A Tour of Formal Tools & Methods for Blockchain

Everett Hildenbrandt Kurt Barry Tannr Allard Tiago Barbosa

MakerDAO

1 Introduction

There are many previous and on-going studies on smart contract verification, branching in methodology, mathematical structures, logic and implementation efforts. This section describes, in great detail, different approaches in the literature, accentuating trends and challenges, characterizing their association with the proposed framework presented in this paper.

1.1 Methodology

For clarity purposes this paper will answer the following research questions (RQ's):

RQ1: What are the formal methodologies that are utilized in the verification of semantic properties in smart contracts?

RQ2: How are specifications or program properties formalized?

RQ3: What types of specifications do these formal methodologies aim to verify?

RQ4: What is the quality of the solutions?

In pursuance of full exploration of papers published related to smart contract formal specification and verification, we queried multiple publication databases using combinations of keywords such as "smart contracts, formal verification, semantic properties, modeling". This databases were Google Scholar, Web of Knowledge and DBLP. In addition Github repositories such as *Smart Contract Languages* and *Ethereum Formal Verification* also contain organized lists of material, some of which is redundant but nevertheless there is a lot of information that hasn't yet been formalized on publications and is relevant as well. We also resorted to other research and survey papers [3, 39, 41, 44] for an expeditious overview on the literature.

The following sub-sections clarify the research questions.

1.2 Results

1.2.1 RQ1: What are the formal methodologies that are utilized in the verification of semantic properties in smart contracts?

Before deep diving into each methodology we present a taxonomy of the literature.

