Conformance Testing of Formal Semantics using Grammar-based Fuzzing

Diego Marmsoler[1] and Achim D. Brucker[1]

University of Exeter, Exeter, UK
{d.marmsoler, a.brucker}@exeter.ac.uk

Abstract. A common problem in verification is to ensure that the formal specification models the real-world system, i.e., the implementation, faithfully. Testing is a technique that can help to bridge the gap between a formal specification and its implementation.
Fuzzing in general and grammar-based fuzzing in particular are successfully used for finding bugs in implementations. Traditional fuzzing applications rely on an implicit test specification that informally can be described as "the program under test does not crash".
In this paper, we present an approach using grammar-based fuzzing to ensure the conformance of a formal specification, namely the formal semantics of the Solidity Programming language, to a real-world implementation. For this, we derive an executable test-oracle from the formal semantics of Solidity in Isabelle/HOL. The derived test oracle is used during the fuzzing of the implementation to validate that the formal semantics and the implementation are in conformance.

Keywords: Conformance Testing · Fuzzing · Verification · Solidity.

1 Introduction

Formal verification is an important means for ensuring that systems are correct, safe, and secure. But any formal verification of a computer program can only be as good as the formal semantics provided for the corresponding programming language. If the formal semantics does not capture the behavior of the real system adequately, the results of the verification can hardly be trusted. Consequently, the conformance of a formal semantics to the actual implementation is crucial in using formal verification to build correct, safe, and secure systems.

Sadly, in many important application areas, systems are not developed with a formal semantics as a point of departure. The formal semantics is often an afterthought, developed by reverse engineering informal documentation or, more often, the behavior of real systems. Thus, the problem of understanding to what extent such a formal semantics captures the behavior of the real system adequately becomes a big challenge.

Testing is a methodology that can help us to understand the relation between a formal semantics and its implementation. For example, one can derive test cases

L. Kovacs and K. Meinke (Eds.): TAP 2022, LNCS, pp. 1–20, 2022.
© 2022 Springer-Verlag. This is the author's version of the work. It is posted at https://www.brucker.ch/bibliography/abstract/marmsoler.ea-conformance-2022 by permission of Springer-Verlag for your personal use. BibTeX, Word, EndNote, RIS

from a test specification expressed in a formal semantics (e.g., using theorem-prover-based testing as discussed in [9]) that are then executed and validated on the real system to test that the real system conforms to the formal specification. Another approach is using property-based testing on the formal semantics (e.g., using [10]) itself to animate and explore the formal specification.

In this paper, we present a new approach: we use grammar-based fuzzing to generate programs that are then executed on the actual system and on the formal semantics. Our goal is, again, to ensure that our, post-hoc developed, formalization complies to the real world system, i.e., our formalization should behave, for the generated test cases identical to the implementation: in other words, the system should conform to its specification. We use grammar-based fuzzing, i.e, a testing technique that generates test cases from a formal grammar. As we are testing a system for compiling and executing Solidty programs, our test cases are Solidity programs. In such a setting, grammar-based testing has been used successfully for generating non-trivial programs that are syntactically correct. Still, there is no guarantee that the generated programs are type correct.

Our case-study is based on a formal semantics of Solidity [36]. Solidity is a programming language for expressing smart contracts (SCs) on the Ethereum blockchain. It is a Turing-complete, statically typed programming language whose concrete syntax has been designed to look familiar to people knowing Java, C, or JavaScript. The following shows a simple (artificial) function of a SC in Solidity for a withdrawal operation:

```
1 function wd(uint256 n, address payable r) public returns(bool) {
2     if (n < address(this).balance) {
3       r.transfer(n);
4       return true;
5     }
6     return false;
7 }
```

While Solidity is in many aspects similar to, e.g., Java, it differs in others. For example, the type system of Solidity provides, e.g., numerous integer types of different sizes (e.g., `uint256`) and the Solidity programs can make use of different types of stores for data (e.g., storage and memory).

In more detail, our contributions are:

1. An approach extending a parse grammar for Solidity to ensure that a generic grammar-based fuzzer generates *type correct* Solidity programs (Sect. 2.2), instead of generating syntactically correct, but often ill-typed, programs.
2. An approach for automatically deriving a test-oracle from a formal specification in Isabelle/HOL that allows to efficiently decide if a test case passes or fails *and* that allows to measure the test coverage in terms of statements and expressions of the target language usually based on an implicit test specification that informally can be described as "no crashes occur".
3. A framework for testing the compliance of a formal semantics of Solidity in Isabelle/HOL to their execution on the Ethereum blockchain (Sect. 2.3).

We evaluate our approach using a denotational semantics, in Isabelle/HOL [35], for a subset of Solidity v0.5.16 [36][1] that we described in [32]. Our formal semantics of Solidity [32] and the implementation of our compliance testing approach is publicly available [33].

Our results (discussed in Sect. 3) suggest that the approach has great potential to detect deviations of a semantics from a reference implementation. In particular, we successfully used the framework to uncover more than 30 such deviations in our original version of the semantics (some of these deviations are discussed in Sect. 3.1).

2 Approach

Fig. 1 provides a high-level overview of our approach. Its main components are:

- a formal semantics of a subset of Solidity v0.5.16 [36] in Isabelle/HOL [32],
- a test oracle that we generate automatically form the formal semantics, and
- a grammar-based fuzzer (based on Grammarinator [23]) that uses an extended (enriched with additional type information) Grammar of Solidity for generating type correct Solidity programs (i.e., test cases).

