Revisiting Formal Verification in VeriSolid: An Analysis and Enhancements

Atefeh Zareh Chahoki[1], Marco Roveri[1], Daniel Amyot[2] and John Mylopoulos[2]

[1]*University of Trento, Italy*
[2]*University of Ottawa, Canada*

Abstract
While the immutability of smart contracts ensures unparalleled trust and transparency, it is also a double-edged sword, as any coding errors can lead to substantial financial losses and complex mitigation endeavours. VeriSolid is a security mitigation tool that offers a 'correct by design' platform, enabling developers to create and verify smart contracts using predefined property templates. After ensuring the correctness of the design, VeriSolid facilitates the automatic generation of equivalent Solidity code. Our motivational example is a prominent decentralized exchange (DEX), Uniswap, which is responsible for over 82% of DEX transactions. Our experiments uncover and resolve verification issues in two out of four VeriSolid templates. Additionally, we propose seven new templates, enhancing the VeriSolid platform through its open-source project. This research endeavor has yielded a significant enhancement in VeriSolid's coverage percentage, which represents the percentage of available software requirement specifications that can be formally represented and verified in VeriSolid. This study increased it from 45.6% to 90.5% based on a dataset consisting of 555 requirement specifications, thus enabling the verification of a wider range of property specifications.

Keywords
Automated Verification, Blockchain, Code Generation, Finite State Machines, Security, Smart Contracts, Solidity, Temporal Logic

1. Introduction and motivation

Smart contracts, evolving from cryptocurrencies pioneered by Bitcoin [1], offer decentralized, trustless execution of computable functions. They serve as a global computing platform, allowing parties to interact and transact without relying on a centralized authority or trusting each other implicitly. Ethereum [2, 3] introduced this transformative technology, thriving due to various blockchain imlementations, new languages, and business adoption [4]. However, blockchain's decentralization unveils vulnerabilities, exemplified by the DAO attack [5], underlining substantial financial risks due to the irreversibility inherent in smart contracts and emphasizing the urgent need for innovative security solutions for its specific vulnerabilities [6].

OVERLAY 2023: 5th Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis, November 7, 2023, Rome, Italy
atefeh.zareh@unitn.it (A. Zareh Chahoki); marco.roveri@unitn.it (M. Roveri); damyot@uottawa.ca (D. Amyot); jmylopou@uottawa.ca (J. Mylopoulos)
https://www.linkedin.com/in/atefeh-zareh/ (A. Zareh Chahoki); https://sites.google.com/view/marco-roveri/home (M. Roveri); https://www.site.uottawa.ca/~damyot/ (D. Amyot); https://www.cs.toronto.edu/~jm/ (J. Mylopoulos)
0009-0003-2004-7762 (A. Zareh Chahoki); 0000-0001-9483-3940 (M. Roveri); 0000-0003-2414-1791 (D. Amyot); 0000-0002-8698-3292 (J. Mylopoulos)

Solutions are varied [7] and include secure programming practices and developer patterns [8] to mitigate common vulnerabilities, automated vulnerability-detection tools [9, 10, 11] targeting specified properties, and formal verification approaches [12, 13], offering robust verification guarantees. The first category relies on programmer adherence, the second lacks contract-specific focus, and the third, although vital for ensuring correctness, poses automation challenges.

We have selected the Uniswap V2 Solidity code from the decentralized exchange (DEX) ecosystem as our real-world motivational example [14]. DEXs operate without intermediaries, facilitating peer-to-peer cryptocurrency transactions in a non-custodial manner. Our selection criteria included transaction count and percentage of total volume across all monitored DEXs on Etherscan.io. Uniswap V2 accounted for 82.51% of total transactions [15].

VeriSolid [16] employs a correct-by-design approach for smart contract development, emphasizing correctness during the design phase before code generation. However, our testing revealed verification errors and challenges in specifying desired properties, resulting in execution outcomes misaligned with defined properties. This paper addresses these issues, significantly improving VeriSolid's coverage from 45.6% to 90.5% across a dataset of 555 specifications, thereby enabling broader property verification.

We offer a concise overview of VeriSolid in Section 2, with an analysis of and improvements to VeriSolid's functionality in Section 3. Our contributions have been integrated into the VeriSolid codebase, now accessible to the community. Section 4 summarizes our findings and outlines potential future enhancements.

