VESC: Towards Temporal Verification of Smart Contracts

Kevin Johanson, Sam Larsen, Yuandong Cyrus Liu

Grinnell College

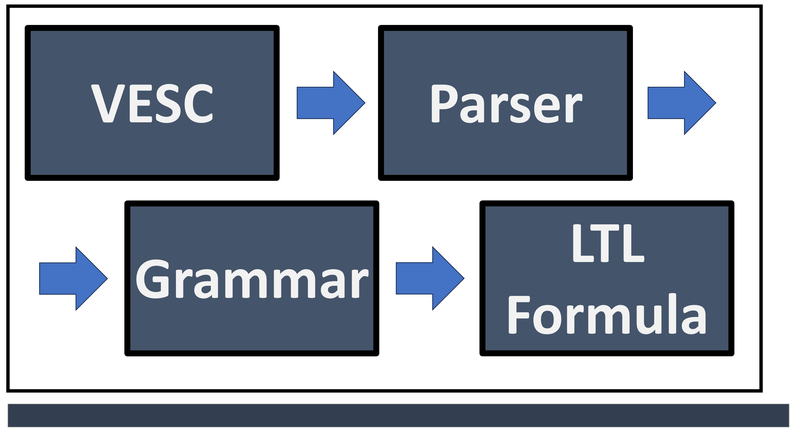
CSC 499
Yuandong Cyrus Liu
Summer 2024

Abstract

Blockchain technologies are applied in diverse domains such as financial systems, supply chains, and identity management, leading to the emergence of various smart contract languages design. These contracts often involve time dependent transactions recorded immutably on the blockchain, making their correctness crucial. This paper addresses the formal verification of temporal behaviors in smart contracts without human interaction. We study 9 recent smart contract languages used in 7 leading blockchains and model 27 common temporal patterns from 2102 benchmarks across 9 domain-specific application categories. We introduce VESC, a temporal specification language that allows developers to specify temporal properties in structured natural language, which VESC compiles into formal linear temporal logic. Our experiments demonstrate that VESC effectively specifies common temporal behaviors, paving the way for automated temporal verification of smart contracts.

Smart Contract Patterns

Smart contract patterns are algorithms and solutions that appear frequently in blockchain code spaces. These patterns help developers by promoting the efficient reuse of code and design, as well as setting standards for security. Smart contract patterns can often be used across any blockchain and can be implemented in any language. We categorize these patterns into four major fields, security, efficiency, access control, and contract management.



Overview

Temporal behaviors are generally easy to describe in natural language, however it's nontrivial to specify them precisely in formal logic which can be interpreted by the machine. Figure 1 shows the Speed Bump pattern from a smart contract. In the code snippet, line 7 `WAIT_PERIOD` is set to 7 days, at line 10, the `withdraw` function requires the current time to be greater than the time of the initial withdrawal request plus the wait period, in this case, seven days later.

```
1   struct Withdrawal {
2       uint amount;
3       uint requestedAt;
4   }
5   mapping(address => uint) private balances;
6   mapping(address => Withdrawal) private withdrawals;
7   uint constant WAIT_PERIOD = 7 days;
8
9   function withdraw() public {
10      if (withdrawals[msg.sender].amount > 0 && now >
    withdrawals[msg.sender].requestedAt + WAIT_PERIOD) {
11          uint amount = withdrawals[msg.sender].amount;
12          withdrawals[msg.sender].amount = 0;
13          msg.sender.transfer(amount);
14      }
15  }
```

Figure 1: *Example of a Speed Bump Security Pattern*

VESC encodes the temporal properties of the code snippet above as simply `Bump(time t)`. For this example, when we parse `Bump(7 days)`, VESC compiles it into an LTL formula:

$$\mathcal{F}(call \implies stall\ \mathcal{U}\ currentTime \equiv callTime + 7\ days)$$

This formula states that when the function (`withdraw`) is called, the transaction is stalled until the current time is seven days after the time of the initial call.

$$\begin{aligned}
expr &::= Security \mid Efficiency \mid AccessControl \mid ContractManagement \\
Security &::= BalanceLimit(n) \mid MathPattern \mid MutexPattern \mid \\
&\quad Timing \mid Move \mid PullOverPush \\
Efficiency &::= IncentiveExecution \mid TightVariable \mid Library \mid StringLimit \\
AccessControl &::= Restriction(n) \mid Challenge(addr, prompt) \mid Whitelist(addr) \mid \\
&\quad Owner(addr) \mid RBAC(addr, role) \mid Multi([addr, addr, ...]) \\
ContractManagement &::= Agree([addr, addr, ...]) \mid Distinct(addr) \mid Proxy([addr, addr, ...]) \\
MPattern &::= Import(Safemath) \mid Redefine(opr) \\
Timing &::= EStop \mid Constraint(t) \mid Rate(n, t) \mid Bump(t) \\
Move &::= Transfer(n, addr) \mid Send(n, addr) \mid Call.value(n, addr, gas) \\
number &::= int \mid float \mid fixed(n) \\
addr &::= 0x(0-9|A-F)\{40\} \\
time &::= int\ sec \mid int\ min \mid int\ hr \mid int\ days
\end{aligned}$$

Figure 2: *VESC Specification Language Grammar*

VESC Specification Language and Implementation

VESC compiles front-end specification language to a linear temporal logic formula.

$\varphi ::= p \in \mathcal{AP} \mid \neg\varphi \mid \mathcal{G}\varphi \mid \mathcal{F}\varphi \mid \varphi \rightarrow \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathcal{X}\varphi \mid \varphi\mathcal{U}\varphi \mid \varphi\mathcal{R}\varphi$

VESC's grammar is defined as a list of expressions (`expr`). These expressions model the four key categories of patterns: `Security`, `Efficiency`, `Access Control`, and `Contract Management`. Each type of expression is broken down further into pattern constructs. Each pattern may take arguments in the form of numbers, time units, or addresses. A full description of VESC's grammar is described in Figure 2. Figure 3 shows VESC output results for each benchmark: the first column presents our VESC expressions, and the second column contains the LTL formulae that VESC compiles to.

VESC Specification	VESC Output LTL Formula
`BalanceLimit (20000)`	`G((Balance <= 20000))`
`Bump(5)`	`F(((call -> stall) U (currentTime == (callTime + 5))))`
`Constraint(1)`	`F(((accessable == True) U (CurrentTime == (StartTime + 1))))`
`Rate(1, 1)`	`F((call -> ((callable == True) -> ((remainingCalls-1) -> X(((remainingCalls == 0) -> ((callable == False) U (currentTime == (initialTime + 1)))))))))`
`EmergencyStop`	`F((EmergencyStop == True))`
`IncentiveExecution(cleanUp, 20)`	`F((cleanUp -> ((callerAddress + 20) /\ (OwnerAddress - 20))))`
`Library(OpenZepplin, std)`	`G(OpenZepplin, SolidityStandardLibrary)`
`StringLimit(32)`	`G((StringLimit <= 32))`
`Owner(addr1)`	`F(contract.owner == FALSE -> contract.setOwner = addr1) ||`

Figure 3: *VESC Sample Input/Output Results*

Acknowledgements

We would like to thank our MAP advisor, Cyrus Liu for his continuing aid in the development of this project.