# Verifying Smart Contracts in Yul via Transformation to CHC by Interpreter Specialization

Elvira Albert[1], Emanuele De Angelis[2], Fabio Fioravanti[3], Alejandro Hernández-Cerezo[1] (✉), and Giulia Matricardi[3]

[1] Complutense University of Madrid, Spain
`elvira@fdi.ucm.es, aleher06@ucm.es`

[2] IASI-CNR, Rome, Italy
`emanuele.deangelis@iasi.cnr.it`

[3] DEc, University 'G. d'Annunzio', Chieti-Pescara, Italy
`fabio.fioravanti@unich.it, giulia.matricardi@unich.it`

**Abstract.** Yul is an intermediate representation that lies in between the (high-level) source code and the (low-level) bytecode languages for Ethereum smart contracts. Although it was proposed to favour the development of verification and optimization techniques, there exists no verifier that can be applied on Yul code directly yet. In this paper, we present a transformational approach to verifying Yul code by transforming it into an equivalent set of Constrained Horn Clauses (CHCs), leading, to the best of our knowledge, to the first approach to directly verify Yul code. Our transformational approach applies the first Futamura projection, i.e., specializes a Yul interpreter written in CHC with respect tothe Yul code to be verified. The verification of the transformed CHC code can rely on existing tools for CHC verification, namely we have used Z3 with the SPACER engine on our case studies.

## 1 Introduction and Motivation

Ethereum smart contracts and their verification have become rather active research topics both because of the novel features introduced by the languages used in the blockchain context (e.g., their gas model [25] opens new opportunities for optimization), and also because of the vulnerability of smart contracts (due to their immutability and public nature together with the fact that they often hold and manipulate financial assets their verification is crucial). Existing verification approaches have been developed either at the level of the source-code or at the *low-level bytecode* –named Ethereum Virtual Machine code– (abbreviated as EVM [25]). At the source-level, being Solidity [23] the most popular programming language that targets EVM bytecode, there are several Solidity verifiers [4,13,22,24]. At the EVM bytecode level, there are fewer verification tools, but still very popular [2,9,19]. While both types of approaches are useful, there is a significant gap in between them: approaches that operate on the source level may overlook information generated during compilation, whereas

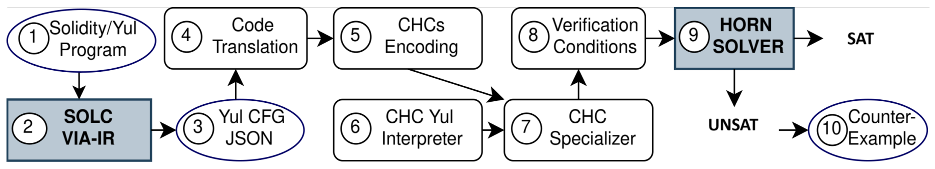

Fig. 1: Overview of the Verification Framework. Grey boxes represent external tools used in the framework, rounded boxes represent our internal components, and ellipses denote the input or output generated by external tools.

approaches that work directly on EVM bytecode offer limited guidance for generating counterexamples or fixing issues at the source level.

Yul [16] has been proposed recently as an intermediate language which lies in between the source-level Solidity code and the low-level EVM bytecode. Yul provides a simple syntax and semantics designed to be easily translated into bytecode, while remaining suitable for manual inspection, formal verification, and optimization. Unlike the EVM, Yul provides high-level syntax to avoid stack operations for variable management and low-level jumps in the control flow. In contrast to Solidity, Yul can be obtained in a Control Flow Graph (CFG) and Single Static Assignment (SSA) form (abbreviated as SSA-CFG in what follows) with explicit opcodes for the computation of expressions. One of the main purposes of Yul has been to help in the development of verification and optimization techniques. While Yul-optimization has been the focus of important efforts within the Solidity compiler team, to the best of our knowledge, there exists no verification tool yet applicable to Yul directly.

Motivated by the maturity of CHC verifiers [8] (such as Eldarica [12] and Spacer [15]) and the success of the interpretative approach to transform code from one language to another, we propose a verification framework from Yul to CHC based on the first Futamura projection [10] as depicted in Figure 1. As we will see, the input is a program that can be given in Solidity form which, importantly, can contain Yul assembly code ①. From there, using the Solidity compiler ②, we obtain a CFG representation in SSA-form of the Yul code ③. The next step is to encode as CHC clauses such Yul code ⑤ using our own code translator ④. We have implemented, as main contributions of the paper, a CHC interpreter of Yul ⑥ and an extension of an existing CHC specializer ⑦ that applies the first Futamura projection to generate a set of CHCs ⑧, representing the verification conditions (VCs) for checking that all the assertions included in the source code ① are met, thereby reducing this verification problem to a satisfiability problem for CHCs: all the assertions in ① are met *iff* the CHCs in ⑧ are satisfiable. Then, the satisfiability is proved, if at all possible (in general, the problem is undecidable), by using an off-the-shelf CHC solver ⑨. If proved unsatisfiable, a counter-example can be generated as well ⑩.

This paper is organized as follows. Section 2 introduces the Yul language and the CHC interpreter, as well as the CHC encoding of the Yul program under analysis. The main challenges when developing the interpreter, that will be discussed in the paper, concern the handling of the different types of memory used by Ethereum smart contracts and the semantics of EVM operations. Section 3 describes the specializer for CHCs that uses unfold/fold transformation rules whose application is guided by a strategy dedicated to generating VCs, known as the *VCG strategy* [20]. In Section 4 we apply our approach to two case studies that show the relevance of our work. The first example presents a contract fully implemented in Solidity; the second, on the other hand, is a contract that incorporates blocks of Yul code, showing how our methodology can also successfully deal with low-level code.

## 2 The Yul Language and CHC Interpreter

*Motivation.* Figure 2a presents a simple example of a Solidity program that includes a fragment written in Yul, inside the `assembly` block. The program defines a single contract, `Operation`, which implements the function `Positive-Difference`. This function computes the absolute difference `diff` between two non-zero, distinct positive integers, `x` and `y`. First, a `require` statement is used to enforce that `x>0, y>0` and `x`$\neq$`y`. From these preconditions, it follows that subtracting the smaller value from the larger can neither underflow nor produce zero, thus making the assertion `diff>0` at line 19 (L19 for short) valid. However, when analyzed using the model-checker built into Solidity's compiler (`solc`), `SolCMC` [4], a spurious counterexample is generated: `x=1, y=2, diff=0`. `SolCMC` issues a warning about potential false positives when Yul code is involved, which explains the inconsistency. Let us note that this version of the contract is more efficient than one written entirely in Solidity because, in the pure Solidity version, the compiler `solc` introduces additional, unnecessary checks, resulting in larger bytecode. Such efficiency gains motivates the introduction of Yul code within Solidity programs.

