Visualizing CHC Verification Conditions for Smart Contracts Auditing

Marco Di Ianni[1,2], Fabio Fioravanti[1], and Giulia Matricardi[1,2]

[1]DEc, University of Chieti-Pescara, Italy
[2]Dottorato di Interesse Nazionale in Blockchain e DLT, University of Camerino, Italy

Smart contracts are blockchain programs that automatically enact certain agreements. Due to their immutable nature, vulnerabilities and bugs in smart contracts can cause significant financial losses and, thus, are often subjected to an auditing process, possibly supported by the use of formal methods In recent years many program analysis and verification methods and tools have been developed [1, 2] that use Constrained Horn Clauses (CHCs) [3], formulas in a fragment of first-order logic, as an intermediate language to represent the *verification conditions* (VCs) for a system and a property. so that, if the CHCs are satisfiable then the property under verification is valid. This approach provides great flexibility because efficient solvers, such as Z3/Spacer[1] and Eldarica[2], can be used to try to check CHC satisfiability (the problem is undecidable, in the general case). More recently, several tools for smart contract verification based on CHCs have been developed, such as SmartAce[3], Verysmart[4], Securify[5], eThor[6], HoRStify[7] and SolCMC [4], a module integrated in the Ethereum's Solidity compiler.

During smart contract auditing it is important for verification engineers to be able to inspect the VCs in CHC format. Unfortunately, these clauses are often difficult to be read by humans. In particular, if we consider the CHCs generated by SolCMC, we note some critical issues: i) CHCs are in SMT-LIB[8] format, a quite verbose format that uses prefix notation, that further hinders readability; ii) CHCs contain several clauses and predicates, that are generated to distinctly represent scenarios such as the success and failure (revert) of function calls, the relationship between the input and output of a function (function summaries), constructor and function initialization; 3) predicates may exhibit mutual dependencies, which may increase the difficulty of understanding the CHCs. Thus, finding a direct correspondence between the generated CHCs and the original Solidity source code can be challenging.

Starting from a Solidity smart contract with properties annotated using `require()` and `assert()`, via SolCMC we obtain a set of CHCs in SMT-LIB format, representing the VCs for the considered contract and properties. Then, in order to improve the readability of CHCs, we first use Eldarica to convert them from SMT-LIB to Prolog format. While still challenging, Prolog is more human-readable than SMT-LIB. After some textual processing, we use Logtalk[9] to generate a Graphviz[10] DOT file representing the Predicate Dependency Graph (PDG), whose nodes correspond to predicates and whose edges represent the predicates dependencies.

The following example shows a smart contract that implements a simple banking functionality allowing users to deposit and withdraw funds from their balance, stored in the contract, via the `deposit()` and `withdraw()` functions, respectively. The CHCs that are generated from this simple contract contain a significant number of clauses (about 30), and are not easily comprehensible.

Solidity contract

```
contract Bank {
    mapping (address => uint) balances;
    function deposit() external payable {
        uint user_balance = balances[msg.sender];
        balances[msg.sender] += msg.value;
        uint new_user_balance = balances[msg.sender];
        assert(new_user_balance == user_balance + msg.value);
    }
    function withdraw(uint amount) public {
        require(amount > 0 && amount <= balances[msg.sender]);
        balances[msg.sender] -= amount;
        (bool success,) = msg.sender.call{value: amount}("");
        require(success);
    }
}
```

[1] https://github.com/Z3Prover/z3
[2] https://github.com/uuverifiers/eldarica
[3] https://github.com/contract-ace/smartace
[4] https://github.com/kupl/VeriSmart-public
[5] https://github.com/eth-sri/securify2
[6] https://secpriv.wien/ethor/
[7] https://www.horstify.org/
[8] https://smtlib.cs.uiowa.edu/
[9] https://logtalk.org/
[10] https://graphviz.org

In the CHC clause extract, `select` and `store` are terms encoding read and write operations on the array associated with the `balances` map from addresses to address balances. Note also that clauses may contain redundant constraints, unnecessary variables and multiple occurrences of large constants.