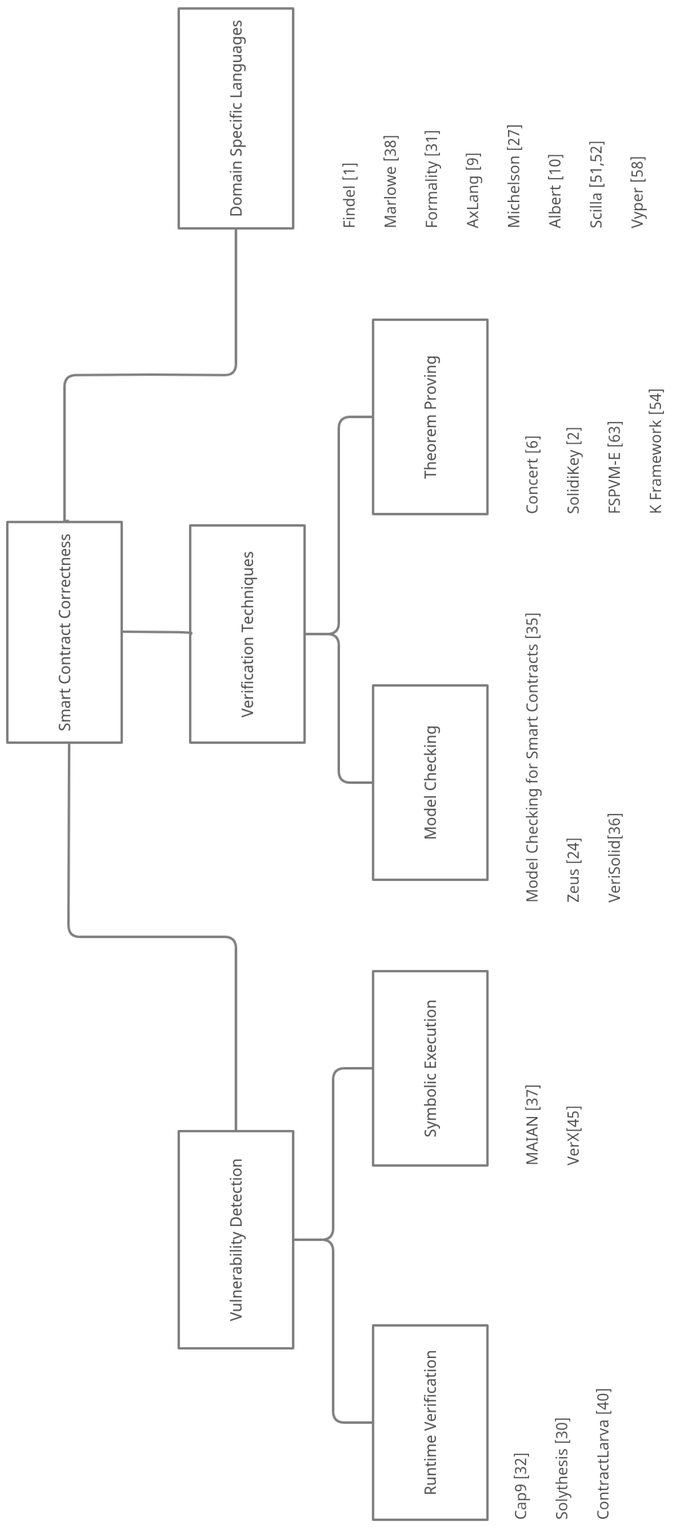

Fig. 1. Taxonomy of Frameworks and Tools

1.2.1.1 Verification Techniques

The procedure to formally verify a program goes as follows: choose a suitable mathematical structure and formalize the system, then model the desired specifications and semantic properties and lastly prove that the formalization satisfies the model of specifications. Formal verification tools strive for the usage of simple human-readable languages for these models and specifications in such a way that results are clear and concise.

Different formal verification techniques have different automation levels, ranging from interactive to automatic, as well as different required operation expertise levels and different property correctness degrees. Based on the literature we will now describe these different approaches:

Model Checking Model checking formalizes a system using a finite-state model. After also formalizing the desired property, an automated model checking software such as NuSMV [9] or SPIN [13] is able to check whether the model satisfies the property by verifying if the same holds for every state of the model. In case of failure the model checker software is able to provide a counter-example for easier bug finding and patching. If success is achieved the system is formally verified for said property. Model checking is quite popular in the literature [41] mainly due to the existence of automated frameworks and suitability of the modelling, although it's considered to be limited [44] as its performance is bottle-necked by the input language of the automatic model checker and the state explosion problem [25]. This approach is often modeled as a contract-level formalization with the use of temporal logic.

Model checking for Smart Contracts [25] (MSCS) proposes a model that is specified in NuSMV input language and the specifications to verify are formalized in Computation Tree Logics, an extension on temporal logic. The model's architecture has 3 layers: the kernel layer that specifies the Ethereum blockchain operations, the application layer that models the smart contracts, and the environment layer that is tuned for each use-case. It accepts a wide variety of properties such as safety, liveness, fairness, reachability, functional correctness and real-time properties. The paper's authors recognize MCSC limits as being a tool only for Ethereum and a lack of precision due to NuSMV input language limited expressiveness.

ZEUS [17] is a framework for automatic formal verification that uses abstract interpretation and symbolic model checking for correctness verification. It consists of a policy builder, a source code translator and a verifier. The user interacts with the policy builder for specifying the desired properties in eXtensible Access Control Markup Language. The source code translator then translates a contract and its properties into a low-level intermediate language that encodes its execution semantics. Lastly, the verifier performs a static analysis to find where the verification predicates need to be asserted and by using Constrained Horn Clauses the verification can be quickly performed by SMT solvers.

VeriSolid [26] is an open source framework that is built using **FSolidM** and has a web interface using WebGME. The goal of VeriSolid is to provide a correct-by-design construction of Solidity smart contracts. The formalization model used is the Behavior-Interaction-Priority model which is a two layer framework for the interactions of the smart contract and its users. The Behaviour layer describes the logic of smart contracts as a Finite State Model and the Interaction layer defines how they communicate, allowing to compositionally model and verify systems of multiple interacting contracts. Properties are formalized and then automatically translated to Computational Tree Logic. Furthermore translation from BIP to NuSMV allows for the verification of the model and if all properties are proven to be satisfied then VeriSolid automatically generates an equivalent Solidity code. The friendly user interface and automation makes this tool an easier option for inexperienced users in formal verification.

