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

Abstract

Smart contracts are programs stored on a blockchain that can be used to automate agreements and transactions. Their immutable nature carries the risk of financial losses due to vulnerabilities and bugs. Consequently, verification processes, often supported by formal methods, are indispensable in mitigating these risks. Constrained Horn Clauses (CHCs) are commonly used for representing verification conditions (VCs) in smart contract analysis and verification, and several tools have been developed based on CHCs. Despite their utility in formal verification, CHCs pose a challenge in terms of human readability. In this paper we present preliminary work in the development of a tool aimed at visualizing CHCs, helping experts inspect VCs and their correspondence with smart contract code during the auditing process. The tool combines different translations in order to generate Predicate Dependency Graphs (PDGs), depicting the relationships among CHC predicates that have been derived from smart contract functions and the properties under verification. While the current representation of PDGs is static, future developments aim to render it dynamic and user-friendly. This enhancement would enable auditors to selectively display sections of the PDGs, interact with nodes and the associated clauses and code, apply CHC transformations and invoke CHC satisfiability solvers.

Keywords

Smart Contract Verification, Formal Verification, Constrained Horn Clauses, CHC visualization

1. Introduction

Originally introduced as the foundational technology for the Bitcoin [1] cryptocurrency, a blockchain is a decentralized, distributed ledger that securely records transactions in an immutable manner.

The blockchain is replicated on multiple nodes that use a consensus mechanism to confirm and add transactions to it. Blocks are cryptographically linked to each other, forming a secure chain that makes tampering difficult. This ensures the blockchain's integrity and immutability.

The potential applications of blockchain technology are numerous, ranging from financial services and supply chain management to healthcare and voting systems. Blockchains can also be used to store smart contracts [2], programs that automatically perform actions and transactions required by an agreement, upon the occurrence of certain conditions, without the need for an intermediary. Once registered on the blockchain, smart contracts become part of the distributed ledger and can be executed by all nodes in the network, guaranteeing the

CILC 2024: 39th Italian Conference on Computational Logic, June 26-28, 2024, Rome, Italy
marco.diianni@phd.unich.it (M. Di Ianni); fabio.fioravanti@unich.it (F. Fioravanti); giulia.matricardi@phd.unich.it (G. Matricardi)

reliability and consistency of transactions. Due to the immutable nature of smart contracts, the presence of a bug or vulnerabilities can lead to huge economic losses[1]. For these reasons, the analysis of the security of smart contracts is a very important topic, and several companies provide smart contract auditing services. The most popular platform for storing and executing smart contracts is Ethereum, overall managing billions of dollars. Smart contracts on Ethereum are typically written in Solidity, the platform's primary programming language. In order to execute smart contracts (or transactions) in Ethereum, it is necessary to pay a fee, called `gas`, which represents the computational cost of operations on this blockchain.

Many of the formal verification tools that have been developed for smart contract analysis use Constrained Horn Clauses (CHCs) (e.g. SeaHorn[2], RustHorn[3], JayHorn[4]) as an intermediate language to represent the verification conditions (VCs) for properties. However, for several reasons, CHCs are difficult to read and interpret (see Section 2). In this paper we present a prototype tool for CHC visualization that we have developed to aid developers analyse Solidity smart contracts. The tool takes as input Solidity smart contracts, annotated with properties, and produces a Predicate Dependency Graph (PDG), whose nodes correspond to predicates and whose edges represent the predicates dependencies.

The rest of the document is structured as follows: in Section 2 we provide some details of smart contract verification, with a focus on CHCs and auditing processes. In Section 3 we describe the CHC visualization tool and in Section 4, we demonstrate the tool's application to a basic smart contract, emphasizing the challenges in interpreting Horn clauses and the outcomes of transformations derived from the tool. In the last section we discuss related work and possible improvements.

2. Smart Contract Verification

Several tools have been developed over the past years to analyse the security of smart contracts, see [3] for a systematic survey. Most of these tools apply static analysis to detect common vulnerabilities and coding errors, such as reentrancy bugs, integer overflows, and unchecked external calls, and only a few tools apply formal verification in order to check the correctness of the program with respect to user provided specifications.

