SynTest-Solidity: Automated Test Case Generation and Fuzzing for Smart Contracts

Olsthoorn, Mitchell; Stallenberg, D.M.; van Deursen, A.; Panichella, A.

DOI
10.1109/ICSE-Companion55297.2022.9793754

Publication date
2022

Document Version
Accepted author manuscript

Published in
The 44th International Conference on Software Engineering - Demonstration Track

SynTest-Solidity: Automated Test Case Generation and Fuzzing for Smart Contracts

Mitchell Olsthoorn
Delft University of Technology
Delft, The Netherlands
M.J.G.Olsthoorn@tudelft.nl

Dimitri Stallenberg
Delft University of Technology
Delft, The Netherlands
D.M.Stallenberg@student.tudelft.nl

Arie van Deursen
Delft University of Technology
Delft, The Netherlands
Arie.vanDeursen@tudelft.nl

Annibale Panichella
Delft University of Technology
Delft, The Netherlands
A.Panichella@tudelft.nl

ABSTRACT

Ethereum is the largest and most prominent smart contract platform. One key property of *Ethereum* is that once a contract is deployed, it can not be updated anymore. This increases the importance of thoroughly testing the behavior and constraints of the smart contract before deployment. Existing approaches in related work either do not scale or are only focused on finding crashing inputs. In this tool demo, we introduce SynTest-Solidity, an automated test case generation and fuzzing framework for Solidity. SynTest-Solidity implements various metaheuristic search algorithms, including random search (traditional fuzzing) and genetic algorithms (*i.e.,* NSGA-II, MOSA, and DynaMOSA). Finally, we performed a preliminary empirical study to assess the effectiveness of SynTest-Solidity in testing Solidity smart contracts.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Software testing and debugging**.

KEYWORDS

search-based software testing, test case generation, fuzzing, software testing, smart contracts

ACM Reference Format:
Mitchell Olsthoorn, Dimitri Stallenberg, Arie van Deursen, and Annibale Panichella. 2022. SynTest-Solidity: Automated Test Case Generation and Fuzzing for Smart Contracts. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516869

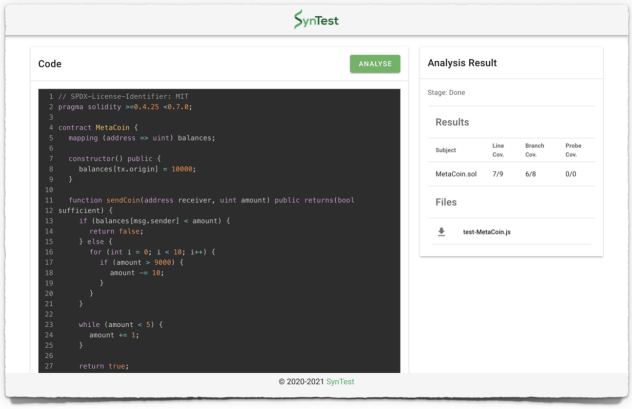

Figure 1: Online web service for generating test cases with SynTest-Solidity.

1 INTRODUCTION

Smart contracts are agreements between multiple parties on how certain tasks (*e.g.,* releasing or transferring funds) need to be executed. More specifically, they are short programs deployed to a distributed ledger (blockchain) that run when predetermined conditions have been met. This allows automating the execution of an agreement with a deterministic outcome without a trusted intermediary. Smart contracts have been gaining popularity in recent years. The largest and most prominent smart contract platform is *Ethereum*, which uses the *Solidity* programming language [9].

One key property of smart contracts is that they can not be updated anymore after their deployment. This property prevents the creator of a smart contract modifying the contract for their own benefit (*e.g.,* only allowing themselves to retrieve funds). However, this introduces certain challenges, such as when a contract contains a bug. Therefore, it is critical to thoroughly test the behavior and constraints of the smart contract as early as possible in the development lifecycle. Since smart contracts have complex interactions, manual testing becomes very difficult and error-prone [3].

Over the last few years, various techniques have been used to assist developers with testing *Solidity* smart contracts, like fuzzing, formal verification, and test case generation. Tools like sFuzz [4] have successfully used fuzzing techniques to produce test input data that causes errors or unwanted effects within the contract. However, since fuzzers only generate input data but no actual (runnable) test cases, they cannot create compositional tests (*i.e.,* a test case with multiple requests) nor test for the desired behavior of the contract.

