The Journal of Systems & Software 168 (2020) 110647

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

In practice

A federated society of bots for smart contract testing

Emanuele Viglianisi[a], Mariano Ceccato[b,*], Paolo Tonella[c]

[a] Fondazione Bruno Kessler, Trento, Italy
[b] University of Verona, Verona, Italy
[c] Università della Svizzera Italiana (USI), Lugano, Switzerland

ARTICLE INFO

Article history:
Received 24 October 2019
Received in revised form 22 April 2020
Accepted 14 May 2020
Available online 16 May 2020

Keywords:
Software testing
Blockchain
Smart contracts

ABSTRACT

Smart contracts are a new type of software that allows its users to perform irreversible transactions on a distributed persistent data storage called the blockchain. The nature of such contracts and the technical details of the blockchain architecture give raise to new kinds of faults, which require specific test behaviours to be exposed. In this paper we present SoCRATES, a generic and extensible framework to test smart contracts running in a blockchain. The key properties of SoCRATES are: (1) it comprises bots that interact with the blockchain according to a set of composable behaviours; (2) it can instantiate a society of bots, which can trigger faults due to multi-user interactions that are impossible to expose with a single bot. Our experimental results show that SoCRATES can expose known faults and detect previously unknown faults in contracts currently published in the Ethereum blockchain. They also show that a society of bots is often more effective than a single bot in fault exposure.

1. Introduction

A recent extension to crypto currencies (e.g., Ethereum) consists of the so-called "smart contracts". *Smart contracts* are programmes stored in the blockchain whose execution is guaranteed by the distributed network of miners. The computation model of smart contracts is quite peculiar and innovative. In fact, once written to the distributed blockchain, a smart contract and all its transactions are immutable, even in case programming defects are later identified, which means that incorrect computations are frozen forever in the blockchain. Thus, thorough and deep testing of smart contracts is crucial, in order to detect programming errors before erroneous transactions are permanently stored in the blockchain.

A contract, by definition, mediates the interaction among multiple users, who play different roles in the contract. For example, Fig. 1 shows a common interaction scenario for a contract that handles tokens: through the contract, the user *spender* authorizes an *initiator* to transfer a certain amount of tokens t on her/his behalf. The *initiator* transfers the tokens from the *spender* account to a *receiver* account. More details on this scenario can be found in Section 3.

To test this scenario, one user is not enough. Three distinct actors shall interact coherently with the contract under test, according to the intended protocol, i.e., a *spender*, who authorizes the *initiator* to move some value to the *receiver*.

Based on the observation that smart contracts are multi-role programmes, we propose SoCRATES (Smart ContRActs TESting), a novel and extensible testing framework, based on a federated society of interacting bots. Each bot impersonates a distinct user (or role) in the contract. Depending on the contract, different roles might exist in the contract code itself or might emerge from the different permissions (e.g., contract-owner, token-owner) assigned to different users. Then, a simulator iteratively assigns an execution slot to each federated bot, with the aim of spotting programming defects hidden in the potentially complex and articulated interactions supported by the contract when multiple players are involved. For instance, SoCRATES may deploy three or more bots to test the smart contract in Listing 1, including a *spender* bot, an *initiator* bot and a *receiver* bot. Each bot instantiates a specific *behaviour* or a combination of predefined behaviours, which allow the bot to generate input values for the submitted transactions according to a set of configurable strategies. SoCRATES includes four predefined behaviours (Random, Boundary, Overflow, Combined) and is extensible with domain specific behaviours that can implement specific strategies of interaction with the contract under test. SoCRATES supports oracle specification taking the form of contract invariants. SoCRATES comes with one predefined generic invariant and five predefined EIP20 specific invariants that can be enabled for the contract under test. SoCRATES can be easily extended with contract specific invariants, as demonstrated in the experiments performed

[*] Corresponding author.
E-mail addresses: eviglianisi@fbk.eu (E. Viglianisi), mariano.ceccato@univr.it (M. Ceccato), paolo.tonella@usi.ch (P. Tonella).

https://doi.org/10.1016/j.jss.2020.110647
0164-1212/© 2020 Elsevier Inc. All rights reserved.

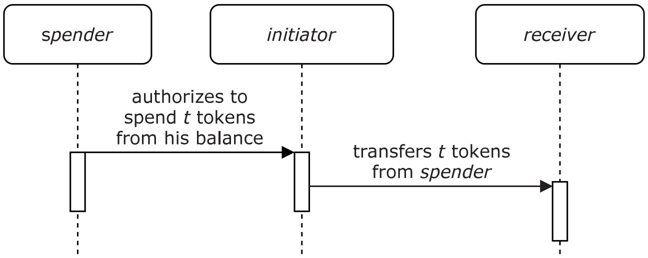
spender
initiator
receiver
authorizes to spend *t* tokens from his balance
transfers *t* tokens from *spender*

Fig. 1. Example of interaction among three distinct users.

on real Ethereum contracts, where six additional contract specific invariants have been added.

Most existing tools for smart contract testing (Grech, 2018; Jiang et al., 2018; Luu et al., 2016) are focused on security issues (e.g., reentrancy vulnerabilities) rather than the functional invariants (e.g., each successful token transfer should log a *Transfer* event) that contracts are supposed to ensure. Correspondingly, they can detect only violations to security properties associated with known vulnerabilities. No general, extensible framework exists for functional testing of smart contracts and the most related tool, Echidna,[1] is based on a completely different approach – namely, contract fuzzing – that is purely random and not based on the composition of a set of (extensible) behaviours. Moreover, Echidna does not include pre-defined invariants to classify the outcome of execution and to detect whether a defect has been exposed.

Our empirical validation shows that SoCRATES is effective in detecting defects in real smart contracts that are actively used in the blockchain and that perform transactions associated with real monetary value. Once executed on 1905 real smart contracts retrieved from Etherscan, SoCRATES reported 148 true invariant violations and only 32 false alarms. We compared SoCRATES with the state of the art smart contract fuzzer Echidna and found that SoCRATES identifies substantially more true invariant violations than those reported by Echidna.

Our work makes the following contributions to the state of the art:

- A novel framework for smart contract testing, based on a number of innovative ideas, such as:
    - a federation of bots that interact autonomously with the contract under test;
    - an extensible and configurable set of bot behaviours;
    - an extensible and configurable set of contract invariants.
- An empirical study conducted on real contracts retrieved from the Ethereum blockchain, including:
    - invariant violations identified by our tool on a data set of real contracts;
    - an empirical comparison with the tool Echidna.

The paper is structured as follows. Section 2 covers some background about blockchain and smart contracts, with a specific focus on the Ethereum platform. Section 3 presents a motivating example, used in Section 4 to explain our technical solution, which is empirically validated in Section 5. Section 6 compares SoCRATES to related work and Section 7 concludes the paper.

2. Background

The first cryptocurrency proposal, *b-money*, was born in the last years of the 20th century. However, the idea of an alternative and distributed currency gained considerable interest in the years of the global financial crisis. In 2009, the very first decentralized cryptocurrency named *Bitcoin* was released. It was developed from the ideas of a person (or group of people) under the alias of *Satoshi Nakamoto*. Bitcoin was possible because of a new consensus protocol based on a distributed data structure called *blockchain*.

2.1. Blockchain and consensus protocol

The blockchain is a data structure which maintains a list of transactions and a mapping between addresses and account states. An account state includes the persistent information associated with an address, such as the amount of cryptocurrency that the address owns. Transactions are records of payments between users. A transaction updates the blockchain from state *S1* to *S2*. The blockchain is implemented as a linked list (*chain*) of elements called blocks replicated among all the participants of the peer-to-peer network. A block is an information package containing a reference to the previous block (hash), a timestamp, a nonce and a list of newly validated transactions.

The consensus protocol, called *Nakamoto Consensus*, guarantees the immutability of the chain. To modify the blockchain, indeed, all the nodes must agree on the sequence of blocks and on the validity of all their transactions. At a high level, to add a new block to the chain, a node called *miner* creates a proposed block by choosing, among those available, a limited sequence of new transactions. Then, each miner attempts to solve the *proof-of-work* puzzle. The first miner who solves the proof-of-work broadcasts its proposed block to all the other nodes in the network. Upon receiving the new proposed block, a node checks its integrity and validity. If the block is valid, the node adds the new block to its local copy of the blockchain. At the end of this process, the creator of the added block (*successful miner*) receives a compensation.

Compensation is one of the core parts of the process and it represents the motivation for the miners involved in the mining process. When a new block is validated, the successful miner receives a compensation equal to the sum of the total transaction fees included in the validated block, plus a newly created amount of cryptocurrency. A transaction fee is an extra amount of cryptocurrency decided and paid by the transaction sender to compensate the miner. A transaction with a high fee is more likely to be included by the miners in the next blocks and to be quickly validated.