Properties for Solidity smart contracts can be specified in the form of Hoare triples using `require()` and `assert()` annotations. If a precondition, specified with `require()`, is satisfied at the beginning of code execution, and the code executes correctly, then a postcondition specified with `assert()` will be checked at the end of code execution. So, while the `assert()` function is used to test internal errors and verify invariants, the `require()` function is used to guarantee valid conditions, such as correct inputs, contract status variables or the return of valid values from calls to external contracts.

Ethereum smart contracts can also fail for other reasons. For instance, transactions can fail due to the lack of a sufficient amount of gas for code execution. The failure of a transaction may

[1]Approximately $773 million in 2023.
Source: https://www.slowmist.com/report/2023-Blockchain-Security-and-AML-Annual-Report(EN).pdf
[2]https://seahorn.github.io/
[3]https://github.com/hopv/rust-horn
[4]https://jayhorn.github.io/jayhorn/

result in several consequences, including the loss of the funds invested in the transaction, the gas fees paid to execute the transaction and, in some cases, the execution of any transactions preceding the failing transaction may be reverted.

Many recent program analysis and verification methods [4, 5] use Constrained Horn Clauses (CHCs) [6] as an intermediate language for specifying (i) the semantics of programs, that can be written in a variety of programming languages, including imperative, functional, object-oriented, and concurrent ones, and (ii) program properties, including safety, termination, and program equivalence. CHCs are a fragment of First-Order Logic with the same syntax and semantics of Constraint Logic Programs (CLP) but that are not intended to be directly executed as programs.

CHCs provide a formal framework for expressing verification conditions (VCs), allowing developers to specify properties and constraints that smart contracts must satisfy. One of the strengths of CHCs lies in their ability to capture complex program behaviors and dependencies and summarize them into a standardized framework. This formal verification process provides mathematical certainty about the behavior of smart contracts, strengthening confidence in their security and reliability. Several tools for smart contract verification based on CHCs have been developed, such as SmartAce [7], Verysmart [8], Securify [9], *e*Thor [10], HoRStify [11] and SolCMC [12], a module integrated in the Ethereum's Solidity compiler.

Smart contracts are often manually reviewed by expert auditors. Auditing aims to identify and assess risks, vulnerabilities, compliance issues and technical defects in the smart contract.

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[5] format, a quite verbose format that uses prefix notation, that further hinders readability, (ii) CHCs contain several clauses and predicates that are generated to represent different scenarios, such as the success and failure of function calls, the relationship between the input and output of a function (function summaries), constructor and function initialization, (iii) 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. Therefore, there is a need for a tool to assist verification engineers in graphically inspecting the VCs in CHC format.

3. CHC Visualization

Providing a visual representation of the dependencies that exist among the CHC predicates can be useful during the auditing process to ease the understanding of the smart contract behaviour and increase the confidence in the correctness of the VC encoding. We have developed *CHCViz*, a tool that takes as input a Solidity smart contract and a property, specified using `require()` and `assert()` annotations, and generates a Predicate Dependency Graph (PDG), whose nodes correspond to CHC predicates and whose edges represent the dependencies among predicates.

The tool has been implemented as a Python script that coordinates the execution of the following stages:

[5]https://smtlib.cs.uiowa.edu/

- **CHC Generation**: generation of CHCs in SMT-LIB format from the Solidity smart contract;
- **Prolog encoding**: translation of the CHCs from the SMT-LIB format to Prolog format;
- **PDG creation**: creation of the PDG in the Graphviz DOT format;
- **SVG generation**: generation of the PDG in Scalable Vector Graphics (SVG) format.