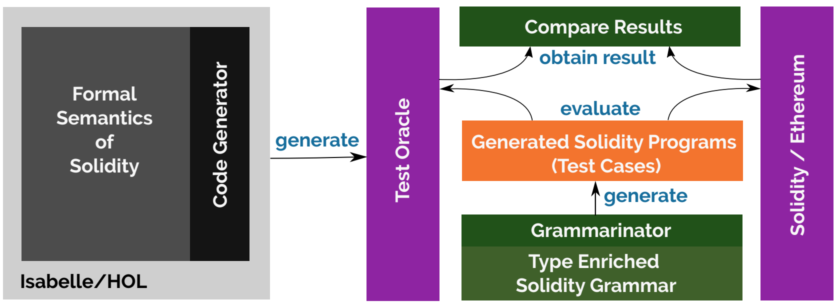

Fig. 1. Conformance Testing by Combining Test and Proof

The overall workflow is as follows: from the formal semantics of Solidity, given as a deep embedding into Isabelle/HOL, we generate a test oracle as executable command line program (see Sect. 2.1). Moreover, from a formal grammar describing the concrete syntax of Solidty, enriched with information to capture a subset of the typing rules of Solidity (see Sect. 2.2), we generated Solidity programs (i.e., test cases) using a grammar-based fuzzer (i.e., a fuzzing test tool that generates test cases systematically from a grammar, ensuring that all test cases are synatically correct with respect to the provided grammar). The results of these two steps feed into the actual test execution: the generated programs (together with their initial program state) are both executed on the Ethereum

[1] This is the currently supported default version of the Truffle test framework.

blockchain (i.e., the implementation) and our formally derived test oracle. Finally, the resulting programs states are compared and if they are equal, the test has passed, otherwise the test failed (see Sect. 2.3). We conclude this section by illustrating our approach by discussing an example SC generated by our framework (Sect. 2.4).

2.1 Deriving a Test Oracle from a Formal Semantics

We base our current work on a formalization of the Solidity in Isabelle/HOL [35], for details, we refer the reader elsewhere [32]. The core of the formal semantics is the semantic function for statements:

$$\mathcal{C}\colon\ \mathbf{C} \rightarrow \mathbf{Environment} \rightarrow \mathbf{State} \rightarrow \mathbf{Nat} \rightarrow (\mathbf{State} \times \mathbf{Nat})_{\bot}$$

Where **C** is the data type capturing the statements of Solidity (e.g., while loops, conditionals, assignments), **Environment** is the environment in which the expression that are part of a statement are evaluated in, and **State** is the program state. So far, this is mostly the standard definition of a semantic function for a denotational semantics. There are a few exceptions though, in particular: the execution of Solidity statements generates Gas costs. The initial balance is passed as **Nat** and the semantic function returns a tuple consisting of the new program state and the updated Gas balance. If an execution runs out of Gas during the execution, the program is terminated (see [32] and [41] for details).

Assuming that the semantics function is executable, in principle, the semantics function can be used as test oracle. In its simplest form, we can use the simplifier of Isabelle to symbolically evaluate the semantic function, in Isabelle. Given a concrete Solidity program (i.e., a statement), a ground environment, state, and a concrete Gas balance, the simplification in Isabelle will yield a state (or none, in case of an exception during the program evaluation).

This straightforward approach is not always desired. Firstly, symbolic execution using Isabelle's simplifier can be very slow. Second, it requires to interface the test framework directly with Isabelle. Thus, we make use of Isabelle's code generator to generate a Haskell implementation of the semantic function that can be compiled into a stand-alone program (i.e., the Test Oracle in Fig. 1). This program takes the same arguments as our semantic evaluation function and returns either an error or a new program state, which is used for determining if our semantics, for a given test case (i.e., a Solidity program and corresponding state) is compliant to the implementation.

Naturally, a formal semantics expressed in higher-order logic (HOL) is more abstract than a program in a programming language, such as Haskell [31]. For example, our semantics makes intensive use of (finite) sets, which results in a semantic function that is not executable in a strict sense. To make it efficiently executable, we need to transform expressions using sets into semantically equivalent expressions using lists. For this, we formally prove in Isabelle conversion lemmas, e.g.:

- **lemma [code]:**
  sorted_list_of_set (set xs) = sort (remdups xs)

– **lemma [code]:**
ffold_init *ct a c* = fold (init *ct*) (remdups (sorted_list_of_set (fset *c*))) *a*
These lemmas, proved for improving our testing approach, are added to the simplifier used for generating code, i.e., automatically converting sets to a more efficient list representation.

Moreover, we need to interface our test oracle with the test system, i.e., our generated program needs to be able to parse (the abstract syntax of) Solidity programs. For this reason, we have chosen Haskell as the target language (instead of one of the other targets Isabelle's code generator supports): Haskell allows us to make use of the automated generation of parsing and pretty-printing functions using Haskell's *deriving* feature.

Out setup is also used by Isabelle's code generator for SML, the language Isabelle's kernel is implemented in. This makes fast and efficient evaluation of Solidity expressions available to Isabelle itself, e.g., the value-statement:

```
value eval 1 stmt SKIP (STR ''089B'') (STR '''') (STR ''0'')
        [(STR ''089'', STR ''100''),(STR ''15f'', STR ''100'')]
        [] [(STR ''v1'', (Value TBool, Stackbool True))]}
```

yields `"STR ''v1==true\n089.balance==100\n15f.balance==100\n''"`. This setup is also automatically used by (potentially unsafe) proof tactics such as "eval" or Isabelle's specification-based testing tools and counter-example generators such as QuickCheck.

2.2 Generate Random Solidity Code

Grammarinator [23] is a grammar-based fuzzing tool: given a formal grammar of a language, Grammarinator generates random programs, which are syntactically correct (but, e.g., could still be ill-typed). The quality of the generated programs, however, depends on the provided grammar. In particular, if the grammar is too relaxed, this leads to the creation of programs which do not even compile. For the purpose of testing the semantics, however, we are interested in programs which we can run and compare with our semantics. Thus, we need to provide a grammar which is strong enough to lead to the creation of compiling programs.