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9223-5/22/05.
https://doi.org/10.1145/3510454.3516869

On the other hand, formal verification approaches aim to mathematically verify the behavior of a contract by transposing the contract into a formal proof language. In general, this approach does not scale and requires developers to provide a complex model of the desired behavior [1]. To the best of our knowledge, we have not found any study that indicates this is different for smart contracts. Lastly, test case generation allows developers to test smart contracts for both bugs and behavior in a more efficient and scalable way. In addition, this allows generated tests to be added to the existing test suite for regression testing purposes. To the best of our knowledge, there exists only one related work that focusses on test case generation for *Solidity* [2]. However, the research prototype used in the study is not specifically adapted for the *Solidity* language. For example, it does take into account *Solidity*-specific types, such as different sizes for integers, nor distingushes between signed and unsigned types. Besides, the tool does not generate assertions.

We have developed a tool, called SynTest-Solidity[1], to allow developers to more effectively and efficiently test their smart contracts. Our tool makes use of a genetic algorithm to evolve a set of initial randomly generated test cases to satisfy predefined test criteria (*i.e.,* function, line, and branch coverage). It does this by extracting objectives from the contract, feeding these into the search algorithm, and evaluating the produced test cases using *Truffle* (de-facto testing library for *Solidity*) and *Ganache* (local development blockchain). Developers can interact with our tool in two different ways. The first is a command line interface (CLI) that simplifies testing during development and allows developers to change the different parameters of the test case generation process. The second is an online web service that makes it possible for developers to use our tool without installing or setting it up locally. SynTest-Solidity is publicly available on NPM[2] and GitHub[3]. Instructions on how to set up and run the tool can be found on both platforms.

We performed a preliminary empirical study to test the effectiveness of SynTest-Solidity at generating test cases for 20 real-world *Solidity* smart contracts. This study shows that SynTest-Solidity can achieve, on average, 61 % branch coverage for these contracts.

2 TOOL

The goal of SynTest-Solidity is to allow developers to effectively and efficiently test the behavior and constraints of their smart contracts. The tool is primarly aimed at developers of smart contracts. In this section, we will discuss how developers can use our tool within their development workflow, explain the internal worflow of the tool, and the technical challenges this tool solves.

2.1 Usage Scenarios

To test their smart contracts, developers can use our tool in two ways. The first is on the command line using our CLI interface and the second is through our online web service.

2.1.1 Command Line Interface. The tool provides an easy-to-use Command Line Interface (CLI) that is publicly available and can be installed through the Node Package Manager (NPM). The CLI is highly configurable and offers a range of options, including changing the search algorithm, its internal parameters, and the search budget. A developer interested in generating test cases for *Solidity* smart contracts needs to install our tool from the NPM repository and run the following command: `syntest-solidity --include="<PATH_TO_THE_CONTRACT>" --algorithm="DynaMOSA" --search-time=120"`. This command will start the test case generation process, which includes analysis of the contract, search optimization, test case evaluation, and finally the assertion generation. These steps will be further discussed in Section 2.2.

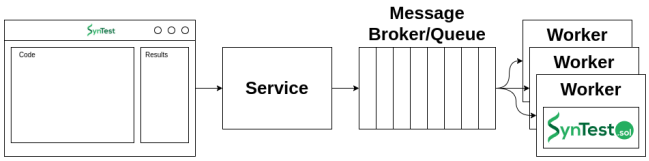

Figure 2: Architectural overview of the SynTest web service.

2.1.2 Web Service. In addition to the CLI, we provide an online web service[4] which allows users to interact with SynTest-Solidity without having to install the tool locally. Fig. 1 depicts the main interface of the web service. A developer can submit their contract to the web service and request it to be analyzed by clicking on the *analyze* button. This will start the test case generation process, at the end of which the generated test cases can be downloaded directly from the webpage. The webpage will also display relevant statistics, such as the number of lines and branches that are covered by the test cases.

Fig. 2 shows the architecture of the backend of the web service. This architecture consists of a webpage written in *Vue* that communicates with the *service* backend through websockets. The *service* backend is built with *Node.js*. Its role is to validate the user input and manage the sessions. Whenever a new contract is submitted, the *service* backend enqueues the job in the *RabbitMQ* message broker. The purpose of the message broker is to keep track of all the current submitted jobs and make sure that the *workers* process them. The workers are *Node.js* programs which retrieve jobs from the broker and perform the actual test case generation. The worker will create the files and folders required for SynTest-Solidity to run, after which it runs the tool and returns the resulting statistics and test files. The architecture is made such that the number of workers can be scaled based on the demand.