Theorem Proving Theorem proving begins with modeling a system mathematically, using a suitable logic with well defined axioms and inference rules. Specifications of semantic properties are also formalized typically using logics, such as Hoare or Reachability Logics for full property analysis. The application of the logic derives new theorems and lemmas than can be then used to prove the desired system's semantic properties. Contrasting with other methods, such as model checking and symbolic execution, when a theorem prover verifies a property it is verified for all possible infinite states of the system. Different logics can be applied but the most common ones are High Order Logic [24], used by Isabelle [45], and Calculus of Inductive Constructions [33], used by Coq [42]. Albeit theorem proving is sound and complete, its use is constrained by the in-depth knowledge and expertise required by the programmer as theorem provers are generally interactive, requiring the user to do a large fraction of the verification. Automation of verification is usually accomplished by resorting to previously built tactics and is a topic of high interest.

Concert [5] is a framework for meta-theoretical and functional reasoning about functional programming languages. Using deep embedding for formalization of the system and Hoare Logic for specifying properties this Coq [42] based framework allows verification of arbitrary specifications. This framework also enables verification of properties in smart contract interactions using shallow embedding to formalize the blockchain underlying behaviour and is broad enough to be used for distinct languages and blockchains. The authors prove their translation embedding soundness but still have yet developed a mechanism that allows reasoning about gas which is considered as upcoming work.

SolidiKey [2] translates smart contracts to Java and formalizes them in dynamic logic, an extension of First Order Logic. The theorem prover used is KeY [10], a framework for deductive verification, which has been developed for proving correctness of Java programs since 1998. Dynamic logic property formalization is very similar to Hoare style triples but the main difference is that sequent calculus is used to reason about formulas. Specifications are then annotated as invariants in the smart contract and the SolidiKey framework subsequently attempts to prove automatically the specified invariants, making an effort to reduce user interaction in the proof. Constraints of this tool, and succeeding work, are the limited model of EVM's memory management, unlike Solidity which uses reference semantics for all assignments, and a lack of trusted translator.

FSPVM-E [46] is a formal symbolic virtual machine (FSPVM) for Ethereum smart contracts (E). FSPVM-E attempts to automatize theorem proving introducing EVI, execution verification isomorphism, which is an extension of CHI, Curry-Howard isomorphism, which combines the completeness provided by theorem proving and automation provided by symbolic execution. FSPVM-E follows a von Newman architecture and so is constituted by three modular parts: GERM, a formal memory framework that supports specifications at code level, Lolisa, an extensible formal intermediate language equivalent to Solidity formalized in Coq and FEther, a formally verified interpreter for Lolisa. While proving properties these three parts work together as GERM simulates the virtual memory of the EVM, Lolisa represents a formalized Solidity syntax, and the FEther execution engine that symbolically executes and verifies the formal model of Lolisa smart contracts. Figure **??** shows a visual explanation of this architecture which intuitively conveys its modularity and re usability. Lolisa exhibits Lolisa is the first implementation of Solidity in Coq to use GADTs, generalized algebric datatypes, and includes all built-in functions of the EVM. The self-correctness of these three components is also verified in Coq itself. Upcoming work will focus on programming in Coq a formally verified translator from Solidity to Lolisa and further generalization of the tool for usage in other blockchains.

K Framework [40] is a complete framework for designing and developing formal language semantics for intuitive reasoning about properties of programs written in any formalized language. Based on matching and reachability logic the K framework has been used to formalize semantics of EVM [32] and Solidity [16], which enables full verification of programs, including gas and memory states details. Semantic properties are formalized as both pre and postconditions, along with reachability claims and reasoning on proofs uses matching logic [36]. Recently the K framework has been used to formally verify Ethereum's 2.0 deposit contract [31] for the upcoming change in consensus mechanism from Proof-of-Work to Proof-of-Stake. This work successfully found bugs and errors in the Vyper language that were unknown until then and successfully proved as well the desired properties of the Deposit Contract. Recently the deductive verifier tool Firefly [15], based on the K framework, has been developed by Runtime Verification Inc. for the verification of Ethereum's smart contracts. This tool uses an hybrid approach to theorem proving, availing the use of symbolic and concrete execution by SMT solvers, as an attempt to reduce the needed interaction in proving specifications.