In general, such a grammar needs to consider typing information and thus consists of more rules as we usually see in grammars used for the generation of parsers. For example, the grammar we used to test our subset of Solidity consists of more than 35 000 lines of ANTLR4 code. Thus, instead of manually creating the grammar, we implemented a tool which generates the corresponding grammar. Roughly speaking, the program consists of six main steps: 1. Generate rules for types 2. Generate rules for identifiers 3. Generate rules for variable declarations 4. Generate rules for expressions 5. Generate rules for lvalues 6. Generate rules for statements In the following, we discuss the rules generated by the tool.

Rules for types. Our subset of Solidity supports four basic types: boolean, address, signed and unsigned integers of 8-256 bit. Thus, the generated grammar contains rules for all of these types.

Rules for identifiers. In Solidity, we usually distinguish between *basic types* and *complex types* such as mappings and arrays. In addition, Solidity supports two different kinds of store to keep corresponding values: *memory* and *storage*. Our grammar provides rules to generate identifiers for all of these types.

In particular, the grammar provides one rule for producing identifiers for each of the basic types. Each of these rules allows for the creation of up to 10 identifiers for each kind of store:

```
IB:  'v_b_' ('s'|'m') [0-9] ;
IA : 'v_a_' ('s'|'m') [0-9] ;
IU8: 'v_u8_' ('s'|'m') [0-9] ; ...
IS8: 'v_s8_' ('s'|'m') [0-9] ; ...
```

In Solidity, mappings are only allowed to be kept in storage which is why we do not need to distinguish between memory and storage for mapping identifiers. The type of a mapping depends on the type of its keys and the type of its values, and thus we need to consider both of them when generating corresponding identifiers. In the following we show one such rule used to generate identifiers for mappings from addresses to unsigned 128-bit integers:

```
IMAU128: 'v_m_a_u128_' [0-9] ;
```

Similar rules are used to generate identifiers for all other types of mappings.

Array types are determined by the type of their values and the size of each of their dimensions. Thus, we need to consider all of them when generating rules for the production of array identifiers. We now show two such rules used to generate identifiers for a two-dimensional storage/memory array of signed 88-bit integers in which the first dimension is of size 3 and the second one of size 2:

```
IAS88_S_32: 'a_s88_32_s' [0-9] ;
IAS88_M_32: 'a_s88_32_m' [0-9] ;
```

Again, similar rules are used to generate identifiers for all other types of arrays.

Rules for variable declarations. In Solidity, mappings can only be used at the SC level and not for local declarations. Thus, the rules for local declarations only need to consider variables of basic types as well as variables of array type. However, the rules need to ensure consistency of the identifier with its type. For base types we only need to combine the corresponding rules for identifiers and types discussed above.

For arrays, however, the situation is more complicated. First, a declaration needs to explicitly state the store in which the array is kept (storage or memory). In addition, Solidity does not allow for uninitialized storage arrays. These two aspects need to be considered when creating rules for the generation of variable declarations. In the following we list two of these rules used to generate variable declarations for a two-dimensional storage/memory array of signed 88-bit integers in which the first dimension is of size 3 and the second one of size 2:

```
TS88 '[3][2]' ' ' STORAGE ' ' IAS88_S_32
   '=' as88_S_32_exp ';'
```

```
  TS88 '[3][2]' ' ' MEMORY ' ' IAS88_M_32
    ('=' (as88_S_32_exp | as88_M_32_exp))? ';'
```

Note that initialization is required for the version dealing with storage variables and optional for the version dealing with memory variables. In addition, in the first case, the variable needs to be initialized with a corresponding storage expression while in the latter case the variable may be initialized with either a storage or a memory expression as long as the types are consistent.

Rules for expressions. Our grammar provides rules to generate expressions for each of the different base types. In particular, expressions of a certain type can be directly generated by literals or identifiers of that type. In addition, an expression of a certain type can be obtained from identifiers of higher types by using corresponding keys. For example, the following rules are used to produce boolean expressions from corresponding mapping identifiers:

```
  | IMBB '[' bexpression ']'
  | IMAB '[' aexpression ']'
  | IMU8B '[' u8expression ']' ...
  | IMS8B '[' s8expression ']' ...
```

Again, the situation is more complicated when it comes to array identifiers. To this end, we first need to add rules to generate corresponding keys:

```
  I1: 'uint256(' [0-0] ')' ;
  I2: 'uint256(' [0-1] ')' ;
  I3: 'uint256(' [0-2] ')' ;
```

Here, $\mathtt{I}_k$ is a rule to access one dimension of an array of size k. Now we can use these rules to create rules to access the various dimensions of an array. For example, the following rules are used to generate an expression of a one-dimensional boolean memory array of size 3:

```
ab_M_3_exp
  : IAB_M_3
  | ab_M_31_exp '[' I1 ']'
  | ab_M_32_exp '[' I2 ']'
  | ab_M_33_exp '[' I3 ']'
  ;
```

```
ab_M_31_exp : IAB_M_31 ;

ab_M_32_exp : IAB_M_32 ;

ab_M_33_exp : IAB_M_33 ;
```

In particular, such an expression can be obtained by either using a corresponding identifier or by an expression of a two-dimensional memory boolean array using a corresponding key value. Finally, we can use the rules for array expressions to create rules for the corresponding base type. For example, the following rules are used to produce boolean expressions from corresponding array identifiers:

```
  | ab_S_1_exp '[' I1 ']' ...
  | ab_M_1_exp '[' I1 ']' ...
```

Having rules for base expressions of a certain type we can then combine them using corresponding operators. For example, the following rules are used to combine boolean expressions using logical operators:

```
  | bexpression OP_EQ bexpression
  | bexpression (OP_AND|OP_OR) bexpression
```

Care needs to be taken if we use operators over integers since in Solidity only certain types of integers may be combined. The following rules apply:

- Signed integers can be compared to other signed integers.
- Unsigned integers can be compared to other unsigned integers.
- Signed integers can only be compared to unsigned integers with smaller size.