2.2. Ethereum

Ethereum is a platform built on top of the concepts behind the *Nakamoto Consensus* protocol and blockchain. Published in *2014* by *Vitalik Buterin*, it gained a lot of interest because of the Turing-Complete *smart contracts* that it supports. The cryptocurrency associated with the Ethereum platform is called Ether. Ethereum extends some basic concepts of the blockchain. Ethereum distinguishes between two types of accounts: *externally owned accounts*, and *contract accounts*. Both types of account have an associated Ether balance. An externally controlled account is controlled by the private key of a user. Such key can be used to sign and submit new transactions. A contract account is instead controlled by its code, a set of instructions stored in the blockchain. A contract account has also an internal storage, stored in the blockchain, to save additional persistent information. Whenever

[1] Echidna https://github.com/trailofbits/echidna.

a contract receives a transaction, the contract code is executed by all the miners in the network, which in this way performs a distributed validation of the contract execution. Since the language used to code contracts is Turing-complete, termination cannot be decided statically. To avoid that malicious users can block the network with endless executions, Ethereum manages resources of the network using a *Gas System*. The *Gas system* is the fee system of Ethereum. Each instruction is associated with a certain amount of *gas*, that must be paid in order for the network to execute such instruction. To invoke a contract, together with the transaction payload (data), the user has also to provide the amount of *gas* necessary for the correct execution of the contract code. Moreover, the sender can set an arbitrary *gasPrice* per unit of *gas*. The expression *gas * gasPrice* gives the amount of Ether spent as transaction fee, to be paid to the miner for the execution of each instruction of the contract code.

2.3. EIP20 tokens

Developed in 2015, *ERC20*[2] *(aka EIP20)* is a standard defining the interface that an Ethereum Smart contract has to implement for the management of custom tokens. A *custom token* is a newly created currency, living within the Ethereum ecosystem. The interface, which documents the basic functionality that an EIP20 token has to implement, contains the following contract functions:

1. **name** (optional): returns the name of the token;
2. **decimals** (optional): returns the number of decimals the token uses;
3. **symbol** (optional): returns the symbol of the token;
4. **totalSupply**: returns the total supply of emitted tokens;
5. **balanceOf**: returns the amount of tokens owned by a specified address;
6. **transfer**: transfers an amount of tokens from a sender address to a destination address;
7. **approve**: approves another address to spend a specific amount of tokens from the sender balance;
8. **allowance**: returns the amount of tokens that spender is allowed to withdraw from owner;
9. **transferFrom**: transfers an amount of tokens from an origin address to a destination address.

EIP20 tokens gained a lot of popularity because they are commonly used as assets for running ICOs (Initial Coin Offerings). Their popularity, importance, and the simple interface associated with them make EIP20 contracts the ideal use case for our experimentation.

3. Motivating example

In this section, we present the intuition and the motivation behind our test case generation framework. We introduce the society of bots and the set of invariants that can be used as testing oracles. The section revolves around the example in Listing 1, a buggy contract with an overflow fault.

Function *buggedTransferFrom* in Listing 1 is an implementation of the EIP20 function *transferFrom*. The aim of this function is to transfer an amount of tokens, passed in as parameter *_value*, from the address *_from* to the address *_to*. In detail, the function accesses two local variables defined within the contract, namely, the maps *balances* (line 26) and *allowed* (line 27). These two maps are used in the EIP20 interface to store, respectively, the tokens held in the balance of each user and the amount of tokens each user allowed other addresses to spend from her user balance using function *approve* (lines 20 to 24). Lines 9–10 are *require* statements that enforce preconditions on input values. For instance, line 9 checks that the amount of tokens approved by *_from* for transfer by *msg.sender* is greater than the amount *_value* to be transferred.

balance	
spender	1,000
receiver	0

allowed		
spender	initiator	0

$$spender \rightarrow approve(initiatior, 100)$$

balance	
spender	1,000
receiver	0

allowed		
spender	initiator	100

$$initiator \rightarrow buggedTransferFrom(spender, receiver, 100)$$

balance	
spender	900
receiver	100

allowed		
spender	initiator	0

Fig. 2. Nominal execution scenario of buggedTransferFrom.

balance	
spender	1,000
receiver	0

allowed		
spender	initiator	0

$$spender \rightarrow approve(initiatior, 2000)$$

balance	
spender	1,000
receiver	0

allowed		
spender	initiator	2,000

$$initiator \rightarrow buggedTransferFrom(spender, receiver, 2000)$$

balance	
spender	$2^{256} - 1,000$
receiver	2000

allowed		
spender	initiator	0

Fig. 3. Attack scenario of buggedTransferFrom.

Lines 12 to 14 are a sequence of operations applied to state variables to actually transfer the tokens from one account to the other. The same amount *_value* is removed from the address *_from* and is added to the balance of the address *_to*. Moreover, the amount of tokens approved by *_from* for transfers performed by *msg.sender* is decreased.

The nominal contract interaction is shown in Fig. 2 as the status of the contract and its changes when user messages are processed. Initially, *spender* has a `balance` of 1000 tokens and the *initiator* is not allowed to move them, because its `allowed` is set to 0. With the first message, the balance of the *spender* does not change, but the *initiator* is granted the right to move 100 tokens from the *spender*'s `balance`. This right is reflected as a change in the `allowed` variable.

With the second message, the *initiator* exploits this allowance and moves 100 tokens from the *spender*'s `balance` to the *receiver*'s `balance`. Moreover, the `allowed` of the *initiator* is reset to 0.

This code contains a programming defect and the error scenario, shown in Fig. 3, reveals this defect. Before executing this

[2] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

scenario, the contract state is in the same initial condition as in the nominal case. The *spender* calls the *approve* function to grant the *initiator* the possibility to spend 2000 tokens from its own account, by sending the first message.

The function *approve* has no check on fund availability and, therefore, the call succeeds even if the amount of tokens to approve exceeds current spender's balance.

At this point, the `allowed` variable value ensures that the *initiator* can spend 2000 tokens from the *spender*. In the next step, the *initiator* sends the second message to move 2000 tokens from the *spender*'s balance using the function *buggedTransferFrom*.

The function *buggedTransferFrom* has no check on the on actual availability of spender's money. It only asserts that the *initiator* has been granted the right to move that amount of tokens from `spender`. The missing check causes the expression at line 12 to compute a negative number that underflows when represented as an unsigned integer. This sets the `spender`'s balance to a huge amount of tokens: $2^{256} - 1000$. Moreover, lines 13 and 14 are correctly executed, causing the *initiator*'s balance to be 1000 and the approved amount of tokens in `allowed` to be reset to 0.

This scenario shows how a contract state that was initially consistent, with a total supply of tokens of 1000, can be turned into an inconsistent state, where new tokens are created (2^{256}) exploiting a bug in a function that is supposed to transfer existing tokens.

```
1  contract SampleToken {
2
3    function buggedTransferFrom(
4        address _from,
5        address _to,
6        uint256 _value
7    )
8      public
9      returns (bool)
10   {
11     require(_value <= allowed[_from][msg.sender
           ]);
12     require(msg.sender!=_from && _from!=_to);
13
14     balances[_from] -= _value;
15     balances[_to] += _value;
16     allowed[_from][msg.sender] -= _value;
17
18     emit Transfer(_from, _to, _value);
19     return true;
20   }
21
22   function approve(address spender, uint _value
         ) public returns (bool) {
23     allowed[msg.sender][spender] = _value;
24     Approval(msg.sender, spender, tokens);
25     return true;
26   }
27
28   function deposit() public payable {
29     /* not reported for brevity */
30   }
31
32   mapping(address => uint256) balances;
33   mapping(address => mapping (address =>
         uint256)) allowed;
34
35 }
```

Listing 1: Example of smart contract function with an overflow defect.

Society of bots: The execution of a contract function is strictly state dependent. For instance, the result of the execution of *buggedTransferFrom* depends on the state variables *balances* and *allowed*. Similarly to object-oriented testing (Fraser and Arcuri, 2013a; Tonella, 2004), our goal is to test the contract functions in as many contract states as possible.

Table 1
Invariants: *Txs* is the set of performed transactions.

ID	Type	Rule
I1	General	$\nexists t \in Txs : successful(t) \wedge overflow(t)$
I2	EIP20	$\sum_{a \in accounts} balanceOf(a) == totalSupply$
I3	EIP20	$\forall t \in Txs : t = transferFrom \wedge successful(t) \implies t.amount <= allowance_{from,sender}$
I4	EIP20	$\forall t \in Txs : t = transferFrom \wedge successful(t) \implies allowance' = allowance - t.amount$
I5	EIP20	$\forall t \in Txs : t \in \{transferFrom, transfer\} \wedge successful(t) \implies Transfer \in events(t)$
I6	EIP20	$\forall t \in Txs : t = approve \wedge successful(t) \implies Approval \in events(t)$

However, testing a smart contract using a single bot limits the smart contract states that we can reach and test. Indeed, many of the smart contract states are only reachable through the cooperation and interaction of different actors.

In the example in Listing 1, to let the *sender* spend *_value* tokens from the account *_from*, *_from* has to explicitly approve the *sender* calling the EIP20 function *approve*. So, in this example, in order to test the *buggedTransferFrom* function, it is necessary to have at least two bots, to play the two different roles: the first bot pre-authorizes and the second bot spends the tokens. In this case two bots are needed, otherwise this defect cannot be exposed.

Overflow: One of the most subtle causes of contract misbehaviour is the overflow of numeric expressions. An expression overflow, indeed, raises no exception and it is up to the programmer to add proper constraints and revert the contract state in case of overflow. Although it is a good practice to use a library called *SafeMath*[3] to perform safe math operations, many contracts exist that do not use such safe library.

Function *buggedTransferFrom* does not check that the account *_from* has the amount of tokens that the*_sender* wants to transfer. Moreover, if *_from* tries to send a huge amount of money, possibly greater than its balance, the expression at line 12 may overflow, leading to an inconsistent/invalid contract state.

4. Test case generation

4.1. Framework

We developed SoCRATES a highly configurable framework for test case generation, which allows testing a smart contract by simulating the interaction of multiple, different bots that operate on it.

