, without needing to redeploy, nor develop any ad-hoc code to fetch the information. Contract inspection can

aid developers to better understand their contracts and even find bugs more easily.

Third, we presented Ukulele, a query language that allows users to retrieve information from the blockchain by writing SQL-like queries. This tool automates the task of sequentially searching into generic opaque structures, and provides an easier way to specify the information we want to acquire in a higher abstraction level. Ukulele distinguishes itself to be able to search through all first class elements such as blocks, transactions, accounts, and contracts. Other tools usually cannot search contract instances, leaving potential useful information behind. On the other hand, Ukulele is even able to call contract functions inside its queries.

For future work, we plan to extend the platform with more tools such as: (i) improved metrics suite (our current SmartMetrics tools still needs improvement); (ii) gas estimation of contract functions; (iii) recommendation system to suggest secure programming practices; (iv) a dashboard to facilitate the usage and interaction with blockchain and smart contracts.

Bibliography

[ABC+13] Vanessa Peña Araya, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. Agile visualization with Roassal. In *Deep Into Pharo*, pages 209–239. Square Bracket Associates, September 2013.

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. In *Proceedings of International Conference on Principles of Security and Trust*, volume 10204, pages 164–186. Springer, 2017.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[BDLF+16] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, 2016. ACM.

[Ber16] Alexandre Bergel. *Agile Visualization*. LULU Press, 2016.

[BGLD17] J. Brant, T. Goubier, J. Lecerf, and S. Ducasse. Smacc: a compiler-compiler, 2017.

[Bit18] BitCoin.org. Bitcoin developer reference. bitcoin core apis, 2018. Bitcoin Project 2009-2018.

[BLGD17] John Brant, Jason Lecerf, Thierry Goubier, and Stéphane Ducasse. SmaCC, a Smalltalk Compiler-Compiler, 2017. http://www.refactory.com/Software/SmaCC/.

[BLPB17] Massimo Bartoletti, Stefano Lande, Livio Pompianu, and Andrea Bracciali. A general framework for blockchain analytics. In *1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, SERIAL '17, pages 7:1–7:6, New York, NY, USA, 2017. ACM.

[BMB96] Lionel C. Briand, Sandro Morasca, and Victor Basili. Property-based software engineering measurement. *Transactions on Software Engineering*, 22(1):68–86, 1996.

[BRDD18a] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse. Ethereum query language. In *1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 1–8, may 2018.

[BRDD18b] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse. Smartinspect: solidity smart contract inspector. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–18, mar 2018. Electronic ISBN: 978-1-5386-5986-1.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[CCI90] Elliot Chikofsky and James Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[CLLZ17] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *Proceedings of SANER*, pages 442–446. IEEE, 2017.

[CNSG15] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. The moldable inspector. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 44–60, New York, NY, USA, 2015. ACM.

[DMO+18] Giuseppe Destefanis, Michele Marchesi, Marco Ortu, Roberto Tonelli, Andrea Bracciali, and Robert M. Hierons. Smart contracts vulnerabilities: a call for blockchain software engineering? In *2018 International Workshop on Blockchain Oriented Software Engineering, IWBOSE@SANER 2018, Campobasso, Italy, March 20, 2018*, pages 19–25, 2018.

[EN10] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition, 2010.

[Eth14] Ethereum Foundation. Ethereum's white paper., 2014.

[Eth16] Ethereum Community. Ethereum homestead documentation, 2016.

[Eth18a] Ethereum Foundation. Json rpc, 2018.

[Eth18b] Ethereum Foundation. Remix documentation release 1., 2018. https://remix.readthedocs.io/en/latest.

[Eth18c] Ethereum Foundation. Solidity documentation release 0.4.24, 2018.

[Few06] S. Few. *Information Dashboard Design*. OReilly, 2006.

[Hev97] A.R. Hevner. Phase containment metrics for software quality improvement. *Information and Software Technology*, 39(13):867 – 877, 1997.