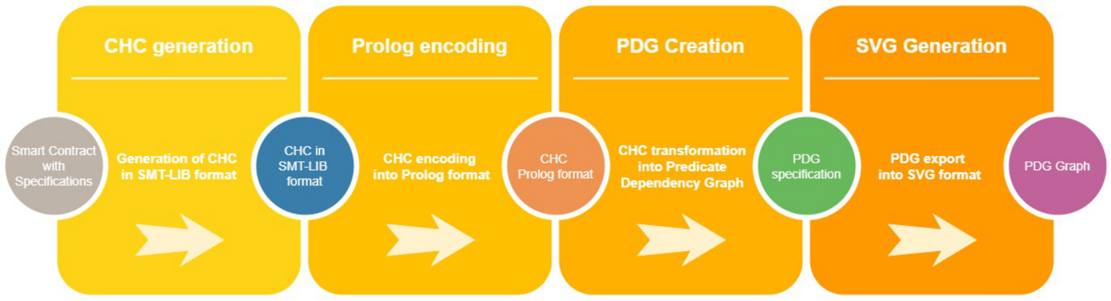

CHC generation
Prolog encoding
PDG Creation
SVG Generation
Smart Contract with Specifications
Generation of CHC in SMT-LIB format
CHC in SMT-LIB format
CHC encoding into Prolog format
CHC Prolog format
CHC transformation into Predicate Dependency Graph
PDG specification
PDG export into SVG format
PDG Graph

Figure 1: Tool Architecture

CHC Generation The process initiates with the generation of the CHCs in SMT-LIB format from the annotated Solidity contract. The CHC generation is performed by invoking the SMTChecker module, also known as SolCMC, of the Solidity compiler[6] by specifying `assert` as the only verification target. The satisfiability of the generated CHCs encode the validity of the property for the contract: if the CHCs are satisfiable then the property is valid.

Prolog Encoding After the generation of CHCs in SMT-LIB format, the subsequent phase involves their translation into Prolog format using Eldarica[7]. The Prolog format is less verbose and more human readable than SMT-LIB and is also the input format required by the tool used in the subsequent phase.

PDG Creation In this phase, taking as input the CHCs in Prolog format generated by Eldarica, we invoke Logtalk[8], a Prolog-based declarative object-oriented logic programming language, for creating the PDG specification that encodes the dependencies among the CHC predicates. The PDG is generated as a GraphViz[9] DOT file containing all the information needed for displaying the graph. The DOT language is very flexible: it supports different attributes of nodes, edges, graphs, and various layout engines and output formats.

SVG Generation In the final phase, the PDG is transformed into an SVG file format using Graphviz. The SVG format has been chosen among other formats, because of its scalability and of its support for interactivity through JavaScript and CSS.

We have run *CHCViz* on 187 contracts taken from [13] and we have obtained the following results. For 174 contracts the PDGs in SVG format were successfully generated, for 10 contracts the CHCs in SMT-LIB format generated by SolCMC contained data type declaration errors, which were manually corrected to subsequently achieve successful SVG generation, and for

[6]SolCMC (version 0.8.25) https://docs.soliditylang.org/en/latest/smtchecker.html
[7]Eldarica (version 2.0.9) https://github.com/uuverifiers/eldarica
[8]Logtalk (version 3.75.0) https://logtalk.org/
[9]GraphViz (version 2.43.0) https://graphviz.org

3 contracts, our tool was unable to generate the DOT file due to a limitation of the Prolog encoder. The *CHCViz* tool, the supplementary material and our experiments are available at https://fmlab.unich.it/chcviz.

4. Visualizing the CHCs for a simple Smart Contract

We now consider a smart contract that implements a simple banking functionality (Listing 1) allowing users to deposit and withdraw funds from their balance, stored in the contract, via the `deposit()` and `withdraw()` functions, respectively. The address of the sender of the message initiating the transaction and the amount of wei, the smallest subunit of Ether (the native cryptocurrency of the Ethereum), sent in the transaction are denoted by `msg.sender` and `msg.value`, respectively. The contract contains `require()` and `assert()` annotations to specify properties stating that (i) for the `deposit()` function: the balance of a user after a deposit must be equal to the previous balance increased by the value specified in the transaction (line 8), (ii) for the `withdraw()` function: (a) in order to withdraw a specific amount from a balance, the specified value must be greater than zero and not greater than the balance of the user requesting the withdrawal (line 12), (b) the return of valid values from external calls is also checked (line 15). The VCs that are generated from this simple contract contain a significant number of CHCs (about 30), and are not easily readable.