The framework is written in Typescript[4] and adopts an object oriented design that supports its configurability and extendability. Each component of the framework is developed as a Typescript class and is compiled in a separate *Node.JS* module. The framework itself is composed of a set of modules which are used to configure the main module, *Simulator*.

The class diagram in Fig. 4 shows the relations between the main classes under the control of the *Simulator*: *Account*, *Bot*, *Behaviour* and *Contract*.

Account: The *Account* class models the state of an Ethereum account. The basic account state is composed of two fields: the account *address* and its *balance* in Ethers. It is up to the test engineer to extend this class including extra fields to fit the contract under test. For instance, if we are testing an EIP20 contract, it could be useful to also store the amount of tokens owned by the account as part of the account state.

[3] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol.

[4] https://www.typescriptlang.org/.

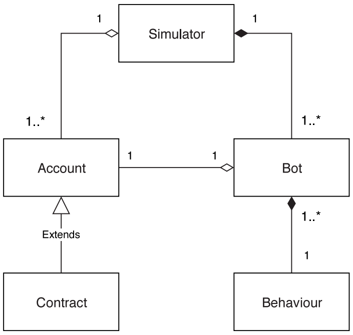

Fig. 4. SoCRATES class diagram.

Contract: The *Contract* class represents an Ethereum smart contract, which is the target of the test generator. In Ethereum, a contract is a special kind of account, characterized by an executable code. The tester can extend also the *Contract* class by adding contract-specific fields (e.g., *totalSupply* for an EIP20 contract). Moreover, the *Contract* class contains a list of the contract public functions (i.e., its Application Binary Interface, *ABI*) and must implement method *sendTransaction* to let SoCRATES interact with the deployed contract instance in the blockchain. The interaction of a contract with the Ethereum blockchain is handled using the library*Web3.JS*.[5]

Bots: A *Bot* is an object associated with one and only one account. A bot is in charge of two tasks: 1. deciding which of the contract functions to call and which parameter values to pass in; and, 2. sending the transaction to the contract under test from its own account. Deciding the function to call and the associated formal parameters is a complex task. In fact, there can be different strategies, each with its pros and cons. In SoCRATES, a bot implements its own strategy by instantiating a *Behaviour*. When the bot is called by the framework to decide what contract function to call, it activates the decision process implemented in its behaviour.

Behaviours: Class *Behaviour* contains an abstract method named *performAction*, whose implementation within subclasses of *Behaviour* define a strategy for choosing which function to call and which parameters to pass in, in order to perform a new bot action. We implemented a preliminary set of four different behaviours that are described in Section 4.2.

Simulator: Class *Simulator* is the core execution controller of the framework. When the simulator starts a new simulation, it takes in input a list of accounts, an instance of the contract under test, and a society of bots. The simulator activates also a configurable set of *invariants*, which must hold true during the entire simulation. The goal of the simulation is indeed to test the contract in different states and to check whether, through the interaction among bots, the contract reaches an invalid state in which an invariant is violated.

The simulation process is described in Fig. 5. It consists of a main loop where, at each step, the simulator lets each bot perform an action and, after each action, it calls method *checkInvariants* to check whether any of the invariants is violated. The simulation ends when an invariant is violated or when it is stopped (e.g., manually or after a fixed amount of time).

We designed the simulator to be ready for use within the Ethereum ecosystem. For such a reason, the simulator runs on top of the *Truffle suite*,[6] in turn composed of *Truffle Test* and *Ganache*. *Truffle Test* is a testing framework that provides developers with tools to easily compile, deploy and test a smart contract. Among other features, the Truffle environment includes *Web3.JS* and other JavaScript libraries useful for unit testing and assertion definition. *Ganache* is a lightweight implementation of the Ethereum blockchain designed for testing purpose. In *Ganache*, part of the blockchain implementation is simulated to avoid the typical delay of blockchain distributed validation. So, upon being sent, a new transaction is immediately validated and included in a new block.

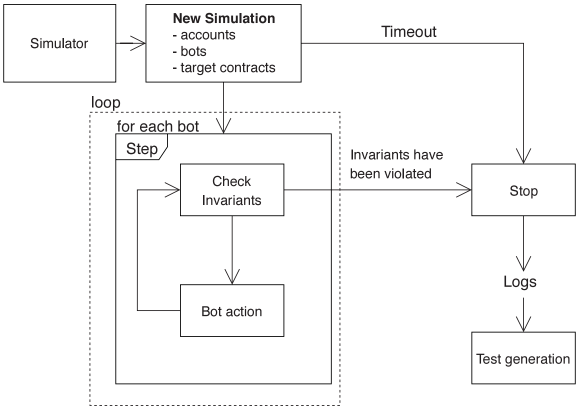

Fig. 5. Workflow of the simulation process.

The key requirement of the *Simulator* is a *Web3.JS* instance connected to a running *Ganache blockchain*. Then, a new simulation can be instantiated: the *Simulator* deploys a new instance of the contract under test, the associated invariants, the bots and their behaviours. The *Web3.JS* instance is used during the entire simulation to interact with the deployed contract, when bot transactions are submitted to *Ganache*. Invariants are defined as *Chai*[7] assertions, checked after processing every submitted transaction.

Test case generation: During the entire simulation process, logs are generated to record each successful transaction. A transaction record includes the involved accounts and the ABI functions invoked, with the associated parameter values. The test case generator takes in input such simulation logs and outputs a test case reproducing the steps that lead to the final state of the simulation. When the simulation terminates with the violation of an invariant, the final state is invalid and correspondingly the test case is a failing one. The output test cases are *Truffle* test cases that do not depend on the configuration of the simulator but only on the number of *Ganache* accounts used during the simulation.

Invariants: To recognize an inconsistent contract state, we define generic and contract-specific invariants, acting as the test oracles for the contract under test. An invariant is a condition which should always hold true in every contract state. If the contract ends up in a state where one of the invariants is violated, this means that the contract has an implementation error. We define a state where an invariant is violated as an *invalid state*. We have identified a preliminary set of six invariants, applicable generically to any contract (I1) or specifically to contracts of a given type (I2, I3, I4, I5, I6). They are reported in Table 1.

- **I1** is a general invariant. Its condition states that there must not exist a successful transaction whose execution causes an overflow.
- **I2** is an EIP20 specific invariant. Its condition states that the sum of the tokens owned by each account (*balanceOf(a)*) must be equal to the total amount of token supply (*totalSupply*).

[5] https://github.com/ethereum/web3.js/.

[6] https://truffleframework.com.

[7] http://www.chaijs.com/.

- **I3** is an EIP20 specific invariant, expressed in terms and *post* conditions on the function *transferFrom*. For a *transferFrom* to be successful, the amount of token to be transferred ($t.amount$) has to be less than or equal to the amount of tokens that the transaction sender is still allowed to spend on behalf of *from* (*allowance[from][msg.sender]*).
- **I4** is an EIP20 specific invariant, expressed in terms of *post* conditions on the function *transferFrom*. After each successful *transferFrom*, the amount of tokens the transaction sender can spend from the *from* address must be decreased by the amount of token transferred ($t.amount$).
- **I5** is an EIP20 specific invariant, expressed in terms of *post* conditions. Both *transfer* and *transferFrom* transactions must emit, if successful, a *Transfer* event consistent with the actual function parameters and the interface specification.
- **I6** is an EIP20 specific invariant, expressed in terms of *post* conditions on the function *approve*. Similarly to the invariant *I5*, a successful transaction *approve* must emit a consistent *Approval* event.

In the case of function *buggedTransferFrom*, invariants I1, I2, I3, I4, I5 are applicable. Although these invariants are expected to have quite general validity, there might be exceptions. Indeed, there could be contracts that legitimately violate these invariants due to their specific business rules. In these cases, contract developers should configure SoCRATES and should manually disable those invariants that are not relevant or not compatible with the specific contract under test.

4.2. *Behaviours*

In this section, we describe the preliminary set of behaviours implemented in SoCRATES. It is a best practice in well-written contracts to check, at the start of each Solidity function, that the function's preconditions hold true, by using one or more *require* statements. If a precondition fails, a transaction failure is reported and the transaction is aborted. Inputs that result in a transaction failure are not useful, because they do not modify the state of the contracts in the blockchain, and they should be avoided, to improve the efficiency of the testing process. For this reason, the behaviours described below support several configurable heuristics aimed at minimizing the possibility of transaction failure due to precondition violation.

RandomBehaviour: The contract function to call and the associated parameter values are chosen in a random fashion. Since a typical Solidity variable type has a huge range of possible values, using *RandomBehaviour* without any constraint will generally result in many transaction failures when the contract function checks its inputs. To reduce the number of failures, the test engineer can limit the range in which random values are chosen by manually specifying some constraints. For instance, the transferred amount of tokens in the EIP20 function *transfer* could be chosen within a limited range of values, such as [0, *balance*+1000], so as to decrease the probability of a transaction failure due to an insufficient amount of tokens.

Listing 2 shows the code fragment that implements RandomBehaviour for parameter value generation. A switch-case statement is used to generate a value of the correct type. A random value is chosen among the possible values in the allowed interval. For instance, line 5 returns a random integer among the values that can be represented with a given number of bits.

```
 1  // ...
 2   switch (paramType) {
 3      case 'int':
 4          bits = get_bits_number(paramType)
 5          return random_int(min_value(bits),
                max_value(bits))
 6      case 'address':
 7          return random(accounts);
 8      case 'bool':
 9          return Math.random() >= 0.5;
10      case 'string':
11          return generate_random_string(
                min_length, max_length)
12      default:
13          throw new Error("Unhandled Solidity
                Type");
14  }
15  // ...
```

Listing 2: Code implementing the RandomBehaviour

BoundaryBehaviour: Parameter values are randomly chosen near to the boundaries of the parameter type. E.g., for type *uint256* (256-bit unsigned int), this behaviour picks a value with uniform probability in the intervals $[0, 999] \cup [2^{256} - 1000, 2^{256} - 1]$.