2. VeriSolid

VeriSolid is a correct-by-design tool that enables users to design contracts by modeling them as state machines. Using the Solidity language, users can specify details such as transition guards, statements, inputs, and contract definitions (e.g., internal variables). The user has access to predefined templates to specify desired properties on the contract. Once verification is triggered, VeriSolid translates the designed model and specified properties into BIP [17, 18] and then into CTL (Computational Tree Logic) [19] specifications in nuXmv [20]. The nuXmv tool subsequently verifies whether the CTL properties hold for the contract model. In the case of negative results, it provides counterexamples at the nuXmv level and then translates them back to the BIP level. In this paper, we assume the reader has basic knowledge of CTL [21].

3. VeriSolid Analysis and Extensions

While specifying our Uniswap model and verifying properties within the VeriSolid environment, a surprising incorrect verification verdict resulting from the second and third templates (Table 1) came to our attention. We explain this issue using a simpler model, i.e., one of the predefined smart contracts of the VeriSolid project called "*RockPaperScissors*" and depicted in Fig. 1.

The problematic scenario can be observed by providing "close#cancelPlay" as values to the second template definition (Table 1), which hence asserts that "the event (close) can happen only after the event (cancelPlay)". Surprisingly, the verification tool yields a result indicating that this property holds, despite our intuitive understanding suggesting otherwise. The same issue is also encountered with the third template, with parameters "close#cancelReveal#finish", which asserts that "if (close) happens, (cancelReveal) can happen only after (finish) happens". Once again, the verification reports that this property holds, despite the property trivially not holding.

Our investigation revealed that in the original VeriSolid tool, *weak until* ($A[xWy]$) properties were incorrectly encoded in nuXmv as *strong until* ($A[xUy]$) properties with an additional fairness constraint. Specifically, this encoding occurred for templates 2 and 3 (Table 1), which led to the generation of fairness constraints for the first (resp. third) parameter of the 2^{nd} (resp. 3^{rd}) template [22]. These wrong encodings are the root cause of the issues described previously.

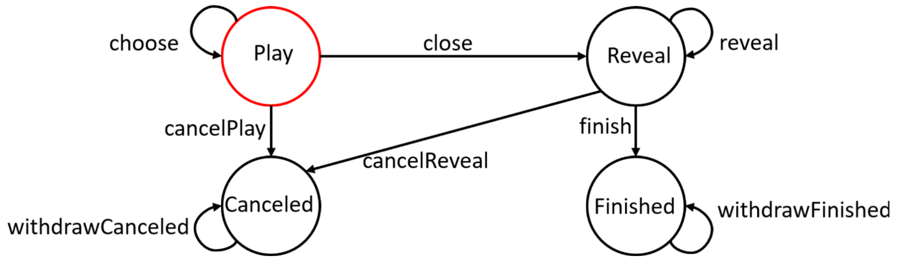

Figure 1: State machine of the *RockPaperScissors* VeriSolid predefined smart contract

Table 1
Safety and liveness property templates, including existing, fixed, implemented, and new ones

	No.	Property Template Description	CTL Formula	Sts.	%
Safety	1	**p** can never happen after **q** (Absence after)	$AG(q \rightarrow AG(\neg p))$	–	2.2
	2	**p** can happen only after **q** (Precedes globally)	$A[\neg pWq]$	fixed	4.5
	3	If **p** happens, **q** can happen only after **r** happens	$AG(p \rightarrow AXA[\neg qWr])$	fixed	–
	5	**p** can never happen (Absence globally)	$AG(\neg p)$	impl	7.4
	7	**p** can never happen before **q** (Absence before)	$A[(\neg p \vee AG(\neg q))Wq]$	impl	0.9
	8	**p** never happens between **q** and **r** (Absence between)	$AG(q \wedge \neg r \rightarrow A[(\neg p \vee AG(\neg r))Wr])$	new	3.2
	9	After **q** until **r**, **p** never happens (Absence Until)	$AG(q \wedge \neg r \rightarrow A[\neg pWr])$	new	1.6
	10	**p** always happens (Universality globally)	$AG(p)$	new	19.8
	11	After **q**, always the case that **p** happens (Universality After)	$AG(q \rightarrow AG(p))$	new	0.9
	12	Between **q** and **r**, always **p** happens (Universality between)	$AG(q \wedge \neg r \rightarrow A[(p \vee AG(\neg r))Wr])$	new	0.4
	13	**p** will eventually happen between **q** and **r** (Existence between)	$AG(q \wedge \neg r \rightarrow A[\neg rW(p \wedge \neg r)])$	new	1.4
Liveness	4	**p** will eventually happen after **q** (Response globally)	$AG(q \rightarrow AF(p))$	–	43.4
	6	**p** will eventually happen (Existence globally)	$AF(p)$	impl	2.2
	14	After **q**, **p** happens eventually (Existence after)	$A[\neg qW(q \wedge AF(p))]$	new	0.7
	15	After **q**, if **p** happens then in response **r** eventually happens (Response after)	$A[\neg qW(q \wedge AG(p \rightarrow AF(r)))]$	new	0.5
	16	If **p** happens then in response **q** eventually happens, followed by **r** eventually happens (Response chain globally)	$AG(p \rightarrow AF(q \wedge AX(AF(r))))$	new	1.4