1.2.1.2 Vulnerability Detection

Symbolic Execution Symbolic execution executes a program replacing the concrete value of program variables as symbolic expressions, in order to discover all its possible execution paths. This method represents the execution of a program as a Control Flow Graph, and in a smart contract it is constructed from its bytecode. Symbolic executions commonly only explores paths up to a threshold as the method suffers from the same path and state explosion constraints as Model Checking. Formalization of specifications tend to be known execution vulnerability patterns. With the Control Flow Graph and formalization of specifications these can then be encoded and checked by an SMT solver such as Z3 [11]. Problems in smart contract symbolic execution occur due to regular usage of hash functions, which handicaps SMT provers and the need for a symbolical model of memory management and inter smart contract communication. As with model checking if the SMT solver does not find any error then the contract is considered verified, if an error is found then the SMT prover provides a counter-example which allows for faster debugging.

MAIAN [27] is a dynamic tool built for vulnerability finding across large sequences of contract calls. There's no need for a formalization of the source code since it finds vulnerabilities directly from EVM bytecode. Formal specifications are inflexible and categorized as greedy, prodigal and suicidal. These specifications are all quite trivial and the properties they verify are elementary but nevertheless relevant. The user can choose which of the specifications to verify and the depth of the search space. After the symbolic execution engine terminates if a vulnerability is found the tool returns a counter-example, automatically forks the Ethereum network and executes a local transaction to confirm the existence of a bug without affecting the state of an existing contract in the Ethereum Blockchain.

Verx [34] is a symbolic execution engine that has a extremely well defined representation of the EVM using abstract interpretation, which allows the verifier to reduce a potential unbounded concrete state space into a finite one without losing precision, and a sound approximation of gas mechanics, which allows it to prune the search space for paths that are unreachable due to gas exhaustion. This engine also models contract calls, which enables the verification of groups of interacting smart contracts. User specifications are formalized in Past Linear Temporal Logic. The authors compare their tool with others present in the literature and reliably conclude that their tool is the state-of-the-art in symbolic execution methods.

Runtime Verification Runtime Verification is the real-time analysis of program execution, and it's based on observing executions of a system and using them to detect if system behaviors abide by its specifications. Analysis is performed by a monitor constructed by the system specifications and using system traces, which in the blockchain context are the blockchain actions. This type of verification can potentially be used to identify incorrect states that could not be reached by model checking or symbolic execution due to the limitations of state or path explosion. On smart contracts this approach is considered expensive due to additional gas consumption by the monitor and the increase in size of bytecode. Additionally there's further discussion on the feasibility of these approaches since some of them would require a hard-fork for an update of the EVM and other blockchains virtual machine.

Cap9 [23] is a pioneer when it comes to runtime verification, being the first and only paper in the literature to propose a smart contract that monitors other smart contracts the same way that a kernel manages the processes system calls on an operating system. When a smart contract needs to interact with critical resources, their "kernel" verifies on runtime and on-chain if the interaction is possible or not, reverting the transaction if an illegal operation occurs. This kernel is a smart contract which reads the smart contract to monitor using the EXTCODECOPY opcode and finds bytecode which needs to be verified on execution. The specifications for how a program should behave are user-defined and added to the kernel on-chain, these are conditions or restrictions for certain bytecode instruction in the EVM allowing for changes and modularity in the specifications. Cap9's approach to runtime verification allows for composability and upgrade-ability but exacerbates gas consumption.