An implementation of the logic behind Boundary Behaviour parameter generation is shown in Listing 3. Function *getBoundaryBigNumber* chooses randomly a value close to the two boundaries *min* and *max*. E.g., if *choice* is equal to 2, it returns an integer value with uniform probability between *min* and *min+delta* (line 6), where *delta* is a pre-defined offset value).

```
 1  // ...
 2   protected getBoundaryBigNumber(min: IBigNumber
        , max: IBigNumber) : any {
 3      choice = RandomValueGenerator.getRandomInt
            (0, 10);
 4      if (choice == 0) return min;
 5      if (choice == 1) return max;
 6      if (choice == 2) return random(min, min.
            plus(this._delta));
 7      if (choice == 3) return random(max.minus(
            this._delta), max);
 8      return random(0, this._delta);
 9   }
10   // ...
```

Listing 3: Code implementing the BoundaryBehaviour

OverflowBehaviour: Input values are selected to pass the function preconditions, while at the same time overflowing an inner expression in the contract function (more details about this behaviour are provided in Section 4.3).

CombinedBehaviour: This behaviour randomly chooses between two or more predefined, atomic behaviours (as the three ones described above). The random choice can be configured by the tester. By default it applies equal probability to all combined behaviours. For instance, *Boundary* + *Random* and *Complete* (i.e., *Boundary* + *Random* + *Overflow*) are examples of combined behaviours that have been evaluated empirically.

```
function batchTransfer(address[] _receivers,
    uint256 _value) public returns (bool) {
    uint cnt = _receivers.length;
    uint256 amount = uint256(cnt) * _value;
    require(cnt > 0 && cnt <= 20);
    require(_value > 0 && balances[msg.sender]
        >= amount);

    balances[msg.sender] = balances[msg.sender
        ].sub(amount);
    for (uint i = 0; i < cnt; i++) {
        balances[_receivers[i]] = balances[
            _receivers[i]].add(_value);
        Transfer(msg.sender, _receivers[i],
            _value);
    }
    return true;
}
```

Listing 4: Example of smart contract function with an overflow defect.

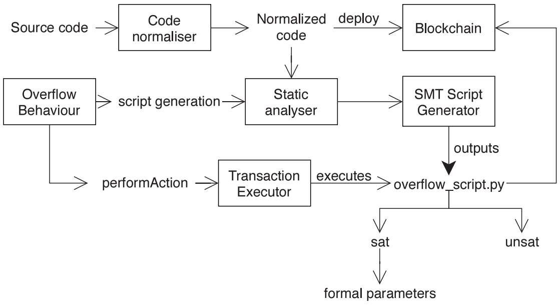

Fig. 6. Workflow of the overflow behaviour.

4.3. Overflow behaviour

The overflow behaviour is substantially more complex than the random and the boundary behaviour, hence requiring a more detailed presentation.

Let us consider as an example the contract function *batchTransfer*.[8] This function sends the same amount *_value* to all the addresses in *_receivers*. Variable *amount* at line 3 holds the total amount of tokens that the sender wants to transfer. The multiplication at line 3, however, could result in an overflow. Even if it might be quite likely that values generated using a *Combined(Random + Boundary)* may result in an overflow at line 3, it is unlikely that the execution passes the *require* conditions at lines 4 and 5. Indeed, without any additional constraints, the list *_receivers* could be of any arbitrary length between 0 and $2^{256} - 1$ and the value of variable *amount* after the overflow could be even greater than the balance of the sender. For this reason, we have introduced a specialized *OverflowBehaviour*, which employs the workflow described in Fig. 6 to generate input values that at the same time overflow some contract expression and respect the contract function's preconditions. The main components of the workflow are: 1. *Static analyser*; 2. *SMT script generator*; 3. *Code normalizer*; 4. *Transaction executor*.

Static analyser: The source code is parsed to obtain its *Abstract Syntax Tree* (AST). The AST is visited to collect, for each function in the contract, all the expressions which could overflow and all the constraints defined by *require* conditions. An expression is considered a candidate for overflow if it contains one of the following operators: $[+, +=, ++, -, -=, --, *, **, *=]$.

SMT script generator: It aims at creating a script for each Solidity function which, using an *satisfiability modulo theories (SMT)* Solver, computes concrete input values that satisfy all the constraints corresponding to the function preconditions and result in an overflow. In particular, the generator uses the result from the static analyser to define a set of symbolic variables and to introduce a set of constraints that must be satisfied to pass the preconditions and to produce an overflow.

The SMT solver constraints for function *batchTransfer* (see Listing 4) are shown in Listing 5 as (simplified) SMT script code.

```
1  // Returns overflow conditions for unsigned int
2  def is_overflow(value, numberOfBits):
3      isAnOverflow = value > (2**numberOfBits) -1
4      isAnUnderflow = value < 0
5      return Or(isAnOverflow, isAnUnderflow)
6
7  // inputs
8  cnt == input._receivers.length
9  amount == cnt * input._value
10
11 // force overflow condition on "amount"
12 is_overflow(amount, 256) == True
13
14 // requires
15 cnt > 0 AND cnt <= 20
16 _value > 0 AND
17 state.balances[sender] > amount
```

Listing 5: Z3 constraints for function *batchTransfer*, trying to overflow the expression at line 3 in Listing 4

The utility function at line 2 (*is_overflow*) checks that the value passed as first parameter is a case of overflow when represented with the number of bits specified in the second parameter. This function is used later at line 12, to impose that variable *amount* overflows its 256 bits.

The conditions at lines 8 and 9 are used to define symbolic variables *cnt* and *amount*, corresponding to assignments at lines 2 and 3 of Listing 4.

Lines 15 and 16 represent the constraints imposed on input values by the *require* preconditions of the smart contract.

It is important to note that the behaviour of a function in a Solidity contract may depend on the global state of the blockchain (e.g., *balances*). Since handling the global state and its possible changes symbolically is not affordable by static code analysis, we adopt the same heuristic strategy commonly used in dynamic symbolic execution and concolic testing (Godefroid et al., 2005; Sen et al., 2005): constraints for the SMT solver include concrete, rather than symbolic, values for contract state variables. The concrete values embedded in the constraints are the values actually observed at run time.

Our implementation of the SMT script generator automatically creates a python script which invokes Z3[9] as SMT solver.

The constrains in Listing 5 are satisfiable and Z3 is actually able to compute values for parameters *_receivers* and *_value*, such that all the *require* statements pass and the multiplication at line 3 overflows (see Listing 4). Because of the overflow, variable *amount* is assigned a small value at line 3. This will cause the *sender* to spend a small amount of tokens (line 7), lower than the sum of tokens transferred to the *receivers* (loop at line 8). Hence, the overflow (violation of invariant I1) will also make the total amount of owned tokens exceed the declared total supply (violation of invariant I2).

Code normalizer: The SMT script generator needs to access the state of the contract to read concrete values of state variables that are embedded in the constraints. To let the generator fully inspect the contract state, the code normalizer changes the visibility of contract variables and methods to public. The normalized code is then deployed to the blockchain.

Transaction executor: The *OverflowBehaviour* starts by creating a set of SMT scripts, one for each expression that may give raise to an overflow. Then, when *performAction* is invoked by our framework, it executes one of the generated scripts. If the SMT script can compute input values that cause overflow, *OverflowBehaviour* invokes the contract function using these input parameters. Otherwise, this behaviour tries another, randomly selected, SMT script.

A successful transaction (i.e., one not rejected by the contract), in which one of the contract's expressions overflows, represents a violation of invariant *I1*. Hence, a fault in the contract is found and SoCRATES emits a test case that documents and reproduces the exposed defect.

[8] https://www.peckshield.com/2018/04/22/batchOverflow/.

[9] https://github.com/Z3Prover/z3.

5. Experimental validation

We assessed our framework SoCRATES by investigating the following research questions:

- **RQ**$_1$ How effective is SoCRATES in detecting invariant violations?
- **RQ**$_2$ How does a society of bots compare to a single bot in detecting invariant violations?
- **RQ**$_3$ What is the effectiveness of each atomic and combined bot behaviour, in terms of number of invariant violations that bots can detect?
- **RQ**$_4$ How does SoCRATES compare with Echidna, in terms of number of invariant violations?
- **RQ**$_5$ How effective is SoCRATES in detecting contract-specific invariants?

The first research question **RQ**$_1$ is intended to investigate if SoCRATES is indeed capable of detecting real invariant violations in smart contracts, generating a reproducible test case for each violation.

The second research question **RQ**$_2$ checks whether a society of bots is more appropriate to test smart contracts than a single bot alone. This research question is meant to validate our initial assumption, which motivated the definition of a multi-bot framework.

The third research questions **RQ**$_3$ is meant to analyse the role of the different atomic and combined bot behaviours in detecting invariant violations.

The fourth research question **RQ**$_4$ compares SoCRATES with the state of the art tool Echidna, on the respective ability of detecting invariant violations and generating test cases.

Finally, the last research question **RQ**$_5$ assesses the validity of our approach beyond generic and EIP20 invariants, by considering contract-specific assertions.