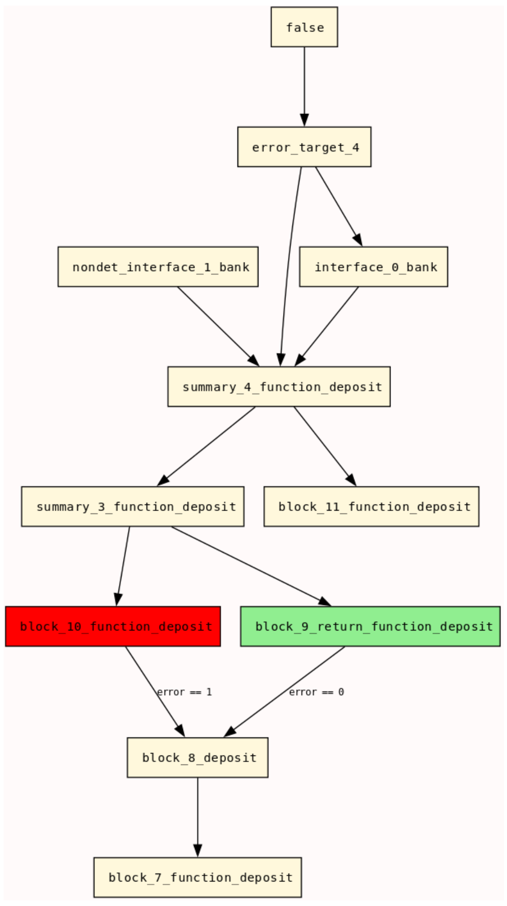
```
1   contract Bank {
2      mapping (address => uint) balances;
3
4      function deposit() external payable {
5         uint user_balance = balances[msg.sender];
6         balances[msg.sender] += msg.value;
7         uint new_user_balance = balances[msg.sender];
8         assert(new_user_balance == user_balance + msg.value);
9      }
10
11     function withdraw(uint amount) public {
12        require(amount > 0 && amount <= balances[msg.sender]);
13        balances[msg.sender] -= amount;
14        (bool success,) = msg.sender.call{value: amount}("");
15        require(success);
16     }
17  }
```

Listing 1. Solidity Contract: a solidity smart contract, named `Bank`, that implements a simple banking functionality.

An extract of a single CHC clause in Prolog format obtained after the CHC Generation and Prolog encoding phases is shown in Listing 2 (you can see the same clause in SMT-LIB format in the Appendix).

```
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 =< 115...935),(N >= 0),(N = (O + P)),(P =< 115...935),(P >= 0),(P = msg.value(E)
    ),(O =< 115...935),(O >= 0),(O = J),(M =< 115...935),(M >= 0),(M = K),(K = Q),(Q >= 0),(Q =<
    115...935),(Q >= 0),(Q = select(mapping(address => uint256)_tuple_accessor_array(I), R)),(R
    =< 146...975),(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 =< 115...935),(X >= 0),(X = select(mapping(address =>
    uint256)_tuple_accessor_array(U), V)),(U = Y),(W =< 115...935),(W >= 0),(W = (Z + A1)),(Z >=
    0),(Z =< 115...935),(Z =< 115...935),(Z >= 0),(Z = select(mapping(address => uint256)
    _tuple_accessor_array(Y), V)),(V =< 146...975),(V >= 0),(V = msg.sender(E)),(B1 = Y),(A1 =<
    115...935),(A1 >= 0),(A1 = msg.value(E)),(J = C1),(C1 >= 0),(C1 =< 115...935),(C1 =<
    115...935),(C1 >= 0),(C1 = select(mapping(address => uint256)_tuple_accessor_array(Y), D1)),(
    D1 =< 146...975),(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).
```

Listing 2. CHC clause extract: a clause in Prolog format.



false
error_target_4
nondet_interface_1_bank
interface_0_bank
summary_4_function_deposit
summary_3_function_deposit
block_11_function_deposit
block_10_function_deposit
block_9_return_function_deposit
error == 1
error == 0
block_8_deposit
block_7_function_deposit

Figure 2: PDG for the `deposit()` function

In the clause shown in Listing 2, (i) 115...935 and 146...975 denote integers with 78 and 49 digits, respectively, corresponding to the maximum allowed value for a transaction and for an account address, and (ii) `select` and `store` are terms encoding read and write operations on the array associated with the `balances` variable, that maps addresses to the corresponding balances. Note also that clauses may contain redundant constraints, unnecessary variables and multiple occurrences of large constants. Figure 2 shows the PDG corresponding to the `deposit()` function of the example smart contract. There we can see how the two possible outcomes of a function call are handled: the case of failure, causing the transaction to revert, is identified by `block_10_function_deposit` while the case of success is identified by `block_9_return_function_deposit`. 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. The complete PDG for the Bank contract is shown in the appendix.