For example, the following rule generates boolean expressions by comparing an unsigned 16-bit integer with a signed 24-bit integer:

```
 u16expression (OP_EQ|OP_LE) s24expression
```

However, our grammar does not provide rules to compare, for example a signed 16-bit integer with an unsigned 24 bit one since these two types are not compatible and thus the corresponding program would not compile.

The rules for the generation of lvalues is similar to the rules for expressions and not discussed further here.

Rules for statements. In general, our grammar provides two types of rules to generate basic Solidity statements: *assignments* and *transfer* commands.

Again, we need to be careful when creating assignment rules since they may easily lead to type errors and thus programs which would not compile. In particular, we need to ensure that lvalues and corresponding expressions have compatible types. This is again simple for expression of basic types. However, the situation is again more complex when it comes to arrays because we need to consider the dimension on the involved arrays. For example, the following rules are used to generate assignments for two-dimensional storage/memory arrays of size 3 and 2:

```
  | ab_S_32_lval '=' (ab_S_32_exp | ab_M_32_exp) ';'
  | ab_M_32_lval '=' (ab_S_32_exp | ab_M_32_exp) ';'
```

Note that Solidity does indeed allow the assignment of arrays located in different stores as long as the dimensions and types are compatible.

The transfer command allows transferring money from one account to another and our grammar provides corresponding rules for 10 different accounts:

```
  | AC0 '.' TRANSFER '(' u8expression ')' ';'
  | AC1 '.' TRANSFER '(' u8expression ')' ';'
  | AC2 '.' TRANSFER '(' u8expression ')' ';'            ...
```

Note that we only allow for unsigned integers of 8-bit expressions to be used in transfers to avoid that balances become empty too fast.

Finally, we can provide rules for higher-level statements such as blocks, conditionals, and loops which completes our grammar:

```
  | '{' declaration statement* '}'
  | 'if' '(' bexpression ')' '{' statement '}'
     'else' '{' statement '}'
  | 'while' '(' bexpression ')' '{' statement '}'
```

2.3 Testing Algorithm

The test framework is fully automated. Alg. 1 shows the core algorithm, where $[\,]$ denotes the empty list and $xs \leftarrow^{+} x$ denotes the list resulting from appending element x to list xs. The algorithm requires three configurations:

noStmt the number of programs we would like to test
noStates the number of states for each program which we would like to test
grammar the Solidity grammar file used to generate Solidity programs

It will then execute `noStmt`×`noStates` tests and return a list of failing statements with corresponding states.

The algorithm proceeds in `noStmt` rounds (line 2-line 20). For each round i, it first generates a random Solidity program `stmt` (line 3). It uses the grammar-based fuzzer Grammarinator [23] and the Solidity grammar `grammar` discussed in Sect. 2.2. Occasionally, the fuzzer may generate a statement twice, which is why we need to check if the statement was already processed (line 4-line 5).

Next, we analyze the generated program and extract all the variable identifiers `vars` (line 6). Note that this step requires to be able to identify an identifier within program code. However, this is possible since, as discussed in Sect. 2.2, our

Algorithm 1: TestSolidity

```
   Data: noStmt
   Data: noStates
   Data: grammar
   Result: results containing statements which lead to different results
 1 results, stmts ← [];
 2 for i ← 0 to noStmt do
 3 |   stmt ← generate(grammar);
 4 |   if stmt ∈ stmts then
 5 |   |   continue;
 6 |   vars ← extract(stmt);
 7 |   istmt ← instrument(stmt);
 8 |   astmt ← parse(istmt);
 9 |   scs ← [];
10 |   for i ← 0 to noStates do
11 |   |   vals ← [];
12 |   |   for var ∈ vars do
13 |   |   |   val ← random(var);
14 |   |   |   vals ←+ var, val;
15 |   |   result ← evaluate(astmt,vals);
16 |   |   scs ←+ createSC(vals,istmt,result);
17 |   result ← execute(scs);
18 |   if ¬result then
19 |   |   results ←+ astmt, vals;
20 |   stmts ←+ stmt;
```

grammar requires a certain structure for variable identifiers. Indeed, our grammar even ensures that the type of an identifier is encoded in its name which is required later on for generating random states.

The generated programs may also contain loops, and thus they may not always terminate. However, for the purpose of testing the implementation against the semantics, termination is actually desirable. Thus, line 7 analyzes the generated program and modifies all loops by adding a variable which increases with each iteration. In addition, it adds a disjunction to the loop condition which asserts that the variable is less than a certain value (see Sect. 2.4 for an example).

The resulting program `istmt` is in concrete Solidity syntax. However, our semantics requires a program given in abstract syntax. Thus, line 8 parses the modified program and returns its abstract syntax tree `astmt`.

Next we generate `noStates` random states for the currently processed program (line 10-line 16). To this end, we first iterate over all the variables extracted from the statement, generate a random value for it and add the variable as well as its value to a list `vals` (line 12-line 14). Note that for a meaningful program, the generated value needs to conform to the type of the identifier. However, as mentioned above, the type of an identifier is encoded into its name which allows for the generation of type-conform values.

Having generated a random state, we can execute the abstract statement using our evaluator. The resulting state is then used to create a corresponding test SC and add it to list `scs`. To this end, we modify a template SC with a single test function: The variables and their generated values `vals` are used to create corresponding variable declarations. The body of the function is given by the concrete statement `istmt`. The result state `result` from the evaluator is used to create corresponding assertions. An example of such an SC is discussed later on in Sect. 2.4.

After creating test SCs for each state we can execute them using the Truffle test framework [14]. The framework deploys the SCs to a local instance of the Ganache blockchain [13] and executes their test methods reporting failing assertions. Note that if an assertion fails, this means that the computed results for one of the variables deviates from the corresponding value of the result state obtained by the evaluator, and thus it will be recorded by the algorithm (line 18-line 19) in list `results`.

Finally, the statement is added to the list of processed statements `stmts` and a new iteration starts.