5.1. Subjects

The empirical validation considers tokens based on the EIP20 interface because (at the time of our experimentation) this was one of the most prominent interfaces for tokens managed by Ethereum smart contracts. We applied SoCRATES to two groups of real smart contracts: (1) recent contracts based on the EIP20 interface; (2) top contracts, i.e., contracts with highest market capitalization.

The first set of smart contracts considered in our empirical validation was selected from those actually active in the blockchain. To sample them, we relied on *Etherscan*,[10] a web portal to inspect the content of the Ethereum blockchain. At a specific page,[11] this site lists the 10,000 most recent valid transactions for smart contracts that define tokens according to the EIP20 interface. For each transaction, it also reports the contract and sender address. We considered the EIP20 interface in particular, because it is an emerging standard being increasingly adopted by smart contracts whenever a custom currency is needed. This query was performed for contracts traded on October 23rd, 2018. The collected 10,000 recent Ethereum transactions are associated with 1625 unique smart contracts.

The second set of smart contracts comprises all the smart contracts listed in the *TOP* smart contracts list, as reported by *Etherscan*. The list contains 887 smart contracts ranked by the token market capitalization value provided by the website *CoinMarketCap*.[12]

The blockchain only stores the compiled byte-code and the interface of such smart contracts. However, it is a relatively common practice for contract owners to also publish the source code, in order to make their smart contracts more transparent and trustful. So, we restricted our selection to smart contracts for which we could fetch the source code. This gave us a sample of 1059 recently used smart contracts that implement the interface EIP20 and 846 TOP contracts; all of them published the source code to make it accessible to users.

5.2. Competitor tools

We considered eight alternative tools (namely, Manticore, Echidna, Oyente, Mythril, Madmax, Zeus, Securify, Maian) described in the literature, which automatically analyse and generate test cases for Ethereum smart contracts. In general, most of these tools are preliminary research prototypes, not enough mature/stable to be used in a large experiment like the one conducted in this paper.

Manticore[13] is a symbolic execution tool, which can generate input values that trigger errors on binary executables and on Solidity smart contracts. However, its support to smart contracts is quite partial and limited. Despite it explicitly declares to offer a monitor to detect overflow errors, we were able to run such monitor only on simple contracts. As a result of our preliminary assessment of Manticore, we submitted a number of issues to the Github project hosting the tool and found issues confirming our problems. Among them, issue #1375,[14] #1374,[15] #1362,[16] #1322.[17]

```
function transfer(address _to, uint256 _value) public returns (bool success) {
    require (_to != 0x0 && _value > 0);

    /* START INSTRUMENTATION */
    transfer_from = msg.sender;
    transfer_to = _to;
    pre_balance_1 = balanceOf[msg.sender];
    pre_balance_2 = balanceOf[_to];
    declared_transferred_amount = _value;
    /* END INSTRUMENTATION */

    if (admins[msg.sender] == true && admins[_to] == true) {
        balanceOf[_to] = balanceOf[_to].add(_value);
        totalSupply = totalSupply.add(_value);
        emit Transfer(msg.sender, _to, _value);

        /* START INSTRUMENTATION */
        event_from = msg.sender;
        event_to = _to;
        event_amount = _value;
        post_balance_1 = balanceOf[msg.sender];
        post_balance_2 = balanceOf[_to];
        /* END INSTRUMENTATION */

        return true;
    }
    ...
}
```

Listing 6: Example of instrumentation needed by Echidna

Echidna[18] is a fuzzing tool that generates random input values for smart contracts that run on the Ethereum virtual machine. It

[10] https://etherscan.io/.

[11] https://etherscan.io/tokentxns.

[12] https://coinmarketcap.com/.

[13] Manticore https://github.com/trailofbits/manticore.

[14] https://github.com/trailofbits/manticore/issues/1375.

[15] https://github.com/trailofbits/manticore/issues/1374.

[16] https://github.com/trailofbits/manticore/issues/1362.

[17] https://github.com/trailofbits/manticore/issues/1322.

[18] Echidna https://github.com/trailofbits/echidna.

can be configured to monitor the contract under test for invariant violation. Whenever fuzzing succeeds in violating an invariant, the sequence of calls with the error inducting input values represents a valuable test case.

Echidna uses a fuzzing algorithm to select the functions to call and to generate their input parameters in a random fashion. The same algorithm is responsible for every action and, unlike SoCRATES, cannot be extended with different behaviours based on a user defined business logic.

Echidna makes the assumption that invariants are implemented directly in Solidity, as part of the contract source code. Because of this design choice, it is impossible to test a smart contract against our invariants without any code editing. In order to compare Echidna with SoCRATES, we need a way to let Echidna support all the invariants in Table 1. To this aim, we manually edited the code of the contracts and instrumented it, promoting selected local variables to contract variables, in order to make such local variables visible to Echidna and useable within Echidna invariants.

For example, the code snippet in Listing 7 shows an implementation of the invariant I5, based on the code instrumentation shown in Listing 6. Invariant I5 checks that every *transfer* emits a consistent *Transfer* event. To do that, we need to save the balances of the addresses before and after the transactions into auxiliary variables (respectively, lines 7–8 and lines 21–22 in Listing 6).

```
// INVARIANT
function echidna_invariant_I5() public returns (bool) {
    if (event_amount != 0) {
        bool consistent_from = event_from == transfer_from;
        bool consistent_to = event_to == transfer_to;
        bool consistent_amount = event_amount == declared_transferred_amount;
        bool consistent_post_balance_to = post_balance_2 == (pre_balance_2 + event_amount);
        bool consistent_post_balance_from = post_balance_1 == (pre_balance_1 - event_amount);
        return consistent_from && consistent_to && consistent_amount && consistent_post_balance_to && consistent_post_balance_from;
    }
    return true;
}
```

Listing 7: Example of Echidna invariant implementation

Other potential alternative tools have been evaluated, but discarded because they are not compatible with our experimental setting that requires the ability to handle real-world smart contracts and to define business logic specific invariants:

Oyente (Luu et al., 2016) is based upon symbolic execution techniques and targets *Mishandled exceptions*, *Transaction-ordering dependence*, *Timestamp dependence* and *Reentrancy* vulnerabilities. The tool generates symbolic constraints from the bytecode in input and outputs the problematic paths containing one of the targets vulnerabilities to the user. However, Oyente does not provide any test case to replicate the vulnerability and this makes a comparison with our tool impossible.

Mythril[19] is based on a symbolic virtual machine that runs Ethereum smart contracts. However, it does not emit concrete test cases to trigger the vulnerabilities, but it only reports the list of discovered vulnerabilities.

Table 2
Experiment configurations.

Experiment	Number of bots	Behaviours
RQ_1	10	Random+Boundary+Overflow
RQ_2	1	Random+Boundary+Overflow
RQ_3	10	Random+Boundary+Overflow, Random+Boundary, Random, Boundary, Overflow
RQ_4	10	Random+Boundary+Overflow
RQ_5	10	Random+Boundary+Overflow

Madmax (Grech, 2018) is a static analysis tool for smart contracts. As such, it does not generate test cases.

Other tools such as *Zeus, Securify, Maian* use symbolic execution to target a limited set of contract vulnerabilities. The survey by Praitheeshan et al. (2019) reports an extended description and comparison of these tools.

In the end, we have decided to evaluate SoCRATES against Echidna, the most mature competitor tool.

5.3. Experimental settings

We ran SoCRATES on top of *Ganache* using its default configuration with 10 active accounts with unlimited ether balance. *Ganache* simulates a completely fresh and empty blockchain environment, with no transaction. Starting from an empty blockchain helps in delivering testing results that are easily interpretable, because tests make no assumption on initial data and produce consistent results across re-execution. Hence, contracts that make assumptions on certain initial data or transactions to be available in the blockchain are currently not compatible with our experimental settings.

Since our test case generation framework includes non-deterministic components, each experiment comprises 10 restarts of the simulator and of the blockchain status. Each simulator has been configured with a maximum of 1000 simulation steps and a timeout of 5 min. This timeout might look too short when compared to the amount of time needed by other tools (Jiang et al., 2018) to fuzz contracts. The difference is that other approaches deploy contracts in a fully functional blockchain, which runs a heavyweight consensus algorithm on each transaction. Instead, we deploy contracts in the testing environment of *Ganache*, which avoids the overhead due to the consensus network, irrelevant for contract testing.

Different experiments have been run with a different number of bots and with different behaviours. Table 2 summarizes the configurations of SoCRATES that we used to set up the experiments for the different research questions.

To speedup the experiment on the large number of subject contracts, it has been split into multiple jobs, each job running on a distinct core of an Intel(R) Xeon(R) CPU E5420 @ 2.50 GHz processor, and each job being assigned 4 GB of RAM.

The complete replication package is available online (Viglianisi et al., 2019), including the contracts used as subjects for our experiments, results, SoCRATES and instructions on how to run it.

5.4. RQ_1 invariant violation detection

SoCRATES was configured with a distinct bot for each account. By default, each bot is configured to contain the full combination of all available behaviours, which in the current version of the implementation means it includes *RandomBehaviour*, *BoundaryBehaviour* and *OverflowBehaviour*. The bot switches randomly among them.

[19] https://github.com/ConsenSys/mythril.

Table 3
Assertion violations detected by SoCRATES.