Solythesis [21] is a source to source Solidity compiler that receives a smart contract and user specified invariant properties and creates an instrumented smart contract that reverts all transactions that violate the invariant. To minimize gas consumption and reduce performance bottlenecks it uses delta update and delta checks when accessing storage, reducing the amount of operations required for verification. Solythesis statically analysis every function of the smart contract and determines a set of state values that can be modified, and specifications that might transgress the specifications. The formalization of properties is accomplished with an domain specific invariant specification language that supports integers, variables, arithmetic expressions, conditional expressions, intermediate value declarations, and constraints. Such a language might present challenges for an every-day programmer looking to apply runtime verification to his contracts.

ContractLarva [30] leverages game theory to introduce a runtime verification technique to ensure properties are not violated. Before executing a transaction on the smart contract an actor stakes a certain amount of currency on the correct execution. Oppositely disagreeing actors are welcome to also stake currency if they believe the transaction will violate a specification. After the transaction is made other parties must reach a conclusion on whether there was a violation or not, using mechanisms similar to those used in decentralized governance. If there was a violation the contract is reverted and the disagreeing actors receive all the stake, if there wasn't, then the contract is left untouched and the executor receives all the stake. Specifications are characterised using automaton-based specifications in the form of dynamic event automata which are finite state automata with symbolic states. Upcoming work should elaborate on the contract revert mechanisms and how to deal with failure, along with more research on the decision mechanism.

1.2.1.3 Domain Specific Languages

In previous chapters we presented work aimed at formal verification and vulnerability detection. Another approach at the issue of smart contract correctness would be to use Domain Specific Languages that employ correct-by-construction syntax and semantics, ruling out many security risks by design. On this chapter we will present work on new smart contract oriented DSL's. It is important to notice that a lot of this DSL's are not only focused on verification but also on ease of implementation of financial contracts. *Composing Contracts: An Adventure in Financial Engineering* [35] introduces a specification for describing complex financial contracts using lazy evaluation and compositional denotational semantics in the form of a combinator library. This paper is referenced and used as base for plenty of the following DSL's.

Findel [1] is a declarative financial DSL that is considered to be a full implementation of Peyton et al [35] in the Ethereum blockchain. As mentioned previously this methodology requires a lazy evaluation of said contracts and so, the authors implement a smart contract that manages other Findel contracts, referred to as the marketplace, which keeps track of users' balances and permits the deployment and execution of Findel contracts. Although this implementation is broad enough to be used in other blockchains there's still further work to be done if one chooses to do so.

Marlowe [19] is a non-turing complete subset of Haskell [14] for the Cardano Blockchain, where the blockchain language is Plutus, also based on Haskell. Since Cardano uses a UTXO model and not an account based model as used in Ethereum, this DSL has several implementation details that make it harder for generalization and usage in other blockchains. This DSL is focused on composing financial contracts, as described in the paper by Peyton et. al [35] and the corresponding created smart contracts when used are evaluated by another smart contract on-chain that does the lazy interpretation of said contract, similar to Findel [1]. Further work shows efficient static analysis of Marlowe contracts [20] using SMT solvers which are able to check if properties of the contract are satisfied. Since Marlowe is not turing-complete problems such as state explosion do not exist, but as of now this tool can only check a small set of properties such as the contract paying every one it is supposed to. Further work will pursue extending static analysis to inspect other contract properties.

Formality [22] is a type system capable of proving theorems and focused on efficiency and simplicity, currently having only 700 LoC of core type-theory written in itself. It's a new project and still lacks any papers that formalize the work in literature, but developers are pushing forward development in their github repository [22], trying to build an improved version of alternative type systems such as Agda [28], Coq [42] and Idris [8]. Despite being a type system focused of formal proofs and verification, Formality aims to be market ready and add blockchain virtual machines such as EVM to its compilation targets, which will enable its usage as a formally verified DSL where formal proofs are automated and uncomplicated.

AxLang [6] is based on Scala [29] which is a statically-typed language that compiles to the Java Virtual Machine (JVM) and mixes object-oriented and functional programming paradigms. Developers can easily design imperative and object-oriented programming features for regular smart contracts and also employ formal verification, all within the same framework. AxLang is a subset of

Scala which compiles to the EVM, enabling the usage of Scala in blockchain applications. Unfortunately, more information on this project is unavailable seeing that the promise of open-sourcing the code by the team hasn't yet been fulfilled.