### 2.1 Introduction to Yul and its SSA-CFG Form

To support reasoning about both Solidity and Yul code, our approach operates on the Yul SSA-CFG representation shown in the example at Figure 2b. In this representation, the control flow is modeled using conditional and unconditional jump instructions, simplifying the CHC generation for program verification. Yul programs are typically divided into two parts: one for deploying the smart contract to the blockchain (`Operation`, line 1-or L1 for brevity), and another representing the deployed contract itself (`Operation_deployed`, L5). Each part includes a list of basic blocks representing their respective main entry point, `blocks` (L2, L6), along with a list of functions, `functions` (L7). This structure enables us to identify `require` and `assert` statements from the original Solidity code: functions related to these checks are named accordingly (e.g., `assert_helper` in L17 is the function that enforces the `diff>0` assertion). Both the main blocks and helper functions contain basic blocks comprising all necessary Yul operations. Each Yul operation receives a set of input values (`in`) and produces one or more

```
contract Operation {

  function positiveDifference(uint256 x, uint256 y)
    public pure returns (uint256) {
      require(x > 0 && y > 0 && x != y,
        "Inputs must be non-zero and different");
      uint256 diff;

      assembly {
        switch gt(x, y)
        case 0 {
          diff := sub(x, y)
        }
        default {
          diff := sub(y, x)
        }
      }

      assert(diff > 0);
      return diff;
    }
}
```

(a) Solidity Program, with a Yul fragment

```
{"Operation": {
    "blocks": [...],
    "functions": {...},
    "subObjects": {
      "Operation_deployed": {
        "blocks": [...],
        "functions": {
          "fun_positiveDifference": {
            "arguments": ["v0","v1"],
            "entry": "Block0",
            "blocks": [{...
                {"in": ["0x00","v27"],
                 "out": ["v28"], "op": "gt"},
                {"in": ["v28"], "out": [],
                 "op": "assert_helper"}
            }]},
          "assert_helper": {
            "arguments": ["v0"], "entry": "Block0",
            "blocks": [
              {"id": "Block0", "instructions":
                [{"in": ["v0"], "out": ["v1"],
                  "op": "iszero"}],
               "exit": {
                  "cond": "v1",
                  "type": "ConditionalJump",
                  "targets": ["Block2","Block1"]
                }},
              {"id": "Block2", "instructions": []},
              {"id": "Block1", "instructions":
                [{"in": [], "out": [],
                  "op": "panic_error_0x01"}]}]}},
          "panic_error_0x01": {
            "arguments": [], "entry": "Block0",
            "blocks": [...,
              {"in": ["0x01","0x04"], "out": [],
               "op": "mstore"},
              {"in": ["0x24","0x00"], "out": [],
                "op": "revert"}]}}}}}}
```

(b) Yul CFG Representation

```
% State
globals([]).
memory([0x00, 0x04, 0x40]).
%Function Declarations
fun(init_contract, [],
    [var(v6), var(v2), var(v4),
     var(v3), var(v5), var(v0)],
    'init_contract_Block0_1').

fun(assert_helper,
    [var(v0)], [var(v1)],
    'assert_helper_Block0_1').

fun(fun_PositiveDifference,
 [var(v0), var(v1)],
 [var(v27), ..., var(v28)],
 'fun_PositiveDifference_45_Block0_1').

fun(panic_error_0x01, [], [],
    'panic_error_0x01_Block0_1').
% For fun_PositiveDifference
at('fun_PositiveDifference_45_Block5_4',
   asgn(var(v28),
   expr(gt([num(0x00), var(v27)])))).

nextlab('fun_PositiveDifference_45_Block5_4',
 'fun_PositiveDifference_45_Block5_5').

at('fun_PositiveDifference_45_Block5_5',
   fun_call(assert_helper,
           [var(v28)], [])).

nextlab('fun_PositiveDifference_45_Block5_5',
      'fun_PositiveDifference_45_ret').

at('fun_PositiveDifference_45_ret',
   ret([])).
% For assert_helper
at('assert_helper_Block0_1',
   asgn(var(v1),
        expr(iszero([var(v0)])))).

nextlab('assert_helper_Block0_1',
        'assert_helper_Block0_jump').

at('assert_helper_Block0_jump',
    cj(var(v1), 'assert_helper_ret',
       'assert_helper_Block1_1')).

at('assert_helper_ret', ret([])).

at('assert_helper_Block1_1',
 fun_call(panic_error_0x01, [], [])).
% For panic_error_0x01
at('panic_error_0x01_Block0_2',
   mstore([num(0x01), mem(0x04)])).

nextlab('panic_error_0x01_Block0_2',
        'panic_error_0x01_Block0_3').

at('panic_error_0x01_Block0_3',
   revert([num(0x24), mem(0x00)])).
```

(c) Fragment of the CHC clauses, in Prolog format, generated from (b)

Fig. 2: Diagram showing the different representations of the source program in our framework. For (b) and (c), only a fragment of the translation is shown.

outputs (`out`). These Yul operations can be either: (1) the application of an EVM opcode to some arguments, identifying the resulting value (if any) as a variable; or (2) a jump to a function included in `functions`. For instance, in the function `fun_positiveDifference` (L8), the `gt` opcode is applied in L12-L13 to determine whether the positive difference (stored in variable `v27`) is zero. This variable is then passed as an argument to the `assert_helper` function (L14-L15), which checks whether the argument is zero (L21-L22) and performs a conditional jump based on the result (L23-L27). If the value is not zero, it calls the `panic_error_0x01` function (L32), which reverts the execution (L37-L38). Before the `revert`, a `MSTORE` operation is executed (L35-L36), storing the value `0x01` in the *Memory* region. To fully understand the effect of these operations, we now describe the main features of the EVM. The EVM is a stack machine that contains 256-bit words, and that manages data through different types of storage areas[4], each with specific purposes and duration.