	I1		I2		I3		I4		I5		I6		TOTAL	
Data set	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP
Recent	0	10	2	61	2	1	3	1	1	2	1	20	9	95
Top	1	7	2	33	3	1	6	0	0	1	1	11	13	53
TOTAL	1	17	4	94	5	2	9	1	1	3	2	31	32	148

Violations detected by SoCRATES have been manually reviewed, to classify them as actual programming errors (TP, true positives) or false alarms (FP, false positives). This classification has been performed manually by the first author of the paper, by comparing the behaviour implemented in the contract with the expected behaviour described in the official documentation of the EIP20 interface.[20]

Empirical results are reported in Table 3, showing how many true positives (TP) and false positives (FP) have been reported by the tool for each invariant, in the first line for the *recent* smart contracts and in the second line for the *top* smart contracts. Results report a total of 95 vulnerable *recent* smart contracts and 53 vulnerable *top* smart contracts.

Invariant I1 (overflow) is violated in 10 contracts in the first data set and in 8 contracts in the second data set, with only one false positive, in the latter data set. Invariant I2 (inconsistent total supply) is the most frequently violated. SoCRATES detected respectively 61 and 33 true positives on the two data sets, with only 2 false positives per data set. Invariant I3 (wrong allowance check) is detected correctly in 2 cases (one per data set), but SoCRATES reported 5 false positives. Invariant I4 (wrong allowance update) is violated in both data sets, with just 1 true positive. Invariant I5 (`Transfer` event) is violated 4 times with only 1 false positive. Finally, invariant I6 (`Approval` event) is violated 33 times, with only 2 false positives (one for each data set.)

Contracts with programming errors often have a significant capitalization. Considering only those with a true positive violation detected by SoCRATES, the average capitalization[21] of defective smart contracts in the first and in the second data set was, respectively, $14M and $71M.

It is worth noting that the most frequently violated invariant is I2. Although in general an invariant violation can be associated with a programming mistake, a violation of invariant I2 may also be regarded as a design choice that makes the contract not fully compliant with the EIP20 interface. Indeed, since part of the EIP20 interface is documented in natural language, there are ambiguities that can lead to different interpretations. For example, we interpret the meaning of variable *totalSupply* as the total amount of tokens that are transferable among accounts. According to this interpretation, SoCRATES classifies as violations those cases where not all tokens in *totalSupply* are distributed, or when there are *frozen* (not transferable) tokens that are still counted in the variable *totalSupply*. Discrepancies might be due to different design choices or interpretations of the EIP20 interface.

The high false positive rate for invariants I3 and I4 is due to contracts that specialize the EIP20 interface in a way that is not documented in the interface itself. For instance, contracts that impose a fee on transactions or that grant some operations only when they come from addresses in a white-list. In these cases, the generic invariant does not apply and the contract tester should use a custom invariant. This custom contract behaviour should be known to a developer, who, consequently, could define and use an appropriate, contract-specific invariant.

Based on these results, we can answer the first research question as follows:

> SoCRATES *was able to identify 148 true invariant violations in 1905 real smart contracts, that have been recently traded in the Ethereum Blockchain or are listed as top token contracts.*

5.5. RQ_2 society of bots vs. single bot

```
function distribute(address[] addresses, uint256 _value) onlyOwner canDistr public{
    for (uint i=0; i < addresses.length; i++) {
        balances[owner] -= _value;
        balances[addresses[i]] += _value;
        emit Transfer(owner, addresses[i], _value);
    }
}
```

Listing 8: Example of uniquely identified violation by 10 bots configuration.

We then replicated the same experiment executed for RQ_1, but instead of deploying a society of bots, we deployed a single bot to test each smart contract. This single bot was still equipped with the default behaviour, which combines all available behaviours (*RandomBehaviour*, *BoundaryBehaviour* and *OverflowBehaviour*). To limit the time needed to run all the many experiments, we limited the investigation to the first data set, i.e. to the recently traded contracts.

Experimental results are shown in Table 4. For each invariant we contrast the results of the full framework with 10 bots (first line) with the variant with only a single bot (second line). For each invariant, the table reports false positives (FP), true positives (TP) and the violations reported by one configuration and not by the other one (column Uni).

For invariant I1 the two configurations are similar, 10 true positives each and 1 uniquely identified violation. Instead, for I2, I3 and I4, the full configuration is clearly preferable, because it reports more true positives and more uniquely identified violations than the limited configuration with only 1 bot, although at the cost of a few more false positives. For I5 the configuration with 10 bots scores a higher number of true positives (2 vs. 1) with the same number of false positive (i.e., 1). For invariant I6, 20 true violations are detected by both configurations, with the full configuration reporting a false positive (vs. no false positive with the single bot).

In total, the configuration with 10 bots reported 95 true positives and 9 false positives, while the configuration with one bot reported 90 true positives and 3 false positives. Moreover, the first configuration detected 10 violations that the second configuration could not detect. On the other hand, the second configuration detected only 4 violations that the first one could not detect.

Overall, the configuration with 10 bots outperformed the configuration with 1 bot. However, it should be noticed that having more bots limits the number of transactions sent by each address, in favour of the number of transactions sent by the society as a whole. Hence, the number of transactions sent by specifically relevant addresses, e.g., for many invariant violations, the contract owner, is less than the transactions that a single bot can send, assuming the single bot impersonates the relevant address. All the uniquely identified violations detected by the configuration with only *1 bot* are, indeed, invariant violations caused by function calls that only the contract owner can execute. on the contrary, the violations detected by the 10 bot configuration

[20] https://eips.ethereum.org/EIPS/eip-20.

[21] Capitalization reported by Etherscan on March 2019.

Table 4
Comparison of assertion violations detected by SoCRATES, either with 10 bots or with 1 bot.

	I1			I2			I3			I4			I5			I6			SUM		
Bots	FP	TP	Uni	FP	TP	Uni	FP	TP	Uni	FP	TP	Uni	FP	TP	Uni	FP	TP	Uni	FP	TP	Uni
10	0	10	1	2	61	6	2	1	1	3	1	1	1	2	1	1	20	0	9	95	10
1	0	10	1	1	59	3	0	0	0	1	0	0	1	1	0	0	20	0	3	90	4

contain several cases in which multiple bots are needed. An example is the vulnerability in Listing 8. Function *distribute* is used by the contract owner to transfer *_value* tokens from its balance to each of the addresses in the *addresses* list. A single bot configuration makes the caller bot act in isolation, calling function *distribute* with an array containing *n* repetitions of its own address. Since the *from* address and the *destination* address are the same, the expression in line 3 will not result in any underflow. On the contrary, in the 10 bots configuration the caller bot is aware of the blockchain context and correspondingly it calls function *distribute* with an array of different bot addresses. Since there are no checks on the total amount of tokens to be transferred, a long list of addresses may cause the expression in line 3 to underflow, resulting in the unwanted creation of new tokens.

Based on these results, we can formulate the following answer to the second research question:

> *The bot society is more effective than a single bot in detecting smart contract invariant violations. In fact, the society could detect more violations and more true positives. Moreover, it could detect 10 invariant violations that a single bot could not detect.*

5.6. RQ_3 effect of bot behaviours

In the third experiment the full bot society is deployed on the first data set, similarly to the other experiments, but with different behaviours. The experiment is replicated to compare these behaviours configurations:

- **Complete**: All the three behaviours: *RandomBehaviour*, *BoundaryBehaviour* and *OverflowBehaviour*;
- **Boundary+Random**: the composition of the two behaviours that randomly perform uniform/biased sampling of the input domain, *RandomBehaviour* and *BoundaryBehaviour*;
- **Overflow**: *OverflowBehaviour* alone;
- **Boundary**: *BoundaryBehaviour* alone;
- **Random**: *RandomBehaviour* alone.

Since these configurations include non-deterministic behaviours, each of them was run 10 times.

The experimental results are shown in Table 5. For each set of behaviours (first column), the table reports how many invariant violations were detected in the 10 runs. The number of violations reported in the table is the sum of the violations observed across the 10 runs. However, in case the same violation is observed multiple times it is counted only once. Violations are reported in distinct columns for the different invariants. Invariant violations have been manually validated and classified into true positive violations (TP) and false positives (FP). Moreover, the table also reports how many true positive invariant violations have been detected uniquely by a configuration and by none of the other configurations (Uni).

The *Complete* behaviour could detect the largest number of violations (i.e., 10) of invariant I1. The *Overflow* behaviour could detect less (i.e., 9) violations, while the other behaviours could detect no violation of I1.

Considering invariant I2, the *Boundary* behaviour detected the largest number of violations. However, the *Complete* behaviour scored the largest number of uniquely identified violations.

On invariants I3 and I4, *Boundary* and *Boundary+Random* behaviours reported the highest number of true positives, while the *Complete* behaviour scored second. Considering invariants I5 and I6, all but the *Overflow* behaviour reported a very similar amount of true positives, but the *Complete* behaviour had the highest number of uniquely identified violations.

In total, as shown in Table 5, rightmost columns, the *Complete* behaviour could detect the largest number of true positives with 95 violations, followed by *Boundary* that detected 90 real violations and *Random + Boundary* with 89 violations.

From these results, we can state that the *Overflow* behaviour is effective only in detecting a small quantity of violations, compared to all the other behaviours. The implementation of the *Overflow* behaviour, indeed, makes a bot perform an action only if it is able to pass all the initial *require* statements and to make a subsequent arithmetic expression overflow. For this reason, the *Overflow* behaviour is very accurate, having no false positive. However, executing many valid transactions, without causing overflow, is needed to evolve the contract state and explore new states, which eventually lead to an invariant violation. For this reason, random behaviours (overall) report a higher number of true positives than the *Overflow* behaviour.