Michelson [18] and **Albert** [7] are two DSL's designed for the Tezos blockchain. Michelson is the native chain programming language and is Turing-complete, low-level and stack based interpreted. It is also statically typed to ensure the well-formedness of the program stack. Contracts specified in Michelson are pure functions that operate on stacks, they receive a stack as an input and return a stack as output, which removes side-effects, facilitating formal verification tools. **Mi-Cho-Coq** is an implementation of a Michelson interpreter in Coq for formal reasoning about contracts using weakest precondition calculus [12]. As Mi-Cho-Coq fully formalises Michelson's type system, syntax and semantics and provides Hoare style reasoning for properties it facilitates the programmer verification of their contracts. Although Michelson exhibits benefits over other DSL's, it is considered a difficult and abnormal target for compilers since it is a stack-based language. In order to mitigate such an issue Albert [7] was created as an intermediate language that keeps track of resources using names for the binding of variables and values allowing to abstract the order of values on Michelson's stack. Albert's compiler to Michelson is written in Coq which enables it's compilation to Mi-Cho-Coq's AST enabling its formal verification with the already existent tool. Even though the compiler is written in Coq it's correctness is yet to be proven and is left as upcoming work.

Scilla [37,38] is an explicitly-typed intermediate-level functional smart contract programming language. By intermediate the authors mean that Scilla is minimalist and implements precisely its formalisation, removing unnecessary syntactic sugar. Scilla's contract are designed as communicating automata where every computation is considered an atomic, pure and effectful transaction. Since Scilla uses high order functions and polymorphic lambda calculus it's quite simple to create on top of Scilla formal verification frameworks, which has been done by the authors for verification of custom domain-specific properties but, an embedding of Scilla into proof assistants such as Coq [42] or Isabelle/HOL [45] is still future work.

Vyper [43] is a contract-oriented, python like programming language that targets the EVM. It was created with the purpose of being maximally human-readable to prevent writing misleading code and easing audits. By default Vyper prevents over and under flows on array accesses and arithmetic. It is also strongly typed and decidable, deliberately limiting its expressiveness to make gas consumption predictable and upper bounded as well as more amenable to formal verification.

1.2.2 RQ2: How are specifications or program properties formalized?

Distinct formalization of systems and properties based on the level of abstraction fit into one of two categories [44]: contract-level formalization that verifies high-level behaviour of the contract, typically modeled using a variant of Temporal

Logic, and program-level formalization, which is more detailed and dependent on source-code and blockchain platform.

1.2.3 RQ3: What types of specifications do these formal methodologies aim to verify?

There is a clear separation between frameworks who allow the user to specify their own smart contract properties [2,5,17,23,25,26,40,46], and the ones who only verify pre-established properties [27]. These predefined properties commonly are if the contract is greedy, the amount of currency out is less than the amount in, reentrancy, integer under/overflows and contract suicide, if the contract can self-destruct.

1.2.4 RQ4: How well do they solve these issues?

The quality of the frameworks can be evaluated by their correctness, broadness and real world applicability. Correctness may be measured by the amount of false negatives or positives that a tool produces when verifying smart contracts. Obviously since theorem proving cannot produce neither, tools that use this method [2,4,5,46] are at an advantage. Some model checking and symbolic execution techniques [26] prune the search space using specific blockchain constraints, which allows them to validate properties with higher precision than others. The correctness of the tool is also limited by the quality of the property specification and so tools that use straightforward property formalization [2,17,21,23,26] have an edge. As mentioned in *RQ3* some frameworks are less broad than others on the formalization of properties, but some of these tools are blockchain agnostic [5,17,40] while others are blockchain specific [2,21,23,25–27,34,46] limiting their use. As far as real world applicability goes, it is fair to say that most of these tools have not seen adoption by the community of smart contract programmers since there still aren't plenty formally verified smart contracts with the exception of Ethereum's 2.0 Deposit Contract [31].