[HM95] M. Hitz and B. Montazeri. Measure coupling and cohesion in object-oriented systems. *Proceedings of International Symposium on Applied Corporate Computing (ISAAC '95)*, October 1995.

[IM15] Chinawat Isradisaikul and Andrew C. Myers. Finding counterexamples from parsing conflicts. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 555–564, New York, NY, USA, 2015. ACM.

[KGC+17] H. Kalodner, S. Goldfeder, A. Chator, M. Möser, and A. Narayanan. Blocksci: Design and applications of a blockchain analysis platform. *ArXiv e-prints*, sep 2017. «««< Updated upstream =======

[KLR13] Jan Kurš, Guillaume Larcheveque, and Lukas Renggli. PetitParser: Building modular parsers. In *Deep Into Pharo*, page 36. Square Bracket Associates, September 2013. »»»> Stashed changes

[LBGD18] Jason Lecerf, John Brant, Thierry Goubier, and Stéphane Ducasse. A reflexive and automated approach to syntactic pattern matching in code transformations. In *IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*, Madrid, Spain, September 2018.

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS'2016 (ACM Conference on Computer and Communications Security)*, 2016.

[LH93] W. Li and S. Henry. Object oriented metrics that predict maintainability. *Journal of System Software*, 23(2):111–122, 1993.

[LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[LKG07] Adrian Lienhard, Adrian Kuhn, and Orla Greevy. Rapid prototyping of visualizations using mondrian. In *Proceedings IEEE International Workshop on Visualizing Software for Understanding*, Vissoft'07, pages 67–70, Los Alamitos, CA, USA, June 2007. IEEE Computer Society.

[Mer93] Gary Merrill. Parsing non-lr(k) grammars with yacc. *Software Practice and Experience*, 2(8):829–850, 1993.

[Mil05] Ashley Mills. Antlr tutorial, 2005.

[MM18] S. Morishima and H. Matsutani. Accelerating blockchain search of full nodes using gpus. In *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 244–248, mar 2018.

[NA05] Oscar Nierstrasz and Franz Achermann. Separating concerns with first-class namespaces. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit, editors, *Aspect-Oriented Software Development*, pages 243–259. Addison-Wesley, 2005.

[PBB07] Leonardo Teixeira Passos, Mariza A. S. Bigonha, and Roberto S. Bigonha. A methodology for removing lalr(k) conflicts. *Journal of Universal Computer Science*, 13(6):737–752, jun 2007.

[PBD+15] Nick Papoulias, N. Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Mercury: Properties and design of a remote debugging solution using reflection. *Journal of Object Technology*, 2015.

[PPMT17] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli. Blockchain-oriented software engineering: challenges and new directions. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 169–171, 2017.

[RDDL17] Henrique Rocha, Stéphane Ducasse, Marcus Denker, and Jason Lecerf. Solidity parsing using smacc: Challenges and irregularities. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*, IWST '17, pages 2:1–2:9, New York, NY, USA, 2017. ACM.

[RSI+17] Michele Ruta, Floriano Scioscia, Saverio Ieva, Giovanna Capurso, and Eugenio Di Sciascio. Supply chain object discovery with semantic-enhanced blockchain. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, pages 60:1–60:2, New York, NY, USA, 2017. ACM.

[SKS11] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 6th edition, 2011.

[SM16] Guido Salvaneschi and Mira Mezini. Debugging reactive programming with reactive inspector. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 728–730, New York, NY, USA, 2016. ACM.

[SR14] Venkatesh Srinivasan and Thomas Reps. *Recovery of Class Hierarchies and Composition Relationships from Machine Code*, pages 61–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[TDMO18] R. Tonelli, G. Destefanis, M. Marchesi, and M. Ortu. Smart contracts software metrics: a first study. *ArXiv e-prints*, feb 2018.

[TNM08] Ewan Tempero, James Noble, and Hayden Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 667–691, Berlin, Heidelberg, 2008. Springer-Verlag.

[TVI+18] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16, may 2018.