We can thus answer our third research question as follows:

> *The combination of all bot behaviours is more effective than atomic behaviours alone. The full combination could detect 95 invariant violations. The composition of Random + Boundary is also quite effective. Indeed, it could find 11 violations that none of the other behaviours could detect.*

5.7. RQ_4 comparison with other tools

In the fourth experiment, we use Echidna to test smart contracts and violate invariants. To that aim, we had to manually instrument the contract code and we had to manually specify our invariants in the format expected by Echidna. Since instrumentation was manual and very time consuming, we applied it on a small subset of contracts. In particular, we randomly sampled 10 contracts among those for which SoCRATES detected violations. We took 3 contracts with I1 violations, 3 contracts with I2 violations. Then, we took 1 contract among those with violations of I3, I4, I5 and I6. We, then, ran Echidna on each instrumented contract for the same amount of time given to SoCRATES in RQ_1, so as to make a fair comparison between the two tools.

Experimental results are reported in Table 6. Echidna could generate test cases that violate 6 out of 10 invariants detected by SoCRATES. While all the three violations of I1 were detected also by Echidna, I2 could be violated only in one case. No violation of I3 and I4 was reported by Echidna, while the two violations of I5 and I6 could be detected also by Echidna.

It should be noticed that whenever fuzzing caused arithmetic overflow (violations of invariant I1), Echidna alone would not have reported any problem without our instrumentation, because Echidna does not include any mechanism to assert the occurrence of overflow. It was only thanks to our manual instrumentation of the contract code, which is not part of Echidna, that overflow became detectable by Echidna.

Echidna could not generate test cases that violate invariants I3 and I4 because, using a single sender address, it was not able

Table 5
Assertion violations detected by different behaviour configurations.

	I1			I2			I3			I4			I5			I6			TOTAL		
	FP	TP	Uni	FP	TP	Uni	FP	TP	Uni	FP	TP	Uni	FP	TP	Uni	FP	TP	Uni	FP	TP	Uni
Complete	0	**10**	**1**	2	61	**3**	2	1	0	3	1	0	1	2	0	1	20	**2**	9	**95**	6
Bound.+Rand.	0	0	0	2	63	2	3	**2**	0	11	**2**	0	1	2	0	1	20	1	18	89	**3**
Overflow	0	9	0	**0**	44	2	**0**	0	0	**0**	0	0	**0**	0	0	**0**	0	0	**0**	53	2
Boundary	0	0	0	2	**64**	2	3	**2**	0	13	2	0	1	**2**	0	1	20	0	20	90	2
Random	0	0	0	**0**	61	2	**0**	0	0	2	0	0	**0**	1	0	1	20	0	3	82	2

Table 6
Assertion violated by test cases generated by Echidna.

ExpressCoin	I1	Violated
AEToken	I1	Violated
DNCEQuity	I1	Violated
DELTAToken	I2	Violated
Yumerium	I2	Not violated
Tube	I2	Not violated
JAAGCoin	I3	Not violated
MKC	I4	Not violated
CoinfairCoin	I5	Violated
Bible	I6	Violated

Table 7
Contract specific invariants.

ID	Rule
CSI1	$\forall t \in Txs : successful(t) \wedge owner \rightarrow owner' \implies owner == t.msg.sender$
CSI2	$\forall t \in Txs : successful(t) \implies totalSupply' >= totalSupply$
CSI3	$\sum_{a \in accounts} balanceOf(a) <= tokenLimit$
CSI4	$\forall t \in Txs : t = enableTokenTransfer \wedge successful(t) \implies t.msg.sender == walletAddress$
CSI5	$totalAllocated <= (ADVISORS + FOUNDERS + HOLDERS + RESERVE)$
CSI6	$\forall t \in Txs : t = getToken \wedge successful(t) \implies owner == t.msg.sender$

to test function *transferFrom*, which uses the concept of token allowance between at least two addresses.

We can thus answer the last research question as follows:

> SoCRATES *outperformed the state-of-the-art tool Echidna, for automated testing of Ethereum smart contracts. Moreover, Echidna required substantial manual intervention to support the invariants of* SoCRATES.

5.8. RQ$_5$ contract specific invariants

So far, we considered only generic invariants, most of which descending from the specification of the EIP20 token interface. However, smart contracts might implement additional features, beyond those required by a general interface. For instance, some contracts support the possibility to increase the total number of available tokens (total supply), by minting of new tokens. Other contracts support limited minting, by setting a hard limit of total supply that cannot be exceeded. These additional features come with additional constraints, that can be expressed as invariants that should be always satisfied by the execution of a correct implementation.

In this last experiment, we extended the list of invariants by defining 6 new contract-specific invariants, targeting 6 properties that are expected to hold and were observed in some of the contracts that we manually inspected when classifying the violations reported by SoCRATES as true/false positive.

Table 7 lists the six new contract-specific invariants the we could identify. They are:

- **CSI1** states that only the *owner* address can submit a transaction that changes the current *owner* address;
- **CSI2** states that, after every transaction, the value of variable *totalSupply* may increase or remain the same, but never decrease;
- **CSI3** states that the sum of the tokens owned by each account (*balanceOf(a)*) must be less than or equal to the value of variable *tokenLimit*;
- according to **CSI4**, an *enableTokenTransfer* is successful only if the transaction's sender address is equal to the *walletAddress*;
- **CSI5** states that the sum of the total allocated tokens must never exceed the sum of variables *ADVISORS, FOUNDERS, HOLDERS, RESERVE*;
- according to **CSI6**, only the owner address can execute function *getToken* successfully.

These invariants are not supposed to hold for *all* contracts. Depending on the contracts' business logic, the developer, who knows the functional requirements of the contract under test, is supposed to decide which of these contract-specific invariants to enable (if any) or to define new invariants that matter.

SoCRATES could detect violations of all these 6 contract-specific invariants in the analysed contracts. Thus, we can answer the last research question as follows:

> SoCRATES *is effective in detecting violations of contract-specific invariants. In fact, it could generate test cases that expose violations of all the 6 contract-specific invariants that we defined for our subject contracts.*

5.9. Threats to validity

Internal validity: The current implementation of SoCRATES comes with some limitations. While random and boundary behaviours support all the features of the Solidity language, the overflow behaviour is able to extract constraints for a subset of the language. In particular, the static analysis it uses is intra-procedural. Moreover, the static analyser handles only *if* statements that can be translated to *require* statements, such as those that control a *fail* or *exit* statement. Loops possibly followed by *fail* or *exit* statements are not supported (this limitation is due to the use of symbolic execution to derive overflow constraints). Despite these limitations of the static analyser, the overflow behaviour remains largely applicable to contracts, since symbolic modelling of the execution is needed only for the preconditions and for the expression that is candidate for overflow. Preconditions are usually not preceded by complex control flow (loops in particular) and many expression candidates can usually be found that are not preceded by unhandled instructions. The effectiveness of the overflow behaviour observed in the empirical validation shows that it could be actually applied in several interesting cases.

Contract normalization (first step in Fig. 6) also does not support all language features. For instance, function visibility change is based on regular expressions that may fail in certain contracts.

In Ethereum, the smart contract constructor is a special function used to instantiate the contract in the blockchain. Normally, constructors have no formal parameter. However, sometimes, constructors require parameters to configure a contract on-the-fly at instantiation-time. Automatic generation of constructor parameter values is at the moment not supported by SoCRATES.

Construct validity: The EIP20 interface was subject to evolution and finalized in late 2017. Some of the contracts considered in our experiments could have been developed following old versions of this interface and invariant violations might not be due to implementation defects but just to mismatches with the most recent version of the interface. To limit this problem, we considered only smart contracts with recent transactions in the blockchain.

External validity: Although we sampled more than a thousand contracts for our experimental validation of SoCRATES, we cannot assume that our results hold for any other arbitrary contract, especially when contract specific invariants are involved. Further experiments on additional case studies are needed to corroborate our findings.

6. Related work

State of the art techniques in automated test case generation do not address the specific challenges of smart contracts.

Automated test generators consider the system under test as decomposed into units to be tested in isolation (Fraser and Arcuri, 2013b; McMinn, 2004; Panichella et al., 2015) or as a whole system accessible for testing purposes through its interface (Gross et al., 2012; Nguyen et al., 2014). However, faults that affect the business logic of a contract cannot be exposed at the unit (Solidity function) level, which rules out most adequacy criteria based on coverage of structural elements of individual code units (e.g., statement/branch coverage). At the same time, testing through the interface requires the interacting test entity to simulate the behaviour of a contract user, which goes beyond coverage of the public interface functions defined in Solidity. Moreover, a single simulated user might be insufficient for faults that can be exposed only when multiple contract users interact with each other.

Search based test case generation can accommodate various kinds of fitness functions that drive the test generation process. Traditionally, structural coverage has been regarded as the key objective of test generation (Fraser and Arcuri, 2013b; McMinn, 2004; Panichella et al., 2015), but there are recent works that consider alternative testing objectives/fitness functions (e.g., energy consumption Bruce et al., 2015, performance Langdon and Harman, 2015; Yoo et al., 2013, quality of service Bozkurt and Harman, 2012). However, no existing work considered a fitness function that can provide guidance to a society of bots interacting with a smart contract. Such an application of search based testing can be achieved in our framework, by defining a proper search based behaviour for bots. The development of such behaviour and of the associated fitness function is part of our planned future work.

Multi agent systems can be designed to exhibit collective behaviour, so as to provide the desired services to the user (Jacyno et al., 2009). Cooperative agents can be engineered to solve a dynamic, distributed problem (Picard et al., 2005), or to work in parallel on different sub tasks (Panait and Luke, 2005). Our design of a bot society that can effectively test a smart contract was largely inspired by these works, with the ultimate goal of ensuring that the bot interactions converge toward the exposure of as many contract defects as possible. In addition to that, the techniques for agent testing have also influenced our choices.