- *Storage*: the contract's persistent memory where data that must survive between calls and transactions is saved. This region is managed through `SSTORE` and `SLOAD` opcodes.
- *Memory*: a volatile area that exists only for the duration of the current call. It is used for temporary operations such as internal calculations or preparing return data. It can be manipulated through `MSTORE` and `MLOAD` opcodes, among other operations.
- *Stack*: a volatile area used to store operands and intermediate results during arithmetic and logical operations. The Yul language does not model the stack directly but rather introduces *local variables* (*Locals*) which are managed through the operational stack.

In addition to the previous areas, the EVM can access information about the blockchain state or the current transaction through several opcodes. For instance, opcode `CALLER` returns the address of the caller of the current function, `CALLDATASIZE` returns the byte size of the calldata (the data sent as part of the transaction) or `NUMBER` returns the current block number. In our CHC interpreter we model the *Storage*, *Memory* and *Locals*, as well as the EVM environmental data.

## 2.2 Translating the Yul Program to CHC

In this section we present the component ④ of the verification framework shown in Figure 1, which is responsible for translating the Yul SSA-CFG ③ into CHCs.

*Preliminaries.* Constrained Horn clauses (CHCs) constitute a class of first order logic formulas where the Horn clause syntax is extended by allowing the use of formulas of an arbitrary, possibly non-Horn, constraint theory. In this paper, we consider CHCs whose constraints are linear integer arithmetic (*LIA*) expressions.

[4] We skip the *transient storage* area, which has been introduced recently in the Cancun hardfork [1], as it does not introduce new challenges.

```
<Program>  ::= <Globals> <Function>* <Clause>*
<Globals>  ::= globals([<GlobalElement>*])
<Function> ::= fun(<FuncName>, <VarList>, <VarList>, <Lab>).
<Clause>   ::= at(<Lab>,<Cmd>) | nextlab(<Lab>,<Lab>)
<Cmd>      ::= <BuiltIn> | <Asgn> | <Jumps> | <FuncCall> | <Ret> | ...
```

Fig. 3: Syntax of CHCs for encoding Yul programs

A CHC (or simply, a *clause*) is a universally quantified formula of the form $H \leftarrow c \wedge L$. The conclusion (or *head*) $H$ is either an atomic formula (atom, for short) or *false*. The premise (or *body*) $c \wedge L$ is a conjunction of a constraint $c$, and a (possibly empty) conjunction $L$ of atoms. A fact is a clause of the form $H \leftarrow c$.

Let $D$ be the usual interpretation for the symbols of theory *LIA*. A set S of CHCs is said to be $D$-satisfiable if it has a least $D$-model. The notion of a CHC we use in this paper is essentially the same[5] as the notion of a clause in Constraint Logic Programming (CLP) [14], whose concrete syntax will be used to present the CHC encodings and showcase the verification framework at work.

Figure 3 depicts a fragment of the grammar for CHC facts encoding Yul programs. Commands (`<Cmd>`) can be built-in operations of Yul (e.g., add, sub) (`<BuiltIn>`), assignments (`<Asgn>`), conditional or unconditional jumps (`<Jumps>`), function calls (`<FuncCall>`), or the `<Ret>` instruction, which indicates the final instruction of a function and can return zero or more elements. We adopt the same variable identifiers `v` that appear in the Yul SSA-CFG, including constants. For reasons of efficiency and readability, in the translation, these variables occur in terms wrapped using different functors, each indicating the memory region accessed: `off(v)` for the *Storage*; `mem(v)` for the *Memory*; `var(v)` for variables for the *Locals*. Function definitions are represented by facts of the form

```
fun(FuncName, Args, LocalVars, EntryLabel).
```

where `FuncName` is the function name, `Args` is the list of variables passed as arguments to the function, `LocalVars` contains the local scope variables used for internal operations, and `EntryLabel` defines the point from which to begin execution (i.e. the label of the first command of the function body). For example, function `assert_helper` in Figure 2c, lines L10-L12, is passed variable `v0` and uses the local variable `v1`. Its entry label is the command labeled by `assert_helper_Block0_1` at line L39.

The `at` and `nextlab` predicates are used to represent the labeled commands of the program, and the flow of control, respectively. The `at(Lab,Cmd)` atom is used to associate a label `Lab` to a command `Cmd`. For instance, the command in the previous example, at lines L39-L41, is associated with the `asgn` command by the following CHC fact:

[5] The term CHC is often used in the verification context [6], where the focus is on the construction of models for CHCs. The term CLP also refers to the notion of execution based on its operational semantics.

```
at(assert_helper_Block0_1, asgn(var(v1), expr(iszero([var(v0)])))).
```

The `nextlab(L,L1)` atom is used to specify the links between two labels: `L1` indicates the label of the command that is written, in the Yul program, immediately after the command with label `L`. For example, `nextlab(assert_helper_Block0_1,assert_helper_Block0_jump)` at lines L43-L44 connects the label of a block with that of a conditional jump command.

```
{"entries": ["Block6",
             "Block7"],
 "id": "Block5",
 "instructions": [
    {"in": ["v23","v24"],
     "op":"PhiFunction",
     "out":["v25"]},
    ...]}
```

Fig. 4: $\phi$-function in `Block5` of `fun_positiveDifference`

Another relevant aspect to address in the CHC encoding is the representation of $\phi$-functions. In SSA form, these expressions assign fresh variables to values modified along different paths in the CFG, ensuring every variable in the program is defined exactly once. They appear at joint nodes in the CFG and have the form $x_0 \leftarrow \phi(x_1 : B_1, \ldots, x_n : B_n)$, where $x_1, \ldots, x_n$ are values from the predecessor blocks $B_1, \ldots, B_n$ renamed in the joint block, and $x_0$ represents the resulting value. `switch`, `if/else`, and `for` statements in Yul introduce $\phi$-functions to handle modifications to the same variable across different paths. Figure 4 shows a $\phi$-function in `Block5` of `Operation_Deployed`. Here, the field `in` lists the $x_i$ values, and `entries` map each $x_i$ to their corresponding predecessor block $B_i$. The variable `v25` represents the value of `diff`, with its value depending on the branch taken in the `switch` statement (L10 in Figure 2a): `v23` corresponds to `x-y` when `x > y` (L11-L13), and `v24` to `y-x` (L14-L16). The translation of a $\phi$-function resembles a standard technique used in eliminating $\phi$-nodes during the generation of executable code, known as the *SSA destruction phase* [21]. For each $\phi$-node preceded by block $B_i$, the variable $x_0$ is assigned the value of $x_i$ immediately after executing $B_i$, ensuring $x_0$ holds the correct value based on the branch taken. This approach preserves the both original control flow and variable definitions without any loss of information. In the previous example, the following predicates are generated:

```
  at(Block6_2, goto(Block5_1_5)).
  at(Block7_2, goto(Block5_1_6)).
  at(Block5_1_5, asgn(var(v25), expr(phiFunction([var(v23)])))).
  nextlab(Block5_1_5, Block5_2).
  at(Block5_1_6, asgn(var(v25), expr(phiFunction([var(v24)])))).
  nextlab(Block5_1_6, Block5_2).
```

Label `Block5_1` is split into two different labels: `Block5_1_5` and `Block5_1_6`, which indicate the corresponding predecessor: `Block6_2` and `Block7_2`, respectively. In both cases, `v25` is assigned the appropriate value, and both paths continue to label `Block5_2`.