2.2 Tool Workflow

2.2.1 Analysis. To generate a test-suite, SynTest-Solidity uses white-box heuristics. Hence, the tool needs access to the source code for instrumentation which allows the tool to collect coverage information at runtime. The different steps of our tool are depicted in Fig. 3. The tool takes as input a smart contract, parses the source code to build the Control Flow Graph (CFG), and extracts the search objectives.

2.2.2 Search. The search process starts after extracting the objectives and instrumenting the code, as shown in Fig. 3. The tool

[1] https://www.syntest.org/
[2] https://www.npmjs.com/package/@syntest/solidity
[3] https://github.com/syntest-framework/syntest-solidity
[4] https://tool.syntest.org/

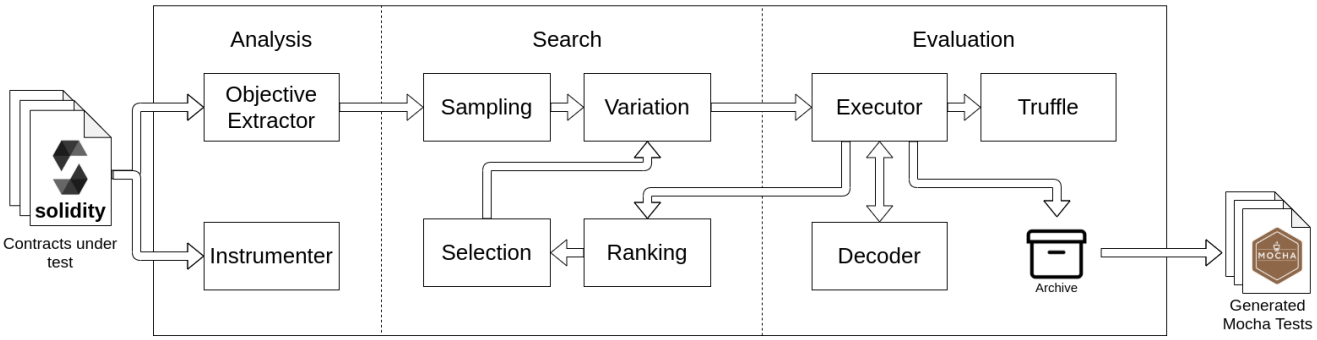

Figure 3: Internal architecture of SynTest-Solidity

boasts a number of different search algorithms, including random search, NSGA-II, MOSA, and DynaMOSA. For the purposes of this tool demo, we focus on the Dynamic Many-Objective Search Algorithm (DynaMOSA), which is the state-of-the-art meta-heuristic for white-box unit testing [6].

DynaMOSA evolves a set of randomly generated test cases. These test cases are generated using a *sampler* that provides the list of callable methods (*i.e.,* `public` and `external` methods) and constructors for the contract under test (CUT). A test case is encoded as a sequence of method calls. The *root* of the sequence is the contract deployment/instantiation made by invoking one of the public *constructors*. The remaining method calls in the sequence are obtained by randomly invoking some `public` and `external` methods in the CUT. Notice that the length of the test case is variable and can change through the generations.

The initial tests are iteratively evolved using *crossover* and *mutation*. The former operator creates new tests by swapping statements between pairs of tests (called *parents*). The latter operator applies small changes to the newly generated test cases, called *offspring*.

The population for the next generation is obtained by selecting the best test cases among parents and offspring using the *preference criterion* and the *non-dominated sorting* [6]. The goal of these two heuristics is to promote test cases that are closer to reaching the uncovered branches and lines in the code. The process is repeated until the predefined budget is depleted.

Note that DynaMOSA only optimizes the yet uncovered branches and lines. Every time an uncovered branch (or line) is reached, the corresponding test case is saved into the *archive*. The final test suite is composed of all test cases stored in the archive.

2.2.3 Text Execution. Our tool needs to execute each test case to determine how close that test is to covering the objectives. This is performed by the *objective functions*, which measure the distance to reaching an uncovered branch or line in the code using state-of-the-art heuristics, *i.e.,* the *approach level* and the normalized *branch distance* [6]. The flow of steps performed during test execution is also shown in Fig. 3.