Fairness constraints must be satisfied infinitely often within the system [23]. However, this encoding does not accurately capture the semantics of *weak until* properties. For instance, in the scenarios described earlier, it would require visiting infinitely often states such as "close" in the second template example and "finish" in the third template example, which is impossible due to the absence of a loopback mechanism. Since FAIRNESS(p) is used in nuXmv to restrict the verification to executions in which p holds infinitely often, this results in the fact that there are no paths satisfying this property; consequently any property becomes trivially true [21]. To address this issue, proper encoding of *weak until* using constructs supported by the verification engine is required. To this extent, we can leverage the equivalence $A[xWy] = \neg E[\neg yU(\neg x \wedge \neg y)]$ (see e.g., [24]), and rewrite each occurrence of *weak until* accordingly wherever needed.

We have also extended the supported verification templates of VeriSolid according to the most common CTL patterns [25]. Table 1 presents the VeriSolid property templates' logical descriptions and categorizations [25] between parentheses. The "CTL Formula" column represents the encoding in CTL. "Sts." corresponds to the status: "–" (unchanged), "fixed" (resolved in this paper), "impl" (introduced in a VeriSolid paper [16], implemented here), and "new" (introduced and implemented here). The last column reveals pattern coverage on a 555-example dataset [25]. The table encompasses all patterns with a score exceeding 1 in [25]. Our work significantly boosts VeriSolid's coverage, from 45.6% to 90.5%, enhancing its property verification capabilities. With these improvements, it is now possible to continue our analysis of the Uniswap smart contract.

4. Conclusion and Future Work

In this paper, we have improved the verification capabilities of VeriSolid, a correct-by-design development environment for smart contracts. Our investigations on this tool revealed incorrect encodings in nuXmv of two property templates. We corrected these issues and further contributed by implementing twelve new property templates. As a result, the verification templates now cover 90.5% of the 555 specification examples collected by Dwyer [25]. All these contributions are now available in the VeriSolid Git repository [26].

For future work, we are considering the following items. First, extend the current verification using advanced features in nuXmv by, e.g., encoding patterns in Linear Temporal Logic (LTL) [27] to then utilize advanced SAT-based model checking algorithms. Second, allow infinite state variables (reals, integers) beyond Boolean variables in guards and use SMT-based verification [28] available in nuXmv. Third, add checks to ensure the specified model is deadlock-free (each state has at least one future state). Fourth, bring back counterexamples in user level inspection. Finally, increase the granularity of the "lock" concept from FSolidM [29] (VeriSolid's predecessor) to eradicate reentrancy vulnerability in a "foolproof" manner, ensuring single transition execution at a time and preventing function nesting. Since this mechanism is overly restrictive, it is recommended to establish distinct locks for each method. Furthermore, we can offer users to define specific locks for individual transitions to ensure mutual exclusivity in concurrent transaction executions.

Acknowledgments

We thank Dr. A. Mavridou for her invaluable support throughout this research endeavor. We extend our appreciation to the anonymous reviewers whose insightful comments greatly contributed to enhancing the paper's quality. Prof. M. Roveri acknowledges partial funding from the CrossCon EU-funded project under Grant Agreement 101070537.

References

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, Decentralized business review (2008). URL: https://bitcoin.org/bitcoin.pdf.

[2] V. Buterin, A next-generation smart contract and decentralized application platform, 2014. URL: https://cobesto.com/preview-file/whitepaper-Ethereum.pdf, Ethereum white paper.

[3] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, 2014. URL: http://gavwood.com/paper.pdf, Ethereum project yellow paper.

[4] S. Dhaiouir, S. Assar, A systematic literature review of blockchain-enabled smart contracts: platforms, languages, consensus, applications and choice criteria, in: Research Challenges in Information Science: 14th International Conference, RCIS 2020, Springer, 2020, pp. 249–266. doi:10.1007/978-3-030-50316-1_15.

[5] Q. DuPont, Experiments in algorithmic governance: A history and ethnography of "the DAO," a failed decentralized autonomous organization, in: Bitcoin and beyond, Routledge, 2017, pp. 157–177. doi:10.4324/9781315211909-8.