Table 1: Rules for assignment, SSTORE, function call, and conditional jump.

| Name | Code | Explanation |
|---|---|---|
| **asgn** | `tr(cf(cmd(L, asgn(X, expr(E))), Env),`<br>`  cf(cmd(L1, C), Env1)) :-`<br>`   eval(E, Env, V), update(Env, X, V, Env1),`<br>`   nextlab(L, L1), at(L1, C).` | Encodes the transition (a single execution step) for an assignment command. It evaluates expression $E$, and uses the resulting value to update $X$ in context. Finally it advances to the next command. |
| **sstore** | `tr(cf(cmd(L,sstore(V0,var(V1))),Env), Cf1) :-`<br>` lookup_local(V1, Env, K2),`<br>` tr(cf(cmd(L, sstore(V0, K2)), Env), Cf1).`<br><br>`tr(cf(cmd(L, sstore(V0, off(V1))), Env),`<br>`  cf(cmd(L1, C), Env1)) :-`<br>`    eval_arg(V0, Env, X0),`<br>`    update(Env, off(V1), X0, Env1),`<br>`    nextlab(L, L1), at(L1, C).` | Encodes the SSTORE command. In the first clause, `lookup_local` is used to retrieve the concrete key corresponding to `var(V1)` into which the value is to be stored and recursively invokes the `sstore` rule. In the second clause, `eval_arg` calculates the value to be written and updates the environment. |
| **fun_call** | `tr(cf(cmd(L, fun_call(F, InList, OutList)),`<br>`        Env), Cf3) :-`<br>`   fun_call_prologue(F,InList,Env, Cf1),`<br>`   reach(Cf1, Cf2),`<br>`   fun_call_epilogue(L,OutList,Cf2, Cf3).` | The function call rule initially invokes the `prologue`, which evaluates the arguments and constructs the configuration for the first label of the function (entry point). The `reach(Cf1, Cf2)` atom encodes reachability from the entry point of the function to one of its exit points. The exit points are computed by `epilogue`, which distinguishes two alternatives: regular termination and abnormal termination (error configuration). For regular termination we can have three cases: return of values (evaluation and assignment of the output, advancement of the label), revert and commit (corresponding to the return opcode). |
| **jumpi** | `tr(cf(cmd(L, jumpi(V0, L1, L2)), Env),`<br>`  cf(cmd(L1,C), Env)) :-`<br>`   eval_arg(V0,Env,X0), X0 = 0, at(L1,C).`<br><br>`tr(cf(cmd(L, jumpi(V0, L1, L2)), Env),`<br>`  cf(cmd(L2,C), Env)) :-`<br>`   eval_arg(V0,Env,X0), X0 =\= 0, at(L2,C).` | Encodes the conditional jump command: the variable `V0` is evaluated for the jump and one of the two possible destinations is chosen according to that value (0 or 1). |

## 2.3 CHC Interpreter of the Yul Language

In our CHC interpreter for Yul programs (component ⑥), the operational semantics is defined by a predicate `tr`, which represents the transition relation that leads from a *configuration*, a program execution state, to a new configuration, when a Yul command is executed. Configurations are represented as terms of the form: `cf(cmd(Lab, Cmd), (D, M, S))` where `Lab` is a program label and `Cmd` is the command with that label. The `(D,M,S)` tuple is used to represent the storage areas of our program as described in Section 2.1, where:

- `D` represents the area that includes both the execution context data and the *Storage*, providing globally accessible variables at every point in the code;
- `M` represents the *Memory* area;
- `S` represents the set of local variables in the *Locals*: they can only be accessed within the context of the function in which they are declared.

In Table 1 a selection of the most representative rules in our interpreter is shown and Table 2 describes some auxiliary predicates used in the former transi-

Table 2: Auxiliary Rules.

| Name | Code | Explanation |
|---|---|---|
| eval | `eval(E, Env, V)` | Represents the evaluation of an expression `E` in the environment `Env`, yielding the value `V` |
| | `eval(add([V0,V1],Env,V2)) :- eval_arg(V0, Env, Y0), eval_arg(V1, Env, X0), V2 = X0 + Y0` | The `eval` rule for addition. Similar rules are defined for other arithmetic and bitwise operations (`mul`, `div`, `and`, `or`, `not`, `shl`, `shr`, ...) for a total of 86 clauses. |
| update | `update(Env, X, V, Env1)` | `update(Env, X, V, Env1)` holds if `Env1` is obtained from the environment `Env` by assigning the value `V` to the program variable `X` |
| nextlab | `nextlab(L, L1)` | Holds if `L1` is the label immediately following `L` |
| at | `at(L1, C)` | Holds if `Cmd` is the corresponding command |
| eval_arg | `eval_arg(X, Env, V)` | Retrieves the value `V` of `X` in the corresponding component of the environment `Env` according to the scope of `X` |

tion rules. Some predicates, such as `eval`, which is used for evaluating arithmetic expressions, do not explicitly modify the configuration, but base their evaluation on the current configuration and the results they produce indirectly affect the new configuration (e.g, the result of an arithmetic operation will be used to update the environment). Note that, since we consider a multistep semantics [5], the `tr` clause for the function call is defined in terms of configuration reachability.

```
reach(Cf,Cf).
reach(Cf1,Cf3) :- tr(Cf1,Cf2), reach(Cf2,Cf3).
```

We have that `reach(Cf1,Cf3)` holds if `Cf3` can be reached from `Cf1` by zero or more steps (i.e. the `reach` predicate is the reflexive and transitive closure of the `tr` predicate).