In **agent testing** (Ekinci et al., 2008; Nguyen et al., 2012; Padgham et al., 2013), a model of the world where agents operate is instantiated and evolved in order to identify limit conditions and configurations that expose functional or non-functional deviations from the expected behaviour. While we also instantiate the world where contracts operate (i.e., the blockchain), our focus is more on the use of a society of autonomous bots to test a smart contract, rather than on the agent society as the target of testing activities. A common idea between the previous work on agent testing and our framework is that *goals* (and correspondingly, behaviours) should drive the testing strategy (Ekinci et al., 2008).

Another related area is that of **self-adaptive systems**. The full behaviour of self-adaptive systems emerges only at runtime, when specific execution contexts are instantiated. Hence, in addition to standard pre-release verification and validation, quality assurance for self adaptive systems is moved to the runtime and relies on monitoring (Tamura et al., 2010, 2013). Surveys on runtime monitoring for self-adaptive systems were conducted by Rabiser et al. (2017) and Vierhauser et al. (2016). A recent attempt to relate dependability constraints to a dynamically changing execution context was implemented in the GODA framework (Mendonça et al., 2016). In this framework, during runtime monitoring, depending on the specific execution context, proper behavioural constraints are checked. Another similar approach for context-dependent adaptation of the monitoring rules is MORPED (Contreras and Mahbub, 2014). These works differ substantially from our approach since their goal is to recognize at runtime the testing execution contexts that have the dynamically changing features required to expose faults in a self-adaptive system, while our goal is to test a smart contract and its evolving state in a (simulated) environment. However, similarly to our society of testing bots, their monitors may also exhibit autonomous and possibly adaptive behaviours.

Existing works in **security testing of smart contracts** (Grech, 2018; Jiang et al., 2018; Luu et al., 2016) are focused on closed catalogues of known security vulnerabilities (Atzei et al., 2017), such as transaction order dependency, timestamp dependency, mishandled exceptions and reentrancy (Luu et al., 2016); gasless send, exception disorder, reentrancy, timestamp dependency, block number dependency, dangerous delegate call and freezing Ether (Jiang et al., 2018); or out of gas vulnerabilities (Grech, 2018). To the best of our knowledge, no general, extensible framework has ever been proposed to support the detection of arbitrary invariant violations – not restricted to specific security vulnerabilities – in the wild, with no assumption on the availability of a database of known vulnerabilities, and taking advantage of a society of autonomous bots.

7. Conclusion

Once deployed in the blockchain, smart contracts are immutable and so are the programming mistakes in their implementation. Hence, thorough testing of smart contracts is crucial to spot implementation defects before deployment.

SoCRATES is a testing framework for smart contracts based on a federated society of bots, designed to simulate the complex interactions of multiple users, often involved with different roles in a contract. Experimental validation shows that our approach is effective in detecting implementation errors that violate contract invariants, even in smart contracts actually used in production, associated with real monetary value. Results also show the importance of instantiating a society rather than single bots and the key role of combined behaviours, as opposed to simple, atomic bot behaviours.

In our future work, we plan to extend our framework by (i) covering more invariants not yet supported by SoCRATES and (ii) integrating more behaviours to improve the testing effectiveness of SoCRATES.

SoCRATES has been released as open source (Viglianisi et al., 2019), together with the replication package that includes the contracts used as subjects for our experiments, results and instructions.

CRediT authorship contribution statement

Emanuele Viglianisi: Conceptualization, Methodology, Software, Validation, Writing - review & editing. **Mariano Ceccato:** Conceptualization, Methodology, Writing - review & editing, Supervision. **Paolo Tonella:** Conceptualization, Methodology, Writing - review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Atzei, N., Bartoletti, M., Cimoli, T., 2017. A survey of attacks on Ethereum smart contracts (SoK). In: Proceedings of the 6th International Conference on the Principles of Security and Trust (POST), Held As Part of the European Joint Conferences on Theory and Practice of Software, (ETAPS), pp. 164–186.

Bozkurt, M., Harman, M., 2012. Optimised realistic test input generation using web services. In: Search Based Software Engineering - 4th International Symposium, SSBSE. pp. 105–120.

Bruce, B.R., Petke, J., Harman, M., 2015. Reducing energy consumption using genetic improvement. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, pp. 1327–1334.

Contreras, A.R., Mahbub, K., 2014. MORPED: Monitor rules for proactive error detection based on run-time and historical data. In: Proceedings of the Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT), pp. 28–35.

Ekinci, E.E., Tiryaki, A.M., Çetin, Ö., Dikenelli, O., 2008. Goal-oriented agent testing revisited. In: Agent-Oriented Software Engineering IX, 9th International Workshop, AOSE. pp. 173–186.

Fraser, G., Arcuri, A., 2013a. Whole test suite generation. IEEE Trans. Softw. Eng. 39 (2), 276–291.

Fraser, G., Arcuri, A., 2013b. Whole test suite generation. IEEE Trans. Softw. Eng. 39 (2), 276–291.

Godefroid, P., Klarlund, N., Sen, K., 2005. DART: directed automated random testing. In: Sarkar, V., Hall, M.W. (Eds.), Programming Language Design and Implementation (PLDI 2005). ACM, pp. 213–223.

Grech, N., 2018. MadMax: Surviving out-of-gas conditions in Ethereum smart contracts. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).

Gross, F., Fraser, G., Zeller, A., 2012. Search-based system testing: high coverage, no false alarms. In: International Symposium on Software Testing and Analysis, ISSTA. pp. 67–77.

Jacyno, M., Bullock, S., Luck, M., Payne, T.R., 2009. Emergent service provisioning and demand estimation through self-organizing agent communities. In: 8th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS. pp. 481–488.

Jiang, B., Liu, Y., Chan, W., 2018. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In: Proceedings of the International Conference on Automated Software Engineering (ASE).

Langdon, W.B., Harman, M., 2015. Optimizing existing software with genetic programming. IEEE Trans. Evol. Comput. 19 (1), 118–135.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A., 2016. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 254–269.

McMinn, P., 2004. Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. 14 (2), 105–156.

Mendonça, D.F., Rodrigues, G.N., Ali, R., Alves, V., Baresi, L., 2016. GODA: a goal-oriented requirements engineering framework for runtime dependability analysis. Inf. Softw. Technol. 80, 245–264.

Nguyen, C.D., Miles, S., Perini, A., Tonella, P., Harman, M., Luck, M., 2012. Evolutionary testing of autonomous software agents. Auton. Agents Multi-Agent Syst. 25 (2), 260–283.

Nguyen, B.N., Robbins, B., Banerjee, I., Memon, A.M., 2014. GUITAR: an innovative tool for automated testing of GUI-driven software. Autom. Softw. Eng. 21 (1), 65–105.

Padgham, L., Zhang, Z., Thangarajah, J., Miller, T., 2013. Model-based test oracle generation for automated unit testing of agent systems. IEEE Trans. Softw. Eng. 39 (9), 1230–1244.

Panait, L., Luke, S., 2005. Cooperative multi-agent learning: The state of the art. Auton. Agents Multi-Agent Syst. 11 (3), 387–434.

Panichella, A., Kifetew, F.M., Tonella, P., 2015. Reformulating branch coverage as a many-objective optimization problem. In: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015. pp. 1–10.

Picard, G., Bernon, C., Gleizes, M.P., 2005. ETTO: emergent timetabling by cooperative self-organization. In: Engineering Self-Organising Systems, Third International Workshop, ESOA. pp. 31–45.

Praitheeshan, P., Pan, L., Yu, J., Liu, J., Doss, R., 2019. Security analysis methods on Ethereum smart contract vulnerabilities: a survey. arXiv preprint arXiv:1908.08605.

Rabiser, R., Guinea, S., Vierhauser, M., Baresi, L., Grünbacher, P., 2017. A comparison framework for runtime monitoring approaches. J. Syst. Softw. 125, 309–321.

Sen, K., Marinov, D., Agha, G., 2005. CUTE: a concolic unit testing engine for C. In: Wermelinger, M., Gall, H. (Eds.), 10^{Th} European Software Engineering Conference and 13th ACM International Symposium on Foundations of Software Engineering (ESEC/FSE '05). ACM, pp. 263–272.

Tamura, G., Villegas, N.M., Müller, H.A., Duchien, L., Seinturier, L., 2013. Improving context-awareness in self-adaptation using the DYNAMICO reference model. In: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 153–162.

Tamura, G., Villegas, N.M., Müller, H.A., Sousa, J.P., Becker, B., Karsai, G., Mankovski, S., Pezzè, M., Schäfer, W., Tahvildari, L., Wong, K., 2010. Towards practical runtime verification and validation of self-adaptive software systems. In: Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers. pp. 108–132.

Tonella, P., 2004. Evolutionary testing of classes. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004, pp. 119–128.

Vierhauser, M., Rabiser, R., Grünbacher, P., 2016. Requirements monitoring frameworks: A systematic review. Inf. Softw. Technol. 80, 89–109.

Viglianisi, E., Ceccato, M., Tonella, P., 2019. A federated society of bots for smart contract testing, replication package. https://github.com/stfbk/socrates-replication-package.

Yoo, S., Harman, M., Ur, S., 2013. GPGPU test suite minimisation: search based software engineering performance improvement using graphics cards. Empir. Softw. Eng. 18 (3), 550–593.