2.4 Example Smart Contract

Listing 1.1 shows parts of an SC generated from our test framework. As mentioned in Sect. 2.3, the SC consists of a single method `test` which contains the generated program. The program itself (`istmt` in Alg. 1) is provided in line 13-line 20. Note that the program contains a while loop, and thus it was modified by our instrumentation (method `instrument` in Alg. 1) which added a counter variable `counter1` to ensure termination.

The extracted storage variables are declared as SC variables (line 2-line 5) whereas the extracted memory/stack variables are declared locally (line 7-line 9). The generated state (`vals` in Alg. 1) was used to initialize the variables extracted from the statement (line 10-line 12).

Finally, the outcome of the evaluator (`result` in Alg. 1) was used to generate assert statements (line 21-line 27).

3 Evaluation

Our framework detected over 30 issues with our original semantics. In the following we discuss some of them and then present statistics about test execution.

```
1  contract TestContract0 {
2      uint8 v_u8_s8;
3      mapping(uint16 => uint8) v_m_u16_u8_9;
4      bool[1][2] a_b_12_s5;
5      ...
6      function test() public {
7          uint104 v_u104_m2;
8          uint104[1][1] memory a_u104_11_m2;
9          ...
10         v_u104_m2=14622709355569675963178665339646;
11         v_m_u16_u8_9[59381]=79;
12         ...
13         int8 counter1=int8(0);
14         while((v_m_u224_s240_1[uint224(444)]==
15             (v_u216_s1-v_u104_m2)) && counter1<int8(10)){
16             0xf7218C33533a3F22e3296F8b1DC0074B399355Eb
17                     .transfer(v_m_u16_u8_9[uint16(0)]);
18             counter1=counter1+int8(1);
19         }
20         ...
21         Assert.equal(v_m_u16_u8_9[59381]==79, true);
22         Assert.equal(a_u104_11_m2[0][0]==
23             8130097819054169632795960896007, true);
24         Assert.equal(
25             0xf7218C33533a3F22e3296F8b1DC0074B399355Eb
26             .balance==100000000000000000000, true);
27         ...
28     }
29 }
```

Extracted storage variables (lines 2–5)

Extracted memory/stack variables (lines 7–9)

Generated input state (lines 10–12)

Generated program (lines 13–20)

Computed result state (lines 21–27)

Listing 1.1. Example test contract generated by our testing framework.

3.1 Examples of Detected Issues

Integer arithmetic. In Solidity, arithmetic operations are allowed if the types of the operators are compatible (see Sect. 2.2). Consider, for example, the following Solidity statements:

```
assert (uint128(1) + int256(1) == int256(2));
assert (uint128(1) + uint256(1) == uint256(2));
assert (int128(1) + int256(1) == int256(2));
// uint256(1) + int128(1) is not allowed
```

As can be seen, arithmetic operations are possible if both operands are either signed or unsigned integers or if one is signed and the other unsigned and the size of the signed one is larger than the size of the unsigned one.

In the original version of our semantics we followed this rule. However, due to a misplaced comparison operator we assigned an error to expressions in which the first operand was an *unsigned* b_1-bit integer and the second operand a *signed* b_2-bit integer and $b_1 < b_2$.

Implicit initialization. In Solidity, uninitialized variables are implicitly assigned a default value. Consider, for example, the following Solidity fragments:

```
int128 x;
assert(x==0);
```

```
bool x;
assert(x==false);
```

```
address x;
assert(x==address(0x0));
```

Here, a variable of type *integer* is implicitly initialized with 0, a variable of type *bool* with `false`, and a variable of type address with address 0.

We followed this specification in our original semantics, however, due to a mistake in the initialization function, signed integer variables were initialized with their size instead of 0.

Storage pointers. In Solidity, assignments between variables denoting storage arrays copy the array from one storage location to another. Consider, for example, the smart contract (SC) `Test1` depicted in Fig. 2. Here, the assignment in line 3 actually copies the whole array from `var1` to `var2`. Thus, the assignment in line 6 only affects the copy in `var2` and `var1` remains unchanged.

This behavior changes, however, if we use a local storage variable. To see this, consider SC `Test2` in Fig. 2. Here, the assignment in line 5 only assigns the storage location of `var1` to `var2`. Thus, the assignment in line 6 actually changes the value of `var1`. In our original semantics we did not make this distinction and rather always copied the complete array.

Array copy. In Solidity, assignments between variables of arrays within different stores require the arrays to be copied between the stores. Consider, for example, the following SC:

```
1  contract Test {
2      uint8[2][2] var1=[[1,1], [1,1]];
```

```
1 contract Test1 {
2   uint8[2] var1=[1,2];
3   uint8[2] var2=var1;
4
5   function test() public {
6     var2[1]=0;
7     assert(var1[1]==1);
8   }
9 }
```

```
1 contract Test2 {
2   uint8[2] var1=[1,2];
3
4   function test() public {
5     uint8[2] storage var2=var1;
6     var2[1]=0;
7     assert(var1[1]==0);
8   }
9 }
```

Fig. 2. Example Solidity smart contract.

```
3
4        function test() public {
5            uint8[2] memory var2 = [2,2];
6            var1[1]=var2;
7            assert(var1[0][0]==1);
8            assert(var1[0][1]==1);
9            assert(var1[1][0]==2);
10           assert(var1[1][1]==2);
11       }
12   }
```

Here, the array `var1` is located in storage whereas the array `var2` is located in memory. Thus, the assignment in line 6 requires the array of `var2` to be copied to the location of the second array in `var1`. This changes all entries in `var1` where the first index is 1.

In our original semantics we considered the requirement to copy arrays in assignments when they are located in different stores. However, we made a mistake in calculating the storage locations: in our original version of the semantics, we would have changed all entries in `var1` where the second index is 1.