## 3 Verification Conditions Generation by CHC Specialization

In this work we focus on the problem of checking whether all the assertions included in a Yul program $p$ are met, and we propose an automatic verification method that reduces it to a satisfiability problem for CHCs.

Let us consider the following clause

`false :- initConf(Cf1), reach(Cf1,Cf2), errorConf(Cf2).` $(Q)$

where: (i) `Cf1` represents an *initial configuration*, that is, the state of the EVM where the smart contract has been successfully deployed on the blockchain, (ii) `reach` represents the interpreter $I$ for Yul programs presented in Section 2.3, and (iii) `Cf2` represents an *error configuration*, that is, a call to the function `panic_error_0x01` resulting from a violation of the condition included in an assertion of the Yul program under analysis. Now, checking whether the assertions in a Yul program $p$ are met reduces to checking the satisfiability of CHCs as follows. Given a set $P$ of CHC facts encoding $p$ (see Section 2.2), the Yul interpreter $I$ (see Section 2.3), and the clause $Q$, we have that all the assertions included in $p$ are met if and only if the set of CHCs $P \cup I \cup \{Q\}$ is satisfiable.

State-of-the-art CHC solvers struggle at checking satisfiability of CHCs that contain complex terms, such as lists, that occur in the Yul interpreter to represent commands and environments. Thus, we apply the satisfiability-preserving unfold / fold CHC transformation rules to specialize it and remove the level of interpretation which is present in $I$. The specialization process realizes the *first Futamura projection* and produces a set *VC* of CHCs, called *verification conditions* (or *VCs*, for short), such that *VC* is satisfiable if and only if $P \cup I \cup \{Q\}$ is satisfiable. Moreover, since the generated VCs contain variables and constants only (see Fig. 6), CHC solvers can check their satisfiability more easily.

During CHC specialization we apply the following transformation rules: *unfolding*, *definition introduction*, and *folding*, according to the *Verification Condition Generation* strategy (*VCG*), similarly to what has been done for C programs in [5]. The rules and the strategy are described in the following subsections.

## 3.1 The CHC Transformation Rules

Let us first recall the definition of the transformation rules used by the specializer (component ⑦ in Figure 1).

*Unfolding.* The *unfolding* rule replaces an atom in the body of a clause with its definition, and is conceptually similar to inlining in imperative programming.

Let `C` be a clause of the form `H :- c,L,A,R`, where `H` and `A` are atoms, `L` and `R` are (possibly empty) conjunctions of atoms, and `c` is a constraint. Given a set `P` of CHCs, by unfolding atom `A` in the body of `C` we replace `C` by the set of clauses obtained by applying one resolution step rooted in `A` with respect to the clauses in `P` whose head unifies with `A`. We denote by *Unf*(`C`,`A`) the set of CHCs obtained by unfolding.

The application of the unfolding rule in the VCG strategy is guided by an annotation function that tells us whether or not a clause should be unfolded with respect to an atom in its body. This annotation guarantees that the number of applications of the unfolding rule is finite.

*Definition introduction.* A new predicate `newr` is introduced by the clause: `newr(X) :- reach(cf1,cf2)`, where `X` is a tuple of all variables occurring in the terms `cf1` and `cf2` representing configurations. Clauses introduced by the *definition introduction* rule are called *definitions*.

*Folding.* The *folding* rule is a special case of an inverse of the unfolding rule. Let `C` be a clause `H :- e,L,B,R` and let `D` be a definition `newr(X) :- A` such that for some renaming substitution $\theta$, $\mathtt{D}\theta$ is of the form `newr(Y) :- B`. Then `C` is folded with respect to `B` by using `D`, thereby deriving the new clause `H :- e,L,newr(Y),R`.

Notably, this version of the folding rule is less general than that in [5], but has simpler applicability conditions and is suitable for VC generation.

### 3.2 The VCG strategy

The transformation rules are applied according to the VCG strategy shown in Figure 5. It takes as input the set $P \cup I \cup \{Q\}$ and produces as output the equisatisfiable set *VC* of verification conditions ⑧ shown in Figure 1. The strategy keeps a set *Is* of CHCs to be specialized and a set *Ds* of definition clauses.

```
$Ds := \emptyset$;   $VC := \emptyset$;   $Is := \{Q\}$;
while there exists a clause C in $Is$ with an atom in its body
(unfold) Let A be the leftmost atom occurring in the body of C.
    $Us := Unf(\mathtt{C},\mathtt{A})$;
    while there exists a clause D in $Us$ whose body contains an unfoldable atom B
        $Us := (Us \setminus \{\mathtt{D}\}) \cup Unf(\mathtt{D},\mathtt{B})$;
(define and fold)  Let $Us$ be the clauses obtained by unfolding.
    while there exists a clause E in $Us$ of the form: H :- e, L, reach(cf1,cf2), R
        Let F be the clause obtained by folding reach(cf1,cf2) in E
        using a suitable definition D.
        $VC := VC \cup \{\mathtt{F}\}$;
        if (a variant of) D is not in $Ds$ then
            $Ds := Ds \cup \{\mathtt{D}\}$;   $Is := Is \cup \{\mathtt{D}\}$;
```

Fig. 5: The VCG strategy

The strategy terminates when all clauses in *Is* have been processed, and no new definitions are introduced.

*Correctness of the VCG strategy.* We say that VCG strategy is correct in the sense that the CHCs it produces as output are equisatisfiable with respect to those provided as input. This is a direct consequence of the correctness of the transformation rules with respect to the least model semantics [5]. In particular, if the VCG strategy terminates on the input set of CHCs $P \cup I \cup \{Q\}$ thereby producing the output *VC*, then $P \cup I \cup \{Q\}$ is satisfiable iff *VC* is satisfiable.

*Termination of the VCG strategy.* The unfolding annotation for VCG marks as unfoldable all atoms except those of the form `reach(cf(cmd(L, Cmd),_Env), _Cf2)` where `L` is the label of the entry point of a function or `Cmd` encodes the conditional jump command (`jumpi`). Thus, possibly recursive function calls and conditional jumps (used for encoding loops) are unfolded in a controlled manner and the (*unfold*) phase of VCG is guaranteed to terminate. Moreover, the (*define and fold*) phase is guaranteed to terminate because only a finite number of new definitions is introduced. Indeed, the definition introduction rule introduces new clauses of the form `new(X) :- reach(cf1,cf2)`, where `X` is the tuple of variables occurring in `cf1` and `cf2`. The new definitions abstract away the constraints representing the actual parameters provided to functions and the expressions occurring in conditional jumps, and therefore can be used to fold all atoms representing different calls to the same function or the same loop head. Consequently, the maximum number of definitions that can be introduced by the VCG is equal to the sum of the number of functions and conditional jumps occurring in the program. Thus, the VCG strategy terminates.