References

1.
2. Ahrendt, W., Bubel, R.: Functional verification of smart contracts via strong data integrity. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications. pp. 9–24. Springer International Publishing, Cham (2020)
3. Almakhour, M., Sliman, L., Samhat, A.E., Mellouk, A.: Verification of smart contracts: A survey. Pervasive and Mobile Computing **67**, 101227 (2020). https://doi.org/https://doi.org/10.1016/j.pmcj.2020.101227, `http://www.sciencedirect.com/science/article/pii/S1574119220300821`
4. Annenkov, D., Milo, M., Nielsen, J.B., Spitters, B.: Verifying, testing and running smart contracts in concert
5. Annenkov, D., Nielsen, J.B., Spitters, B.: Concert: a smart contract certification framework in coq. Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (Jan 2020). https://doi.org/10.1145/3372885.3373829, `http://dx.doi.org/10.1145/3372885.3373829`
6. Axioni: Axlang. https://medium.com/axoni/axlang-formally-verifiable-smart-contracts-for-the-ethereum-ecosystem-6201203be4e8 (2018)
7. Bernardo, B., Cauderlier, R., Pesin, B., Tesson, J.: Albert, an intermediate smart-contract language for the tezos blockchain (2020)
8. BRADY, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming **23**, 552–593 (9 2013). https://doi.org/10.1017/S095679681300018X, `https://journals.cambridge.org/article_S095679681300018X`
9. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: Nusmv: a new symbolic model checker. STTT **2**, 410–425 (03 2000). https://doi.org/10.1007/s100090050046
10. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) Security in Pervasive Computing. pp. 193–209. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
11. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. p. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
12. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer-Verlag, Berlin, Heidelberg (1990)
13. Holzmann, G.: Spin Model Checker, the: Primer and Reference Manual. Addison-Wesley Professional, first edn. (2003)
14. Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: A history of haskell: Being lazy with class. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. p. 12–1–12–55. HOPL III, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1238844.1238856, `https://doi.org/10.1145/1238844.1238856`
15. Inc, R.V.: Firefly. `https://fireflyblockchain.com/` (2019)
16. Jiao, J., Kan, S., Lin, S., Sanan, D., Liu, Y., Sun, J.: Semantic understanding of smart contracts: Executable operational semantics of solidity. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1695–1712. IEEE Computer Society, Los Alamitos, CA, USA (may 2020). https://doi.org/10.1109/SP40000.2020.00066, `https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00066`

17. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: Analyzing safety of smart contracts (01 2018). https://doi.org/10.14722/ndss.2018.23092
18. Labs, N.: Michelson. `https://tezos.gitlab.io/007/michelson.html` (2018)
19. Lamela Seijas, P., Nemish, A., Smith, D., Thompson, S.: Marlowe: Implementing and analysing financial contracts on blockchain. In: Bernhard, M., Bracciali, A., Camp, L.J., Matsuo, S., Maurushat, A., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security. pp. 496–511. Springer International Publishing, Cham (2020)
20. Lamela Seijas, P., Smith, D., Thompson, S.: Efficient Static Analysis of Marlowe Contracts, pp. 161–177 (10 2020). https://doi.org/10.1007/978-3-030-61467-$6_1$1
21. Li, A., Choi, J.A., Long, F.: Securing smart contract with runtime validation. Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (Jun 2020). https://doi.org/10.1145/3385412.3385982, `http://dx.doi.org/10.1145/3385412.3385982`
22. Maia, V.: Formality. `https://github.com/moonad/Formality` (2020)
23. Mandrykin, M., O'Shannessy, J., Payne, J., Shchepetkov, I.: Formal specification of a security framework for smart contracts (2020)
24. Miller, D., Nadathur, G.: Programming with Higher-Order Logic. Cambridge University Press, USA, 1st edn. (2012)
25. Nehaï, Z., Piriou, P., Daumas, F.: Model-checking of smart contracts. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). pp. 980–987 (2018). https://doi.org/10.1109/$Cybermatics_2$018.2018.00185
26. Nelaturu, K., Mavridoul, A., Veneris, A., Laszka, A.: Verified development and deployment of multiple interacting smart contracts with verisolid. In: 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). pp. 1–9 (2020). https://doi.org/10.1109/ICBC48266.2020.9169428
27. Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference. p. 653–663. ACSAC '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3274694.3274743, `https://doi.org/10.1145/3274694.3274743`
28. Norell, U.: Dependently typed programming in agda. In: Proceedings of the 6th International Conference on Advanced Functional Programming. p. 230–266. AFP'08, Springer-Verlag, Berlin, Heidelberg (2008)
29. Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: The scala language specification (2004)
30. Pace, G., Ellul, J., Azzopardi, S.: Monitoring smart contracts: Contractlarva and open challenges beyond (11 2018)
31. Park, D., Zhang, Y., Rosu, G.: End-to-end formal verification of ethereum 2.0 deposit smart contract. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 151–164. Springer International Publishing, Cham (2020)
32. Park, D., Zhang, Y., Saxena, M., Daian, P., Roşu, G.: A formal verification tool for ethereum vm bytecode. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 912–915. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3264591, `https://doi.org/10.1145/3236024.3264591`