5. Related Work and Conclusions

Some other tools focus on the visualization of smart contracts on the Ethereum platform, such as Surya[10], Solgraph[11] and Solidity Auditor[12] which generate Control Flow Graphs (CFG), Smart-Graph [14], that generates UML diagrams from Solidity source code, and SmartInspect [15] for visualizing the contract stored state. Other approaches for visualising blockchain and smart contracts are compared in [16]. Our approach differs from the above mentioned works in that we focus on the visual representation of the CHCs derived from the smart contract, not on the smart contract itself.

Ensuring the correctness of smart contracts poses a significant challenge, requiring auditors to employ supportive tools for effective verification. In addition to automated verification tools, CHC visualization tools can be a valuable aid in comprehending predicate dependencies, and increase the confidence in the correctness of the encoding of the considered property.

In this paper, we have presented a prototype tool aimed at visualizing the dependencies among the CHCs derived from Solidity contracts. The tool can be extended and improved along several directions. The CHC generation module can be extended to employ translation schemes based on different operational semantics [17], or for programs written in different programming languages (for instance, Seahorn could be used to generate CHCs for LLVM-based languages). The Prolog Encoding and the PDG Creation modules could be improved or rewritten in order to have better control over the content of the output DOT file. For instance, it could be useful to annotate the CHCs with the source code from which they are generated, enabling a clearer understanding of the correspondence between code and clauses. Other improvements include the possibility of interacting with the PDG for exploring the CHCs (e.g. expanding, collapsing, browsing) and invoking CHC solvers to check the validity of the property. Furthermore, the tool could evolve towards a GUI for VeriMAP [18] for selectively applying satisfiability-preserving CHC transformations with the aim of improving solver effectiveness. In this case, it would be essential to keep track of the applied transformations. In this way, verification engineers could experiment with different optimization strategies while maintaining the ability to revert to previous states, ensuring both agility and reliability. In conclusion, by providing advanced visualization and analysis capabilities, the tool could help auditors to more easily detect specification violations and improve the overall reliability of smart contracts.

Acknowledgments
We thank Emanuele De Angelis, Alberto Pettorossi and Maurizio Proietti for useful discussions. This work has been partially supported by the project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU, and subproject SMARTK (CUP D23C24000220006). Fabio Fioravanti is member of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

[10]https://github.com/ConsenSys/surya
[11]https://github.com/raineorshine/solgraph
[12]https://github.com/Consensys/vscode-solidity-auditor

References

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2008).

[2] N. Szabo, Formalizing and securing relationships on public networks, First monday (1997).

[3] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, H.-N. Lee, Ethereum smart contract analysis tools: A systematic review, IEEE Access 10 (2022) 57037–57062.

[4] E. D. Angelis, F. Fioravanti, J. P. Gallagher, M. V. Hermenegildo, A. Pettorossi, M. Proietti, Analysis and transformation of constrained Horn clauses for program verification, Theory and Practice of Logic Programming 22 (2022) 974–1042.

[5] A. Gurfinkel, Program verification with constrained Horn clauses, in: Computer Aided Verification - 34th International Conference, Proceedings, Part I, Springer, 2022.

[6] J. Jaffar, M. Maher, Constraint logic programming: A survey, Journal of Logic Programming 19/20 (1994) 503–581.

[7] S. Wesley, M. Christakis, J. A. Navas, R. Trefler, V. Wüstholz, A. Gurfinkel, Verifying solidity smart contracts via communication abstraction in smartace, in: Verification, Model Checking and Abstract Interpretation, Springer International Publishing, 2022.