For the sake of efficiency, when there is no risk of non-termination, we call some atoms instead of fully unfolding them.

Each step of the verification pipeline shown in Figure 1 is fully automatic, except for the task of specifying the initial configuration, that is, the label of the first instruction of the function under analysis and its call context. The interpreter and the specializer, which have been implemented as a module of the VeriMAP system [7], are available at `https://github.com/chc-lab/yul-chc`.

## 3.3 Example application of the VCG strategy

Let us show the application of the VCG strategy on the property we are studying for contract `Operation` in Figure 2. After some iterations, the VCG strategy reaches the conditional jump (`jumpi`) encoding the `switch` statement shown in Figure 2a at L10, and therefore introduces the following definition `D_jumpi`

```
jumpi(A,B,...,U,...) :- reach(
  cf( cmd(..., jumpi(var(v24), case0, default)),                    (1)
      ([('msg.value',A),...],[(0,B),...],[...,(v24,U),...])),       (2)
  cf( cmd(panic_error_0x01,abort),...)) ).                          (3)
```

The term at line (1) encodes the configuration whose command is the conditional jump (`jumpi`). According to the value of `var(v24)`, it either jumps to the command at label `case0` or to the command at label `default`. The term at line (2) encodes the environment mapping storage and memory locations, as well as local variables, to their values (e.g., `(v24,U)` maps the local variable `v24` to its value represented by the logic variable `U`). Finally, the term at line (3) encodes the *error configuration* whose command is `abort` and whose label is `panic_error_0x01` (the final environment is omitted). The error configuration has been introduced in the interpreter for verification purposes and, in the rest of this section, we will use `errCf` to denote it.

Then, the VCG strategy proceeds by unfolding the atom `reach(cf(...), errCf)` occurring in the definition of the predicate `jumpi`. After some unfolding steps, we obtain the following two clauses.

```
jumpi(A,B,...,U,...) :- U=0, reach(cfCase0,errCf).
jumpi(A,B,...,U,...) :- U=1, reach(cfDefault,errCf).
```

where `U` is a logic variable whose value is `0` if `gt(x,y)`, and `1` otherwise, and `cfCase0` and `cfDefault` are configurations representing the two alternative branches of the `switch` command. In particular, the command occurring in `cfCase0` is `asgn(var(v26),expr(sub([var(v1),var(v0)])))`, encoding the Yul statement `diff := sub(x,y)` at L12 of Figure 2a. Similarly, the command occurring in `cfDefault` is `asgn(var(v26),expr(sub([var(v1),var(v0)])))`, encoding the Yul statement `diff := sub(y,x)` at L14 of Figure 2a. The unfolding process stops when no clause contains atoms that are annotated as unfoldable. One of the clauses obtained by unfolding is a clause, say `U_jumpi_1`, of the form:

```
jumpi(A,B,...,U,...) :- U=0, G1=F-E, ...,
  reach(cf(cmd(...,fun_call(cleanup_t_rational_0_by_1,...),...),
        cf(cmd(...,ret([var(v4)])),...))  ),
  reach(cf(cmd(assert,asgn(var(v1),expr(iszero([var(v0)])))),...),
        errCf)).
```

Recall that, for function calls, the `fun_call_epilogue` predicate of the interpreter considers two alternatives: the regular termination of the function and its abnormal termination leading to the error configuration. Indeed, clause `U_jumpi_1` contains two `reach` atoms. The first one represents the regular termination of `cleanup_t_rational_0_by_1`, producing as result `v4`, whereas the second atom represents the abnormal termination of the helper function `assert`. Note that `assert` is responsible for the evaluation of the condition `diff > 0` at line 9 of Figure 2a: if the condition does not hold, it calls the auxiliary function `panic_error_0x01` leading to the error configuration `errCf`. Now, the VCG strategy introduces two additional definitions :

```
cleanup_t_rational_0_by_1(...) :-
  reach(cf(cmd(...,fun_call(cleanup_t_rational_0_by_1,...),...),
        cf(cmd(...,ret([var(v4)])),...))  ).
assert_ERR(...) :-
  reach(cf(cmd(assert,asgn(var(v1),expr(iszero([var(v0)])))),...),
        errCf)).
```

and uses the new definitions to fold `U_jumpi_1` as follows:

```
jumpi(A,B,...,U,...) :- U=0, G1=F-E, ...,
                cleanup_t_rational_0_by_1(...), assert_ERR(...).
```

The VCG strategy continues by performing the (*define and fold*) step on the other clauses obtained from the (*unfold*) step. The complete specialization of the definition `D_jumpi` is shown in Figure 6. On the left side, we show the CHCs for `case 0`, with the constraints `U=0` (`gt(x,y)` holds) and `G1=F-E` (`diff := sub(x, y)`). As already mentioned, clauses at line 1 and 6 represent the regular termination of `cleanup_t_rational_0_by_1` and the abnormal termination of the helper function `assert`, respectively, and differ only for the constraints obtained by the evaluation of comparison statements included in the function `positiveDefinition`. The clause at line 11 represents the abnormal termination of `cleanup_t_rational_0_by_1`. On the right side, we show similar CHCs for the `default` case, with the constraints `U=1` (`gt(x,y)` does not hold) and `F1=E-F` (`diff := sub(y, x)`).

```
jumpi(A,B,E,F,U,...,E1,F1,G1) :-
   U=0, G1=F-E, G1>E1, F1=1, ...,
   cleanup_t_rational_0_by_1(...),
   assert_ERR(...).

jumpi(...) :-
   U=0, G1=F-E, G1=<E1, F1=0, ...,
   cleanup_t_rational_0_by_1(...),
   assert_ERR(...).

jumpi(...) :-
   U=0, G1=F-E, ...,
   cleanup_t_rational_0_by_1_ERR(...).
```

```
jumpi(...) :-
    U=1, G1=E-F, G1>E1, F1=1, ...,
    cleanup_t_rational_0_by_1(...),
    assert_ERR(...).

jumpi(...) :-
    U=1, G1=E-F, G1=<E1, F1=0, ...,
    cleanup_t_rational_0_by_1(...),
    assert_ERR(...).

jumpi(...) :-
    U=1, G1=E-F, ...,
    cleanup_t_rational_0_by_1_ERR(...).
```

Fig. 6: CHCs obtained from the specialization of `jumpi` presented in Section 3.3, corresponding to the `switch` statement in the Yul code.