CHC clause extract in Prolog format

```
...
block_10_function_deposit(A,B,C,D,E,F,G,H,I,J,K) :- \+(L),(A = 1),\+(((L; (M = N)), (\+((M = N)); \+(L)))),(N =<
    115792089237316195423570985008687907853269984665640564039457584007913129639935),(N >= 0),(N = (O + P)),(P =<
    115792089237316195423570985008687907853269984665640564039457584007913129639935),(P >= 0),(P = msg.value(E)),(O =<
    115792089237316195423570985008687907853269984665640564039457584007913129639935),(O >= 0),(O = J),(M =<
    115792089237316195423570985008687907853269984665640564039457584007913129639935),(M >= 0),(M = K),(K = Q),(Q >= 0),(Q =<
    115792089237316195423570985008687907853269984665640564039457584007913129639935),(Q =<
    115792089237316195423570985008687907853269984665640564039457584007913129639935),(Q >= 0),(Q = select(mapping(address =>
    uint256)_tuple_accessor_array(I), R)),(R =< 1461501637330902918203684832716283019655932542975),(R >= 0),(R = msg.sender(E)),(
    S = I),(T = I),(I = mapping(address => uint256)_tuple(store(mapping(address => uint256)_tuple_accessor_array(U), V, W),
    mapping(address => uint256)_tuple_accessor_length(U))),(X =<
    115792089237316195423570985008687907853269984665640564039457584007913129639935),(X >= 0),(X = select(mapping(address =>
    uint256)_tuple_accessor_array(U), V)),(U = Y),(W =<
    115792089237316195423570985008687907853269984665640564039457584007913129639935),(W >= 0),(W = (Z + A1)),(Z >= 0),(Z =<
    115792089237316195423570985008687907853269984665640564039457584007913129639935),(Z =<
    115792089237316195423570985008687907853269984665640564039457584007913129639935),(Z >= 0),(Z = select(mapping(address =>
    uint256)_tuple_accessor_array(Y), V)),(V =< 1461501637330902918203684832716283019655932542975),(V >= 0),(V = msg.sender(E)),(
    B1 = Y),(A1 =< 115792089237316195423570985008687907853269984665640564039457584007913129639935),(A1 >= 0),(A1 = msg.value(E))
    ,(J = C1),(C1 >= 0),(C1 =< 115792089237316195423570985008687907853269984665640564039457584007913129639935),(C1 =<
    115792089237316195423570985008687907853269984665640564039457584007913129639935),(C1 >= 0),(C1 = select(mapping(address =>
    uint256)_tuple_accessor_array(Y), D1)),(D1 =< 1461501637330902918203684832716283019655932542975),(D1 >= 0),(D1 = msg.sender(E
    )),(E1 = Y),(F1 = 0),(G1 = 0),block_8_deposit(H1,B,C,D,E,F,G,H,Y,G1,F1).
...
false :- error_target_4.
```

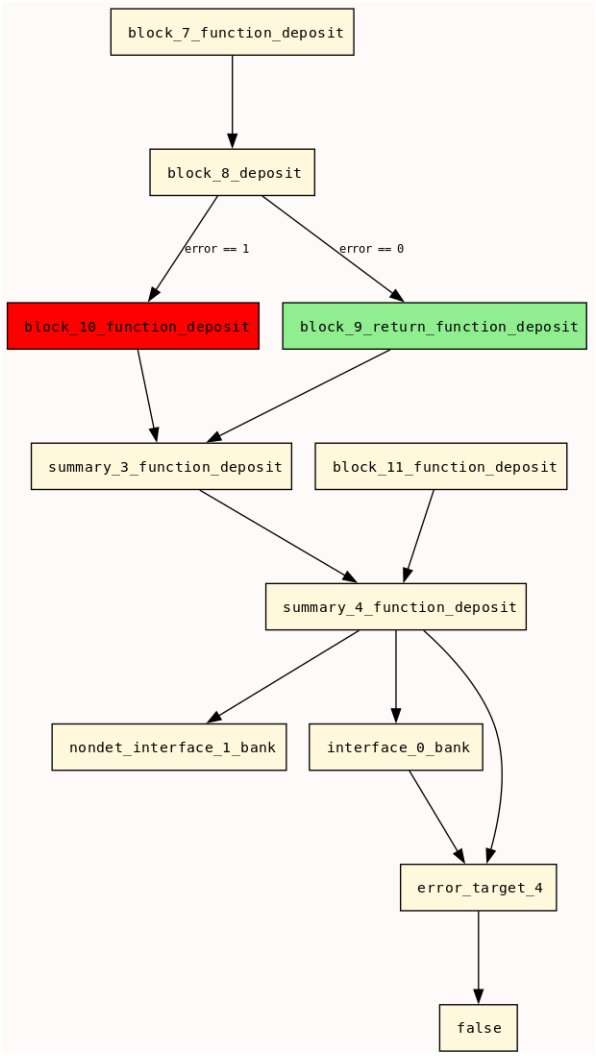

Figure 1: PDG for the `deposit()` function

Figure 1 shows the PDG corresponding to the `deposit` function of the example smart contract. Here we can see how the two cases of failure (revert identified by `block_10_function_deposit`) and success (identified by `block_9_return_function_deposit`) of a call to the function are handled. The `summary_4_function_deposit` predicate is used to keep track of the relationship between the function's input and output, derived from all its possible executions. This is linked to `error_target_4` which occurs in the query `false :- error_target_4` that is used to check the satisfiability of the CHCs.

Currently, the PDG representation is static and only allows to check the correspondence between predicate symbols and Solidity code, through manual review. However, we plan to develop a tool that makes the PDG visualization dynamic and user-friendly. Such a tool could allow direct interaction with graph nodes, allowing users to inspect and edit associated clauses, e.g. by using automated transformation methods, and offer selective visualisation options for particular sections of the PDG, giving auditors greater flexibility in their analytical approach and increasing their confidence in the results of formal verification. Furthermore, preserving variable names during conversion from SMT-LIB format to Prolog could help improve the accuracy and usability of the tool.

References

[1] E. De Angelis et al. "Analysis and transformation of constrained Horn clauses for program verification". In: *Theory and Practice of Logic Programming* 22.6 (2022), pp. 974–1042.

[2] A. Gurfinkel. "Program verification with constrained horn clauses". In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*. Ed. by S. Shoham and Y. Vizel. Vol. 13371. Lecture Notes in Computer Science. Springer, 2022, pp. 19–29.

[3] J. Jaffar and M. Maher. "Constraint logic programming: A survey". In: *Journal of Logic Programming* 19/20 (1994), pp. 503–581.

[4] Rodrigo Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023). ISSN: 2471-2566.