33. Paulin-Mohring, C.: Introduction to the calculus of inductive constructions (11 2014)
34. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: Verx: Safety verification of smart contracts. pp. 1661–1677 (05 2020). https://doi.org/10.1109/SP40000.2020.00024
35. Peyton Jones, S., Eber, J.M., Seward, J.: Composing contracts: An adventure in financial engineering (functional pearl). SIGPLAN Not. **35**(9), 280–292 (Sep 2000). https://doi.org/10.1145/357766.351267, `https://doi.org/10.1145/357766.351267`
36. Rosu, G.: Matching logic - extended abstract. In: Fernandez, M. (ed.) 26th International Conference on Rewriting Techniques and Applications, RTA 2015. pp. 5–21. Leibniz International Proceedings in Informatics, LIPIcs, Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing (Jun 2015). https://doi.org/10.4230/LIPIcs.RTA.2015.5, 26th International Conference on Rewriting Techniques and Applications, RTA 2015 ; Conference date: 29-06-2015 Through 01-07-2015
37. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language. CoRR **abs/1801.00687** (2018), `http://arxiv.org/abs/1801.00687`
38. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with scilla. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360611, `https://doi.org/10.1145/3360611`
39. Singh, A., Parizi, R., Zhang, Q., Choo, K.K.R., Dehghantanha, A.: Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. Computers Security **88**, 101654 (10 2019). https://doi.org/10.1016/j.cose.2019.101654
40. Stefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G.: Semantics-based program verifiers for all languages. SIGPLAN Not. **51**(10), 74–91 (Oct 2016). https://doi.org/10.1145/3022671.2984027, `https://doi.org/10.1145/3022671.2984027`
41. Sánchez-Gómez, N., Torres-Valderrama, J., García-García, J.A., Gutiérrez, J.J., Escalona, M.J.: Model-based software design and testing in blockchain smart contracts: A systematic literature review. IEEE Access **8**, 164556–164569 (2020). https://doi.org/10.1109/ACCESS.2020.3021502
42. development team, T.C.: The Coq proof assistant reference manual. LogiCal Project (2004), `http://coq.inria.fr`, version 8.0
43. team, V.: Vyper. `https://vyper.readthedocs.io/en/stable/` (2020)
44. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification (2020)
45. Wenzel, M.: Isabelle/isar—a generic framework for human-readable proof documents. From Insight to Proof—Festschrift in Honour of Andrzej Trybulec **10**(23), 277–298 (2007)
46. Yang, Z., Lei, H., Qian, W.: A hybrid formal verification system in coq for ensuring the reliability and security of ethereum-based service smart contracts. IEEE Access **PP**, 1–1 (01 2020). https://doi.org/10.1109/ACCESS.2020.2969437