## 4 Case Studies

Let us show the relevance of our work on two case studies. The first, `Auction`, analyzes a smart contract extracted from the article [18], while the second, `Splitter`, examines a modified version of the `PaymentSplitter` [17] from the popular OpenZeppelin smart contract, which includes inline assembly. For each case study, we start with an initial configuration `Cf1` and show that no execution path can lead to an error configuration `Cf2`.

```
contract Auction {
  uint public bid = 0;
  uint public cash = 0;
  address payable public winner;

  constructor() {
      winner = payable(address(0));
  }

  function offer() public payable {
      uint new_bid = msg.value - 1015 wei;
      require(new_bid > bid);

      if (winner != payable(address(0))) {
          assert(bid <= cash);
          winner.transfer(bid);
          cash -= bid;
      }

      bid = new_bid;
      cash += msg.value;
      winner = payable(msg.sender);
  }
}
```

Fig. 7: Auction contract

The `Auction` contract in Figure 7 implements a simple auction in Solidity: bidders send Ether, from which a net bid (`new_bid`) is calculated, which must exceed the current stored one. If there is a previous winner, the contract checks that the balance (`cash`) is sufficient to repay it, makes the transfer, and updates the status. Here, `require(new_bid > bid)` acts as a precondition, verifying that the incoming bid is indeed higher than the current bid before continuing. Only if this condition is met, the flow continues. The block `if(winner != address(0))` determines whether it is a bid following the first bid: only in this case is the previous winner refunded. `assert(bid <= cash)` is a post-condition and at the same time a contract invariant, which guarantees that the internal state (`cash`) is always sufficient to cover the amount to be returned (`bid`). If by a logical error the state is inconsistent (e.g. `cash < bid`), the assert will fail and cause irreversible revert, signaling a critical bug. In this way, `require` protects the input conditions, while `assert` ensures that the financial correctness invariant of the contract remains valid once the internal transactions are executed. For this specific example, the initial configuration used in clause $Q$ of Section 3 is defined as follows:

```
Cf1 = cf(cmd(external_fun_offer_85_Block0_3,
                fun_call(fun_offer_85, [], [])), Env1)
```

which corresponds to the initial configuration where, in the post-deployment environment (`Env1`), a call is made to the `fun_offer_85` function (the function under consideration which contains the `assert`). An equivalent version of this same contract, in which some operations were replaced by bytecode, was subjected to the same verification process: the results obtained exactly match those of the original Solidity contract, confirming the unique ability of our approach to handle contracts in both source code and bytecode.

```
function releasable(address account)
  public view returns (uint256) {
    require(msg.sender == account);
    require(account == payee0
            | account == payee1
            | account == payee2);
    require(totalShares > 0);
    require(amountp0 >= released0
        && amountp1 >= released1
        && amountp2 >= released2);

    uint256 sumPend; uint256 yours;

    assembly {
      for { let i := 0 } lt(i, 3)
        { i := add(i, 1) } {
          let payee := sload(i)
          let amount := sload(add(6, i))
          let rel := sload(add(9, i))
          let pend := sub(amount, rel)
          sumPend := add(sumPend, pend)
          if eq(payee, account)
            { yours := pend }
        }
    }
    assert(address(this).balance
           >= sumPend);
    return yours;
}
```

Fig. 8: Releasable function

The `Splitter` contract (see the full contract in the repository) allows funds to be split between three predetermined addresses, each with a fixed payee quota defined at the time of creation. Each time one of the three `payees` sends ETH to the contract, the `receive` function checks that the sender's address matches one of the stakeholders and that the amount deposited is exactly equal to its `share`, accumulating the total amount deposited for each stakeholder. To check how much remains to be withdrawn, each payee may invoke the `releasable` function (Figure 8) which, in `assembly`, iterates over the three addresses, calculates the difference between the `amount` paid in and the amount already `released` for each, and identifies the amount due. To ensure the integrity of the state, a final `assert` in the releasable function ensures that the balance of the contract is at least equal to the sum of all outstanding amounts, preventing inconsistent conditions. The `release` function then allows the payee to withdraw a validated amount: it checks, via `require`, the origin of the call, the contract balance, the positive amount and compliance with the limit calculated by releasable, and then updates the released counters and transfers ETH with call. For this specific example we define:

```
Cf1 = cf(cmd(external_fun_releasable_259_Block2_3,
            fun_call(fun_releasable_259, [var(v0)], [])),Env1),
```

where `v0` is the variable representing the value passed as input to the function. Our verifier confirms that the property holds as expected. As with contract `Operation` in Figure 2, the version of this contract written entirely in Solidity is less optimized than the one including Yul code, making the Yul-based version more desirable for deployment and more used in practice by smart contract developers. However, as we have noted before, there is no other tool that can verify Yul code.

## 5 Conclusions, Related and Future Work

We have proposed a transformational approach to the verification of Ethereum smart contracts which consists in transforming their intermediate Yul representation into an equivalent CHC program which can be directly used as input to off-the-shelf CHC verifiers (like Eldarica [12] or Spacer [15]). Our work is based on the interpretative approach to compilation [10,6] which has been successfully applied to transform low-level code into higher-level representation (e.g., transforming Java bytecode to CHC [11]) and to transform from one high-level code to another (e.g., transforming C to CHC [5] and other imperative programs [20]). The advantages of the interpretative approach include, among others, faster development time as –assuming that a specializer is available (as it was in our case)– one just needs to implement the Yul interpreter in CHC, as well as higher reliability in the correctness of the implementation as –assuming that the specializer is a trusted component– one just needs to ensure the correctness of the interpreter implementation.

On the one hand, compared to existing verifiers for Ethereum smart contracts [4,13,22,24], we are behind their capabilities as some of them are being developed already for a number of years and are able to prove complex properties such as callback freeness [3]. Our work is a first step towards the direction of building a strong verifier able to prove complex properties as well. On the other hand, our work is covering an important gap in the verification of Ethereum smart contracts as there is no other tool able to directly analyze Yul code yet. However, Yul is being proposed by the Ethereum community as the target language for high-level optimization and analysis stages so that all target platforms equally can benefit from progress at the Yul level. Hence our work has the potential of providing such benefits to all target platforms as well.