[8] S. So, M. Lee, J. Park, H. Lee, H. Oh, Verismart: A highly precise safety verifier for Ethereum smart contracts, in: 2020 IEEE Symposium on Security and Privacy (SP), IEEE, 2020.

[9] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, M. Vechev, Securify: Practical security analysis of smart contracts, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, 2018.

[10] C. Schneidewind, I. Grishchenko, M. Scherer, M. Maffei, eThor: Practical and provably sound static analysis of Ethereum smart contracts, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 621–640.

[11] S. Holler, S. Biewer, C. Schneidewind, Horstify: Sound security analysis of smart contracts, in: 2023 IEEE 36th Computer Security Foundations Symposium (CSF), IEEE, 2023.

[12] R. Otoni, M. Marescotti, L. Alt, P. Eugster, A. Hyvärinen, N. Sharygina, A solicitous approach to smart contract verification, ACM Trans. Priv. Secur. 26 (2023).

[13] M. Bartoletti, F. Fioravanti, G. Matricardi, R. Pettinau, F. Sainas, Towards Benchmarking of Solidity Verification Tools, in: 5th International Workshop on Formal Methods for Blockchains (FMBC 2024), Open Access Series in Informatics (OASIcs), 2024, pp. 6:1–6:15.

[14] A. Zhukov, V. Korkhov, Smartgraph: Static analysis tool for solidity smart contracts, in: Computational Science and Its Applications – ICCSA Workshops, Springer, 2023.

[15] S. Bragagnolo, H. Rocha, M. Denker, S. Ducasse, Smartinspect: solidity smart contract inspector, in: 2018 International workshop on blockchain oriented software engineering (IWBOSE), Ieee, 2018, pp. 9–18.

[16] F. Härer, H.-G. Fill, A comparison of approaches for visualizing blockchains and smart contracts, Jusletter IT (2019).

[17] E. De Angelis, F. Fioravanti, A. Pettorossi, M. Proietti, Semantics-based generation of verification conditions via program specialization, Sci. Comput. Program. 147 (2017) 78–108.

[18] E. De Angelis, F. Fioravanti, A. Pettorossi, M. Proietti, VeriMAP: A tool for verifying programs through transformations, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2014, pp. 568–574.

6. Appendix

In Listing 3 a CHC clause in SMT-LIB format is shown. You can see the same clause in Prolog format in Listing 1. This is to highlight the differences in verbosity and readability between the Prolog and SMT-LIB formats.