A test case is first converted into a JavaScript test. This test is then executed on a fresh *Ganache* blockchain instance. This local blockchain instance hosts the CUT deployed using the *Truffle framework*. The test execution results are then collected and used to compute the approach level and the branch distance for the yet uncovered branches and lines.

2.3 Solidity-specific Features

SynTest-Solidity provides support for all data types and other features that are unique and specific to *Solidity*. In the following, we briefly elaborate on these features and how they are handled.

2.3.1 Data Types. The *Solidity* programing language includes multiple data types, including boolean, number, bytes, strings, and arrays. Compared to other languages (*e.g.,* Java), *Solidity* includes multiple subtypes for both integers and doubles. There are two main subtypes of integers: signed integers (`int`) and unsigned integers (`uint`). These subtypes also have different sizes, ranging from `uint8` up to `uint256`, which correspond to 8 and 256 bits, respectively. Similarly, double numbers can be both signed (`fixed`) or unsigned (`ufixed`) and have different sizes in the number of bits (*e.g.,* `ufixed128x18`, etc.).

SynTest-Solidity handles all these subtypes as it encodes integers (and float numbers) with an extra bit for the sign and different upper and lower bounds depending on the number of required bits. Note that SynTest-Solidity generates tests in JavaScript, which uses 52 bits for numbers. To allow representing larger numbers (up to 256 bits) required for *Solidity*, SynTest-Solidity uses the `BigNumber` library that allows arbitrary-precision arithmetic.

2.3.2 Solidity Addresses. `Address` is a special data type in *Solidity* and represent the intended recipient of a *transaction*. An `address` has 160 bits or 40 hex characters. An address always has a `0x` prefix in its hexadecimal format (base 16 notation). SynTest-Solidity uses two strategies to handle and instantiate addresses. The first strategy extracts address literals from the source code of the CUT. These constants are used as seeds when generating test cases with 50 % probability. This means that constant addresses in the code are (with some probability) used in the generated test cases. The address `0x0` (or zero-address) is a special constant used to indicate that a new contract is being deployed.

The second strategy uses pre-allocated addresses that are allocated by the *Truffle* framework when the contract is deployed at the beginning of each test case. These pre-allocated addresses

are accessible with the statement `account[index]`, where `index` points to the pre-allocated address to consider.

2.3.3 Transactions. Interactions with smart contracts are made via *transactions*. Transactions correspond (1) to either sending Ether to another account, (2) executing a contract method/function, or (3) adding a new contract to the network. Hence, a method call in the test case is required to have a recipient address in addition to the actual input parameters for the method being invoked. Therefore, a method call is encoded in SynTest-Solidity as an array of (1) input parameters, (2) return values, and (3) the address of the recipient.

2.3.4 Assertions. SynTest-Solidity generates assertions at the end of the search process as a post-process step. It does this by collecting the runtime values (*e.g.,* contract states and return values of invoked methods) from the final execution of the test cases. A specific type of assertion regards the runtime exceptions that can be thrown when the state-reverting security conditions (*i.e.,* `revert` and `require`) are not satisfied. If a test case triggers these runtime exceptions, the assertion generated by our tool asserts that the expected exception is triggered.

3 EVALUATION

To evaluate SynTest-Solidity, we tested 20 *Solidity* smart contracts submitted to *etherscan.io*. In particular, the selected contracts are written in Solidity versions 5 and 6, which are currently supported by our tool. We randomly selected these contracts among those that have been submitted to *etherscan.io* multiple times for security checking between January and June 2021. For the selection, we excluded duplicates and analyzed their cyclomatic complexity (`CC`) to exclude trivial contracts with no branching statements. As suggested by existing guidelines in the literature [6, 7], test case generation tools should be assessed on code units (e.g., classes in Java) with a cyclomatic complexity above two (`CC`>2). In our context, the 20 selected contracts have functions with cyclomatic complexity above three. The contracts and their characteristics are summarized in Table 1. The size of the contracts ranges from 23 LOC for `MetaCoin` to 307 for `Revive`.

While the benchmark might not be large enough for a complete empirical assessment, our goal is to show the ability of our tool in generating test cases with high code coverage and assertions for non-trivial, real-world smart contracts. Assessing the tools on a larger and more extensive benchmark is part of our future agenda.