Finally, while our work is based on the VCG approach in [5] for C programs, they differ in some significant aspects. First, the interpreter has to take into account the Yul syntax and semantics as well as the EVM memory model, which contains blockchain-specific components, and the management of the different ways in which the execution of a Yul function can terminate (ret, commit, revert, error). There are important differences as well between the VCG strategy for Yul and C: (1) the unfolding annotation for C programs considers as non-unfoldable the entry points of functions and conditional jumps, as well as the junction points of conditional jumps. However, for Yul, the junction points are not directly recognizable thus in the current version of the strategy we do not consider them; (2) the VCG strategy for Yul requires extending the unfolding annotation to keep track of atoms that are associated with the contract deployment phase on the blockchain; (3) the VCG strategy for C programs makes use of additional annotations that help to reduce the number of new predicate definitions as well as the number of their arguments. For instance, the VCG strategy can be configured to use annotations that keep track of (i) the absence of side-effects for functions thus reducing the number of variables for predicates, and (ii) to the impossibility for auxiliary functions of reaching an error configuration thus reducing non-determinism and the number of clauses. This fine tuning of the VCG strategy for C programs was instrumental to achieve the optimal time and space complexity presented in [5]. For Yul, we aim to extend the strategy to get similar results in the near future. Besides, although we have implemented all components of the overall verification framework, namely the translator from Yul to CHCs, the Yul interpreter and the adaptations required in the specializer, we have not automated the overall process yet. Our current work is focused on implementing the full pipeline that connects all components as well as making a thorough experimental evaluation using real contracts.

**Acknowledgments.** This work was partially funded by: the Spanish MCI, Comunidad de Madrid, AEI and FEDER (EU) projects PID2021-122830OB-C41 and TEC-2024/COM-235, the Italian PNRR MUR project PE0000013-FAIR and the MUR PRIN 2022 Project DOMAIN (2022TSYYKJ, CUP B53D23013220006, PNRR M4.C2.1.1) funded by the European Union - NextGenerationEU. Emanuele De Angelis and Fabio Fioravanti are members of the INdAM-GNCS research group.

## References

1. Alexey Akhunov and Moody Salem. EIP-1153: Transient storage opcodes. `https://eips.ethereum.org/EIPS/eip-1153`, June 2018. Ethereum Improvement Proposals, no. 1153.
2. Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. SAFEVM: a safety verifier for Ethereum smart contracts. In *Proc. of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 386–389, 2019.
3. Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. Relaxed Effective Callback Freedom: A Parametric Correctness Condition for Sequential Modules With Callbacks. *IEEE Trans. Dependable Secur. Comput.*, 20(3):2256–2273, 2023.
4. Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. SolCMC: Solidity Compiler's Model Checker. In *Computer Aided Verification (CAV)*, volume 13371 of *LNCS*, pages 325–338. Springer, 2022.
5. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions via program specialization. *Science of Computer Programming*, 147:78–108, 2017.
6. Emanuele De Angelis, Fabio Fioravanti, John P. Gallagher, Manuel V. Hermenegildo, Alberto Pettorossi, and Maurizio Proietti. Analysis and Transformation of Constrained Horn Clauses for Program Verification. *Theory Pract. Log. Program.*, 22(6):974–1042, 2022.
7. Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. VeriMAP: A Tool for Verifying Programs through Transformations. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 568–574. Springer, 2014.

8. Emanuele De Angelis and Hari Govind Vediramana Krishnan. Competition of Solvers for Constrained Horn Clauses (CHC-COMP 2023). In *TOOLympics Challenge 2023*, pages 38–51, Cham, 2025. Springer Nature Switzerland.
9. Dxo, Mate Soos, Zoe Paraskevopoulou, Martin Lundfall, and Mikael Brockman. Hevm, a Fast Symbolic Execution Framework for EVM Bytecode. In *Computer Aided Verification (CAV)*, volume 14681 of *LNCS*, pages 453–465. Springer, 2024.
10. Yoshihiko Futamura. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *High. Order Symb. Comput.*, 12(4):381–391, 1999.
11. Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. Decompilation of Java bytecode to Prolog by partial evaluation. *Inf. Softw. Technol.*, 51(10):1409–1427, 2009.
12. Hossein Hojjat and Philipp Rümmer. The ELDARICA Horn Solver. In *Formal Methods in Computer Aided Design*, pages 1–7. IEEE, 2018.
13. Daniel Jackson, Chandrakana Nandi, and Mooly Sagiv. Certora Technology White Paper. https://docs.certora.com/en/latest/docs/whitepaper/index.html, 2022.
14. Joxan Jaffar and Michael J. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19-20:503–581, 1994. Special Issue: Ten Years of Logic Programming.
15. Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-Based Model Checking for Recursive Programs. In *Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 17–34. Springer, 2014.
16. Vasileios Koutavas, Yu-Yang Lin, and Nikos Tzevelekos. An Operational Semantics for Yul. In *Software Engineering and Formal Methods*, pages 328–346. Springer-Verlag, 2024.
17. OpenZeppelin. PaymentSplitter Contract. https://docs.openzeppelin.com/contracts/4.x/api/finance#PaymentSplitter.
18. Rodrigo Otoni, Matteo Marescotti, Leonardo Alt, Patrick Eugster, Antti Hyvärinen, and Natasha Sharygina. A Solicitous Approach to Smart Contract Verification. *ACM Trans. Priv. Secur.*, 26(2), March 2023.
19. Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Rosu. A formal verification tool for Ethereum VM bytecode. In *ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering*, pages 912–915. ACM, 2018.
20. Julio C. Peralta, John P. Gallagher, and Hüseyin Sağlam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In *Static Analysis Symposium*, volume 1503 of *LNCS*, pages 246–261. Springer, 1998.
21. Fabrice Rastello and Florent Bouchez-Tichadou, editors. *SSA-based Compiler Design*. Springer, 2022.
22. Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1678–1694, 2020.
23. Solidity Team. Solidity Documentation. Release 0.8.30. https://docs.soliditylang.org/_/downloads/en/v0.8.30/pdf/, May 2025.
24. Scott Wesley, Maria Christakis, Jorge A. Navas, Richard J. Trefler, Valentin Wüstholz, and Arie Gurfinkel. Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 13182 of *LNCS*, pages 425–449. Springer, 2022.
25. Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.