3.2 Statistics

After fixing all the detected bugs, we run the framework for several days which resulted in more than 10 000 successful tests. To cross-validate the effectiveness of the testing framework we also collected coverage information for the semantics using the Hpc tool [19]. The results are summarized in Fig. 3: Out of 123 definitions, 121 were executed during the tests. In addition, 186 alternatives (out of 524) and 1 592 expressions (out of 2 394) were executed.

Hpc also generates detailed coverage reports for every module. When inspecting these reports it turns out that the low number of covered alternatives is mainly because of missing executions of error cases (e.g. ill-typed programs). Consider, for example, Fig. 4 which shows an excerpt of the coverage report for the semantics of conditionals. From the figure we can observe that most of the code is indeed executed. In particular, the fuzzer generated programs and corresponding states which triggered the execution of both: the true and the false

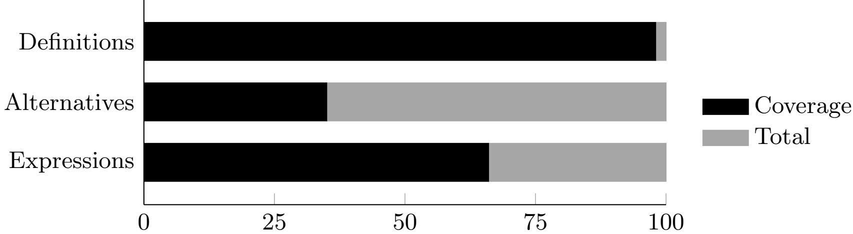

Fig. 3. Overall test coverage of semantics.

branch. In addition, we can see that the only code which was not executed is the one dealing with erroneous situations such as non-compiling programs.

```
208 stmt bal (ITE ex s1 s2) e st =
209   let {
210     gas = costs_min (ITE ex s1 s2) e st;
211   } in (if Arith.less_eq_nat bal gas then (Out_of_Gas, gas)
212          else (case Expressions.expr ex e st of {
213                 Nothing -> (Error, gas);
214                 Just (Storage.KValue b, Environment.Value Valuetypes.TBool) ->
215                   (case (if b == ReadShow.showL_b_o_o_l True
216                           then stmt (Arith.minus_nat bal gas) s1 e st
217                           else stmt (Arith.minus_nat bal gas) s2 e st)
218                     of {
219                     (res, c) -> (res, Arith.plus_nat c gas);
220                   });
221               }));
```

Fig. 4. Coverage report for conditional from `Statements.hs.html` [33]

This is because our framework only generates well-formed Solidity programs and thus the error cases are not executed. While this significantly increases the amount of good test cases it can also be a limitation if deviations occur in error cases. In particular, the framework does not allow to test that a program which does not compile leads to an error in the semantics. While this may be acceptable in some situations, it may not be acceptable for other situations which is why we are currently working on this as mentioned in future work.

4 Related Work

Grammar-based fuzzing. Fuzzing is a common technique to automatic testing [17]. Grammar-based testing is one technique for fuzzing structured inputs. First work in this area dates back to 1970s [22,38]. Modern tools in this area comprise, for example Grammarinator [23] and LangFuzz [24]. A stochastic approach is provided by Kifetew et al. [28].

The problem with these types of approaches is that traditional grammars are too relaxed (recall Sect. 2.2) and thus create only few relevant inputs. This problem with grammar-based fuzzing is well-known and attempts have been made to

improve the results. For example, Godefroid et al. [20] combines grammar-based fuzzing with whitebox testing to increase the number of meaningful tests. Thus, they achieve an increase from 11.9% coverage to 20% for testing a JavaScript interpreter using grammar-based whitebox fuzzing. Another work in this area is due to Majumdar and Xu [30] which provide an approach which combines grammar-based testing with concolic execution based on symbolic grammars to significantly reduces the number of test cases to achieve similar coverage.

While all these works are related to our work, the objectives of the approaches are different. Work in this area usually generates test input to look for runtime errors whereas with our work we want to detect semantic deviations between a semantic specification and a reference implementation.

Validation of semantics. According to Blazy and Leroy [7], there are five basic methods to validate formal semantics:

M1 Manual review and debugging
M2 Proving properties of the semantics, such as type preservation and determinism
M3 Using verified translations and trusted semantics
M4 Validating executable semantics, e.g. testing against test suites and experimental testing
M5 Using equivalent, alternate versions of the semantics

For example, many of the current available semantics for Solidity [2,5,15,34] are validated using M2.

If the semantics is executable then M4 is a common approach. For example, [8] validates the formal semantics of the Document Object Model (DOM), in Isabelle/HOL, by symbolically executing test cases from the official compliance test suite. Similarly, Filaretti and Maffeis [18], provide a formal semantics for PHP and validate it by executing 216 tests from the PHP Zend test suite and comparing the results with the Zend Engine. Another example is Politz et al. [37] which provide an executable semantics for JavaScript and validated it by executing 11 606 tests from the ES5 conformance suite. There exist even some examples of Soldity semantics which use this approach. Jiao et al. [26], for example, provide an executable semantics for Solidity in $\mathbb{K}$ [39] and validate it by executing 464 tests from the Solidity compiler test suite [1]. Another example is due to Yang and Lei [42] which provide an executable semantics for Solidity in Coq [40]. While all these works focus on the validation of semantics, none of them employ automatic fuzzing techniques to do so.

Using grammar based fuzzing for semantic validation. There are some examples of work which try to automate the task to validate semantics. For example, Guagliardo and Libkin [21] provide a formal semantics for SQL queries and validate it by implementing a custom query generator to generate 100 000 tests.

Most closely to our work, however, is the work of Bereczky et al. [6] where they validate formal semantics by property-based cross-testing. Here the authors describe an approach in which they use grammars to synthesize programs which

they then use to compare a semantics in the $\mathbb{K}$ framework [39] to a reference implementation There are, however, two notable differences to our work:

- They use their approach for validating the semantics of Erlang [4], a functional programming language. Thus, they avoid many of the problems occurring when you use the approach for an imperative language such as Solidity. In particular, they did not need to combine the grammar based fuzzer with random state generation as we did.
- In addition, their grammar does not seem to consider typing information which usually leads to a high number of low-quality inputs.