[6] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on Ethereum smart contracts (SoK), in: Principles of Security and Trust, Springer, 2017, pp. 164–186. doi:10.1007/978-3-662-54455-6_8.

[7] H. Chen, M. Pendleton, L. Njilla, S. Xu, A survey on Ethereum systems security: Vulnerabilities, attacks, and defenses, ACM Computing Surveys 53 (2021-05-31) 1–43. doi:10.1145/3391195.

[8] The Solidity Authors, Security considerations – Solidity 0.8.14 documentation, 2022. URL: https://docs.soliditylang.org/en/develop/security-considerations.html.

[9] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, Formal verification of smart contracts: Short paper, in: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, 2016, pp. 91–96. doi:10.1145/2993600.2993611.

[10] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 2016, pp. 254–269. doi:10.1145/2976749.2978309.

[11] P. Tsankov, A. Dan, et al., Securify: Practical security analysis of smart contracts, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 67–82. doi:10.1145/3243734.3243780.

[12] Y. Hirai, Defining the ethereum virtual machine for interactive theorem provers, in: International Conference on Financial Cryptography and Data Security. FC 2017, Springer, 2017, pp. 520–535. doi:10.1007/978-3-319-70278-0_33.

[13] I. Grishchenko, M. Maffei, C. Schneidewind, A semantic framework for the security analysis of Ethereum smart contracts, in: Principles of Security and Trust: 7th International Conference, POST 2018, Springer, 2018, pp. 243–269. doi:10.1007/978-3-319-89722-6_10.

[14] etherscan.io, Uniswap (UNI) token tracker | Etherscan, 2023. URL: https://etherscan.io/token/0x1f9840a85d5af5bf1d1762f925bdaddc4201f984.

[15] etherscan.io, DEX activity | Etherscan, 2023. URL: http://etherscan.io/stat/dextracker.

[16] A. Mavridou, A. Laszka, E. Stachtiari, A. Dubey, VeriSolid: Correct-by-design smart contracts for Ethereum, in: International Conference on Financial Cryptography and Data Security. FC 2019, Springer, 2019, pp. 446–465. doi:978-3-030-32101-7_27.

[17] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, J. Sifakis, Rigorous component-based system design using the BIP framework, IEEE Softw. 28 (2011) 41–48. URL: https://doi.org/10.1109/MS.2011.27. doi:10.1109/MS.2011.27.

[18] S. Bliudze, A. Cimatti, M. Jaber, S. Mover, M. Roveri, W. Saab, Q. Wang, Formal verification of infinite-state BIP models, in: International Symposium on Automated Technology for Verification and Analysis, Springer, 2015, pp. 326–343. doi:10.1007/978-3-319-24953-7_25.

[19] E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in: Workshop on logic of programs, Springer, 1981, pp. 52–71. doi:10.1007/BFb0025774.

[20] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, The nuXmv symbolic model checker, in: Computer Aided Verification: 26th International Conference, CAV 2014, Springer, 2014, pp. 334–342. doi:10.1007/978-3-319-08867-9_22.

[21] C. Baier, J.-P. Katoen, Principles of model checking, MIT Press, 2008.

[22] A. Mavridou, VerifyContract.js, 2023. URL: https://github.com/anmavrid/smart-contracts/blob/master/src/plugins/VerifyContract/VerifyContract.js.

[23] R. Cavada, A. Cimatti, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, NuSMV 2.6 tutorial, 2015. URL: https://nusmv.fbk.eu/NuSMV/tutorial/v26/tutorial.pdf.

[24] M. Dwyer, Property pattern mappings for CTL, 2023. URL: https://matthewbdwyer.github.io/psp/patterns/ctl.html.

[25] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Patterns in property specifications for finite-state verification, in: Proceedings of the 21st International Conference on Software Engineering, 1999, pp. 411–420. doi:10.1145/302405.302672.

[26] A. Mavridou, VerifyContract.js, 2023. URL: https://github.com/anmavrid/smart-contracts.

[27] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, IEEE Computer Society, 1977, pp. 46–57. URL: https://doi.org/10.1109/SFCS.1977.32. doi:10.1109/SFCS.1977.32.

[28] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, Satisfiability Modulo Theories, volume 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2009. doi:10.3233/978-1-58603-929-5-825.

[29] A. Mavridou, A. Laszka, Designing secure Ethereum smart contracts: A finite state machine based approach, in: Financial Cryptography and Data Security. FC 2018, Springer, 2018, pp. 523–540. doi:978-3-662-58387-6_28.