CHC clause extract in SMT-LIB format

```
(declare-fun |block_10_receive_39_82_0| (Int Int |abi_type| |crypto_type| |tx_type| |state_type| |mapping(address
    => uint256)_tuple| |state_type| |mapping(address => uint256)_tuple| Int Int ) Bool)
(assert
(forall ((abi_0 |abi_type|) (amount_41_0 Int) (amount_41_1 Int) (balances_4_length_pair_0 |mapping(address =>
    uint256)_tuple|) (balances_4_length_pair_1 |mapping(address => uint256)_tuple|) (balances_4_length_pair_2 |
    mapping(address => uint256)_tuple|) (crypto_0 |crypto_type|) (error_0 Int) (error_1 Int) (error_2 Int) (
    expr_11_1 Int) (expr_12_1 Int) (expr_14_length_pair_0 |mapping(address => uint256)_tuple|) (
    expr_14_length_pair_1 |mapping(address => uint256)_tuple|) (expr_14_length_pair_2 |mapping(address => uint256)
    _tuple|) (expr_16_1 Int) (expr_17_1 Int) (expr_17_2 Int) (expr_19_1 Int) (expr_20_1 Int) (
    expr_24_length_pair_0 |mapping(address => uint256)_tuple|) (expr_26_1 Int) (expr_27_1 Int) (expr_30_0 Int) (
    expr_31_0 Int) (expr_33_1 Int) (expr_34_1 Int) (expr_35_1 Bool) (expr_9_length_pair_0 |mapping(address =>
    uint256)_tuple|) (new_user_balance_23_0 Int) (new_user_balance_23_1 Int) (new_user_balance_23_2 Int) (
    old_user_balance_8_0 Int) (old_user_balance_8_1 Int) (old_user_balance_8_2 Int) (state_0 |state_type|) (
    state_1 |state_type|) (state_2 |state_type|) (success_67_1 Bool) (this_0 Int) (tx_0 |tx_type|))
(=> (and (and (block_8__38_82_0 error_0 this_0 abi_0 crypto_0 tx_0 state_0 balances_4_length_pair_0 state_1
    balances_4_length_pair_1 old_user_balance_8_1 new_user_balance_23_1) (and (= expr_35_1 (= expr_30_0 expr_34_1)
    ) (and (=> true (and (>= expr_34_1 0) (<= expr_34_1 115...935))) (and (= expr_34_1 (+ expr_31_0 expr_33_1)) (
    and (=> true (and (>= expr_33_1 0) (<= expr_33_1 115...935))) (and (= expr_33_1 (|msg.value| tx_0)) (and (=>
    true (and (>= expr_31_0 0) (<= expr_31_0 115...935))) (and (= expr_31_0 old_user_balance_8_2) (and (=> true (
    and (>= expr_30_0 0) (<= expr_30_0 115...935))) (and (= expr_30_0 new_user_balance_23_2) (and (=
    new_user_balance_23_2 expr_27_1) (and (and (>= expr_27_1 0) (<= expr_27_1 115...935)) (and (=> true (and (>=
    expr_27_1 0) (<= expr_27_1 115...935)))
(and (= expr_27_1 (select (|mapping(address => uint256)_tuple_accessor_array| balances_4_length_pair_2) expr_26_1)
    ) (and (=> true (and (>= expr_26_1 0) (<= expr_26_1 146...975))) (and (= expr_26_1 (|msg.sender| tx_0)) (and
    (= expr_24_length_pair_0 balances_4_length_pair_2) (and (= expr_14_length_pair_2 balances_4_length_pair_2) (
    and (= balances_4_length_pair_2 (|mapping(address => uint256)_tuple| (store (|mapping(address => uint256)
    _tuple_accessor_array| expr_14_length_pair_1) expr_16_1 expr_20_1) (|mapping(address => uint256)
    _tuple_accessor_length| expr_14_length_pair_1))) (and (=> true (and (>= expr_17_2 0) (<= expr_17_2 115...935))
    ) (and (= expr_17_2 (select (|mapping(address => uint256)_tuple_accessor_array| expr_14_length_pair_1)
    expr_16_1)) (and (= expr_14_length_pair_1 balances_4_length_pair_1) (and (=> true (and (>= expr_20_1 0) (<=
    expr_20_1 115...935))) (and (= expr_20_1 (+ expr_17_1 expr_19_1)) (and (and (>= expr_17_1 0) (<= expr_17_1
    115...935)) (and (=> true (and (>= expr_17_1 0) (<= expr_17_1 115...935))) (and (= expr_17_1 (select (|mapping
    (address => uint256)_tuple_accessor_array| balances_4_length_pair_1) expr_16_1)) (and (=> true (and (>=
    expr_16_1 0) (<= expr_16_1 146...975))) (and (= expr_16_1 (|msg.sender| tx_0)) (and (= expr_14_length_pair_0
    balances_4_length_pair_1) (and (=> true (and (>= expr_19_1 0) (<= expr_19_1 115...935))) (and (= expr_19_1 (|
    msg.value| tx_0)) (and (= old_user_balance_8_2 expr_12_1) (and (and (>= expr_12_1 0) (<= expr_12_1 115...935))
     (and (=> true (and (>= expr_12_1 0) (<= expr_12_1 115...935))) (and (= expr_12_1 (select (|mapping(address =>
     uint256)_tuple_accessor_array| balances_4_length_pair_1) expr_11_1)) (and (=> true (and (>= expr_11_1 0) (<=
    expr_11_1 146...975))) (and (= expr_11_1 (|msg.sender| tx_0)) (and (= expr_9_length_pair_0
    balances_4_length_pair_1) (and (= new_user_balance_23_1 0) (and (= old_user_balance_8_1 0) true)))))))))))))))
    ))))))))))))))))))))))))) (and (and true (not expr_35_1)) (= error_1 1))) (block_10_receive_39_82_0 error_1
    this_0 abi_0 crypto_0 tx_0 state_0 balances_4_length_pair_0 state_1 balances_4_length_pair_2
    old_user_balance_8_2 new_user_balance_23_2))))
```