Compiler testing. Another relevant area of related work is the domain of compiler testing (see [11] for an overview). In particular, one could classify our work as a grammar-directed compiler testing approach with a formally verified test oracle. Our approach of enriching the input grammar with additional information to generate type-correct programs is closely related to the use of attributed translation grammars of Duncan and Hutchison [16]. One notable difference to our work, however, is that work in this area usually does not use a formally verified test oracle.

Integrating test and proof. Besides the works in the area of grammar-based fuzzing for semantic validation, there are several works combining test and proof. Here, we see two areas particularly closely related to our work: First, the integration of tools inspired by QuickCheck [12] into interactive theorem provers, e.g., Isabelle/HOL [10,27]. These tools are particularly valuable for finding count-examples prior to proof attempts of stated lemmata, saving resources in proof attempts that are deemed to fail. Moreover, such tools support the validation of the internal consistency of a formal semantics. Second, there is at least one tool, namely, HOL-TestGen [9], that derives actual test cases from a formal specification given in Isabelle/HOL. Notably, HOL-TestGen does not only generate test-cases, it also generates the test oracle that is used, during test execution, for checking if a test case passed or not. Compared to our work, HOL-TestGen generates a (small) test oracle for each individual test case while, in our current work, we derive a generic test oracle covering the whole formal semantics that can be used to check arbitrary test cases.

5 Discussion

Soundness of results. An important aspect to consider is the correctness of Alg. 1. Most of the steps performed by the algorithm are concerned with optimizing the quality of the produced test cases and as such they are not critical to the soundness of the approach. There is, however, one step which is indeed critical to ensure soundness which is the parsing of a generated program to an abstract syntax tree. If the parser modifies the structure of the program, then this could lead to wrong test results. Thus, it is important to ensure that the parser does not modify the structure of a program.

Grammar quality. The quality of the results produced by this approach strongly depends on the grammar which is used for the fuzzing of programs. If the grammar is too relaxed, the fuzzer will generate mostly non-compiling programs which are not very useful for testing the semantics. If the grammar is too restrictive, some types of programs will not be tested, at all. In general, the quality of the grammar can be improved by inspecting code coverage reports. If the reports indicate that certain parts of the semantics were not executed, at all, then, the grammar is probably too restrictive and needs to be adapted.

Random states. Finally, we would like to point out that as of now, program states are generated in a purely randomized fashion. While we do ensure that the values satisfy the type of a variable, there may be more efficient ways to generate states. Thus, improvements in this aspect could further increase efficiency of the test cases.

6 Conclusion

The problem that formal semantics need to be validated against their implementation is well-known, see, e.g., [3,25,29]. We address this problem, in this paper, by presenting an approach to validate formal semantics against a reference implementation using grammar-based fuzzing in combination with a test oracle generated from a formal semantics given in an interactive theorem prover, namely, Isabelle/HOL.

We evaluate the approach by using it to test conformance of a Solidity semantics against their reference implementation. Our results are promising in that the framework was able to uncover more than 30 deviations in the original semantics. In addition, an analysis of code coverage shows that the approach leads to high coverage results of more than 98% for top-level definitions, more than 66% for expressions, and more than 35% for alternatives. Inspecting the code coverage reports revealed that most of the code not executed deals with erroneous situations. While it is indeed desirable to keep the number of erroneous programs low, there should still be some programs creates which is why future work should investigate how to include a low percentage of erroneous programs.

Availability. Our formalization, the test framework, and the evaluation results are available under BSD license (SPDX-License-Identifier: BSD-2-Clause) [33].

Acknowledgements. We would like to thank Tobias Nipkow for useful discussions about the compliance testing. Moreover, we would like to thank Silvio Degenhardt for his support with implementing the semantics.

References

1. Solidity., https://github.com/ethereum/solidity, last checked on 2022-03-29.

2. Ahrendt, W., Bubel, R.: Functional verification of smart contracts via strong data integrity. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications. pp. 9–24. Springer (2020)
3. Arenis, S.F., Westphal, B., Dietsch, D., Muñiz, M., Andisha, A.S., Podelski, A.: Ready for testing: ensuring conformance to industrial standards through formal verification. Formal Asp. Comput. **28**(3), 499–527 (2016). https://doi.org/10.1007/s00165-016-0365-3
4. Armstrong, J.: Programming Erlang: software for a concurrent world. Pragmatic Bookshelf (2013)
5. Bartoletti, M., Galletta, L., Murgia, M.: A minimal core calculus for Solidity contracts. In: Pérez-Solà, C., Navarro-Arribas, G., Biryukov, A., Garcia-Alfaro, J. (eds.) Data Privacy Management, Cryptocurrencies and Blockchain Technology. pp. 233–243. Springer (2019)
6. Bereczky, P., Horpácsi, D., Kőszegi, J., Szeier, S., Thompson, S.: Validating formal semantics by property-based cross-testing. In: IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages. p. 150–161. IFL 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3462172.3462200
7. Blazy, S., Leroy, X.: Mechanized semantics for the clight subset of the c language. Journal of Automated Reasoning **43**(3), 263–288 (2009)
8. Brucker, A.D., Herzberg, M.: Formalizing (web) standards: An application of test and proof. In: Dubois, C., Wolff, B. (eds.) TAP 2018: Tests And Proofs, pp. 159–166. No. 10889 in LNCS, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_9
9. Brucker, A.D., Wolff, B.: On theorem prover-based testing. Formal Aspects of Computing **25**(5), 683–721 (2013). https://doi.org/10.1007/s00165-012-0222-y
10. Bulwahn, L.: The new Quickcheck for Isabelle – random, exhaustive and symbolic testing under one roof. In: Hawblitzel, C., Miller, D. (eds.) Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7679, pp. 92–108. Springer (2012). https://doi.org/10.1007/978-3-642-35308-6_10
11. Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., Zhang, L.: A survey of compiler testing. ACM Comput. Surv. **53**(1) (feb 2020). https://doi.org/10.1145/3363562
12. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: the fifth ACM SIGPLAN international conference on Functional programming. pp. 268–279. ACM Press (2000). https://doi.org/10.1145/351240.351266
13. ConsenSys Software Inc.: Ganache. https://www.trufflesuite.com/docs/ganache/, Accessed: 2021-05-01
14. ConsenSys Software Inc.: Truffle. https://www.trufflesuite.com/truffle, Accessed: 2021-05-01
15. Crafa, S., Di Pirro, M., Zucca, E.: Is Solidity solid enough? In: Bracciali, A., Clark, J., Pintore, F., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security. pp. 138–153. Springer (2020)
16. Duncan, A.G., Hutchison, J.S.: Using attributed grammars to test designs and implementations. In: Proceedings of the 5th International Conference on Software Engineering. pp. 170–178. ICSE '81, IEEE Press (1981)
17. Felderer, M., Büchler, M., Johns, M., Brucker, A.D., Breu, R., Pretschner, A.: Security testing: A survey. Advances in Computers **101**, 1–51 (Mar 2016). https://doi.org/10.1016/bs.adcom.2015.11.003

