Revisiting Formal Verification in VeriSolid:
An Analysis and Enhancements

Atefeh Zareh Chahoki\textsuperscript{1}, Marco Roveri\textsuperscript{1}, Daniel Amyot\textsuperscript{2} and John Mylopoulos\textsuperscript{2}

\textsuperscript{1}University of Trento, Italy
\textsuperscript{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 implementations, 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].
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 \( (A [x W y]) \) properties were incorrectly encoded in nuXmv as strong until \( (A [x U y]) \) 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\textsuperscript{nd} (resp. 3\textsuperscript{rd}) template [22]. These wrong encodings are the root cause of the issues described previously.

![State machine of the RockPaperScissors VeriSolid predefined smart contract](image)

**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

<table>
<thead>
<tr>
<th>No.</th>
<th>Property Template Description</th>
<th>CTL Formula</th>
<th>Sts.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p can never happen after q (Absence after)</td>
<td>( AG(q \rightarrow AG(\neg p)) )</td>
<td>–</td>
<td>2.2</td>
</tr>
<tr>
<td>2</td>
<td>p can happen only after q (Precedes globally)</td>
<td>( A[\neg p W q] )</td>
<td>fixed</td>
<td>4.5</td>
</tr>
<tr>
<td>3</td>
<td>If p happens, q can happen only after r happens</td>
<td>( AG(p \rightarrow AX A[\neg q W r]) )</td>
<td>fixed</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>p will eventually happen after q (Response globally)</td>
<td>( AG(q \rightarrow AF(p)) )</td>
<td>–</td>
<td>43.4</td>
</tr>
<tr>
<td>5</td>
<td>p will eventually happen (Existence globally)</td>
<td>( AF(p) )</td>
<td>impl</td>
<td>2.2</td>
</tr>
<tr>
<td>6</td>
<td>After q, p happens eventually (Existence after)</td>
<td>( A[\neg q W (q \land AF(p))] )</td>
<td>new</td>
<td>0.7</td>
</tr>
<tr>
<td>7</td>
<td>After q, if p happens then in response r eventually happens (Response after)</td>
<td>( A[\neg q W (q \land AG(p \rightarrow AF(r)))] )</td>
<td>new</td>
<td>0.5</td>
</tr>
<tr>
<td>8</td>
<td>p will eventually happen between q and r (Existence between)</td>
<td>( AG(q \land \neg r \rightarrow A[\neg r W (p \land \neg r)]) )</td>
<td>new</td>
<td>1.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No.</th>
<th>Property Template Description</th>
<th>CTL Formula</th>
<th>Sts.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>After q until r, p never happens (Absence Until)</td>
<td>( AG(q \land \neg r \rightarrow A[\neg p W r]) )</td>
<td>new</td>
<td>1.6</td>
</tr>
<tr>
<td>10</td>
<td>p always happens (Universality globally)</td>
<td>( AG(p) )</td>
<td>new</td>
<td>19.8</td>
</tr>
<tr>
<td>11</td>
<td>After q, always the case that p happens (Universality After)</td>
<td>( AG(q \rightarrow AG(p)) )</td>
<td>new</td>
<td>0.9</td>
</tr>
<tr>
<td>12</td>
<td>Between q and r, always p happens (Universality between)</td>
<td>( AG(q \land \neg r \rightarrow A[(p \lor AG(\neg r)) W r]) )</td>
<td>new</td>
<td>0.4</td>
</tr>
<tr>
<td>13</td>
<td>p will eventually happen between q and r (Existence between)</td>
<td>( AG(q \land \neg r \rightarrow A[\neg r W (p \land \neg r)]) )</td>
<td>new</td>
<td>1.4</td>
</tr>
<tr>
<td>14</td>
<td>After q, p happens eventually (Existence after)</td>
<td>( A[\neg q W (q \land AF(p))] )</td>
<td>new</td>
<td>0.7</td>
</tr>
<tr>
<td>15</td>
<td>After q, if p happens then in response r eventually happens (Response after)</td>
<td>( A[\neg q W (q \land AG(p \rightarrow AF(r)))] )</td>
<td>new</td>
<td>0.5</td>
</tr>
<tr>
<td>16</td>
<td>If p happens then in response q eventually happens, followed by r eventually happens (Response chain globally)</td>
<td>( AG(p \rightarrow AF(q \land AX (AF(r)))) )</td>
<td>new</td>
<td>1.4</td>
</tr>
</tbody>
</table>
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 \( A[x W y] = \neg E[\neg y U (x \land \neg y)] \) (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


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