Listing 3. CHC clause extract: a clause in SMT-LIB format

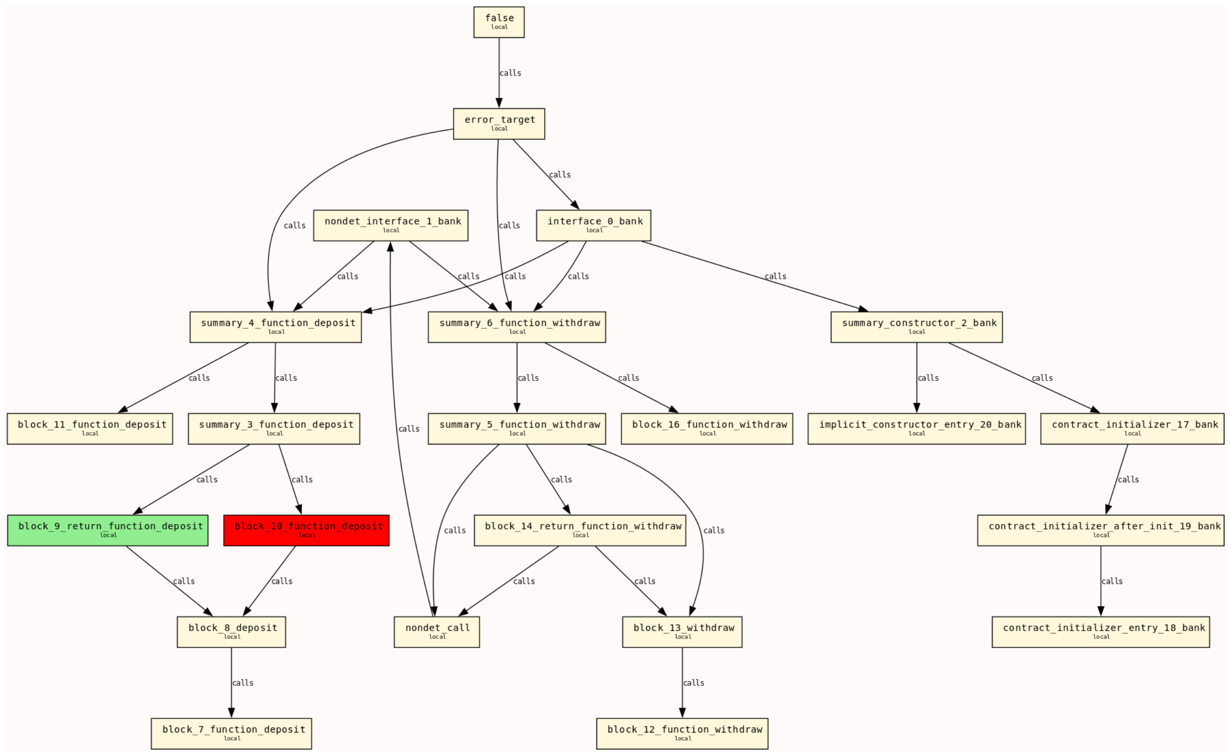

false
error_target
nondet_interface_1_bank
interface_0_bank
summary_4_function_deposit
summary_6_function_withdraw
summary_constructor_2_bank
block_11_function_deposit
summary_3_function_deposit
summary_5_function_withdraw
block_16_function_withdraw
implicit_constructor_entry_20_bank
contract_initializer_17_bank
block_9_return_function_deposit
block_10_function_deposit
block_14_return_function_withdraw
contract_initializer_after_init_19_bank
block_8_deposit
nondet_call
block_13_withdraw
contract_initializer_entry_18_bank
block_7_function_deposit
block_12_function_withdraw
local
calls

Figure 3: Complete PDG for the Bank contract

The complete PDG for the Bank contract is shown in Figure 3. Despite the modest complexity of the contract, it is surprising to observe the size of the generated graph. In addition, blocks relating to the initialisation of the contract and the implicit constructor are also included, adding further complexity and detail to the graphical representation.