18. Filaretti, D., Maffeis, S.: An executable formal semantics of php. In: Jones, R. (ed.) ECOOP 2014 – Object-Oriented Programming. pp. 567–592. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
19. Gill, A., Runciman, C.: Haskell program coverage. In: Haskell Workshop. pp. 1–12. Haskell '07, ACM (2007). https://doi.org/10.1145/1291201.1291203
20. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. SIGPLAN Not. **43**(6), 206–215 (jun 2008). https://doi.org/10.1145/1379022.1375607
21. Guagliardo, P., Libkin, L.: A formal semantics of SQL queries, its validation, and applications. Proc. VLDB Endow. **11**(1), 27–39 (sep 2017). https://doi.org/10.14778/3151113.3151116
22. Hanford, K.V.: Automatic generation of test cases. IBM Systems Journal **9**(4), 242–257 (1970)
23. Hodován, R., Kiss, A., Gyimóthy, T.: Grammarinator: A Grammar-Based Open Source Fuzzer. In: Automating TEST Case Design. pp. 45–48. A-TEST 2018, ACM (2018). https://doi.org/10.1145/3278186.3278193
24. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: 21st USENIX Security Symposium (USENIX Security 12). pp. 445–458. USENIX Association, Bellevue, WA (Aug 2012)
25. Horl, J., Aichernig, B.K.: Validating voice communication requirements using lightweight formal methods. IEEE Software **17**(3), 21–27 (May 2000). https://doi.org/10.1109/52.896246
26. Jiao, J., Kan, S., Lin, S.W., Sanan, D., Liu, Y., Sun, J.: Semantic understanding of smart contracts: executable operational semantics of Solidity. In: SP. pp. 1695–1712. IEEE (2020)
27. Kappelmann, K., Bulwahn, L., Willenbrink, S.: Speccheck - specification-based testing for isabelle/ml. Archive of Formal Proofs (Jul 2021), https://isa-afp.org/entries/SpecCheck.html, Formal proof development
28. Kifetew, F.M., Tiella, R., Tonella, P.: Combining stochastic grammars and genetic programming for coverage testing at the system level. In: Le Goues, C., Yoo, S. (eds.) Search-Based Software Engineering. pp. 138–152. Springer International Publishing, Cham (2014)
29. Kristoffersen, F., Walter, T.: TTCN: towards a formal semantics and validation of test suites. Computer Networks and ISDN Systems **29**(1), 15–47 (1996). https://doi.org/10.1016/S0169-7552(96)00016-5
30. Majumdar, R., Xu, R.G.: Directed test generation using symbolic grammars. In: The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers. p. 553–556. Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1295014.1295039
31. Marlow, S.: Haskell 2010 language report. Available online https://www.haskell.org/onlinereport/haskell2010/ (2010)
32. Marmsoler, D., Brucker, A.D.: A denotational semantics of Solidity in Isabelle/HOL. In: Calinescu, R., Pasareanu, C. (eds.) Software Engineering and Formal Methods (SEFM). Lecture Notes in Computer Science, Springer-Verlag, Heidelberg (2021), https://www.brucker.ch/bibliography/abstract/marmsoler.ea-solidity-semantics-2021
33. Marmsoler, D., Brucker, A.D.: A denotational semantics of Solidity in Isabelle/HOL: Implementation and test data (Oct 2021). https://doi.org/10.5281/zenodo.5573225

34. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for Ethereum. In: Goldberg, I., Moore, T. (eds.) Financial Cryptography and Data Security. pp. 446–465. Springer (2019)
35. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
36. Online: Solidity documentation. https://docs.soliditylang.org/en/v0.5.16/, Accessed: 2021-05-01
37. Politz, J.G., Carroll, M.J., Lerner, B.S., Pombrio, J., Krishnamurthi, S.: A tested semantics for getters, setters, and eval in javascript. In: Proceedings of the 8th Symposium on Dynamic Languages. p. 1–16. DLS '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2384577.2384579
38. Purdom, P.: A sentence generator for testing parsers. BIT Numerical Mathematics **12**(3), 366–375 (1972)
39. Roșu, G., Șerbănută, T.F.: An overview of the K semantic framework. The Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012, membrane computing and programming
40. The Coq development team: The Coq proof assistant reference manual. LogiCal Project (2004), version 8.0
41. Wood, G.: Ethereum: A secure decentralised generalised transation ledger (version 2021-04-21). Tech. rep.
42. Yang, Z., Lei, H.: Lolisa: Formal syntax and semantics for a subset of the Solidity programming language in mathematical tool Coq. Mathematical Problems in Engineering **2020**, 6191537 (2020)