Empirical set-up In this evaluation, we use the parameter values suggested in the related literature [6]. More specifically, we run DynaMOSA, which is the state-of-the-art search algorithm for unit-level test case generation [6]. In addition, we use a population size of 10 test cases. New test cases are generated using the *single-point crossover* with probability $p_c = 0.80$. Test cases are further changed using the *uniform mutation* with the probability $p_m = 1/n$, where n is the length of the test case. This operator either adds, deletes, or changes statements within each test case. These three mutation operators are equally probable.

We set an overall search budget of 30 minutes per smart contract. This search budget is larger than the one usually used in unit-test generation in other languages (e.g., Java [6]). This is because SynTest-Solidity has to deploy the contract under test to the smart contract network before each test case.

Table 1: Average (median) coverage achieved by SynTest-Solidity over 20 independents runs

Contract	LOC	Coverage		
		Function	Branch	Line
AavePoolReward	108	0.92	0.50	0.60
Baz	33	1.00	0.95	0.95
BirdOracle	134	0.89	0.59	0.66
Core_Fi_V3	62	0.88	0.56	0.59
CryptoGhost	165	1.00	0.84	0.79
CryptoSecureBankToken	254	0.93	0.50	0.73
DJCoin	195	0.88	0.67	0.77
EdenCoin	67	1.00	1.00	1.00
FreakCoin	139	1.00	0.60	0.69
GAZ_ERC20	71	0.55	0.55	0.54
INS	109	1.00	0.50	0.50
MetaCoin	23	1.00	0.88	0.89
Revive	307	0.84	0.41	0.64
Rootkit_finance	61	0.88	0.59	0.61
SLTDETHlpReward	291	0.78	0.49	0.63
Straight_Fire_Finance	62	0.88	0.59	0.61
TetherToken	98	1.00	0.50	0.58
ThriftToken	96	0.91	0.50	0.63
TimeMiner	174	1.00	0.67	0.81
WOLF	80	1.00	0.40	0.68
Mean		0.91	0.64	0.70

Empirical results. We run SynTest-Solidity 20 times for each smart contract to account for the randomness of the search process. Table 1 reports the median results achieved by SynTest with regard to function, branch, and line coverage. In all cases, SynTest-Solidity was able to generate test suites with high function coverage, which is 91 % on average. For branch coverage, the results vary between 40 % achieved on `Wolf` and 100 % achieved for `EdenCoin`. The produced branch coverage is greater than 50 % in all smart contracts except three. As a consequence, SynTest-Solidity yielded an average branch overage of 61 %. The results for line coverage are in-line with those achieved for branch coverage. Indeed, the mean line coverage is 68 %, with a minimum value of 54 % obtained for `GAZ_ERC20` and a maximum value of 100 % for `EdenCoin`.

4 CONCLUSIONS AND FUTURE WORK

In this paper, we demonstrated our tool, called SynTest-Solidity, that allows developers to effectively and efficiently test *Solidity* smart contracts. It achieves this by automating the process of test case generation using state-of-the-art metaheuristic search algorithms. As part of our future plan, we will extend the framework with linkage learning-based evolutionary algorithms [8], MOSA [5], and sFuzz [4]. Using these additional algorithms, we plant to perform a more extensive evaluation. SynTest is a modular framework that allows the tool to be extented to other languages (*e.g., JavaScript* or *TypeScript*). This is something, we will be working on in the future. Finally, we are planning to create a plugin for our tool for the most popular code editors to make it even easier for developers to use it.

REFERENCES

[1] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press.

[2] Stefan Driessen, Dario Di Nucci, Geert Monsieur, and Willem-Jan van den Heuvel. 2021. AGSolT: a Tool for Automated Test-Case Generation for Solidity Smart Contracts. *arXiv preprint arXiv:2102.08864* (2021).

[3] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing Transaction-Reverting Statements in Ethereum Smart Contracts. *arXiv preprint arXiv:2108.10799* (2021).

[4] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

[5] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.

[6] Annibale Panichella, Fitsum Mesheha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (Feb 2018), 122–158.

[7] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. 2021. Sbst tool competition 2021. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 20–27.

[8] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2021. Improving Test Case Generation for REST APIs Through Hierarchical Clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 117–128.

[9] Maximilian Wohrer and Uwe Zdun. 2018. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2–8.