Formalisms Report
Multiple Interacting Smart Contracts with VeriSolid

Keerthi Nelaturu*, Anastasia Mavridou†, Andreas Veneris*, Aron Laszka‡
* University of Toronto † SGT Inc. / NASA Ames Research Center ‡
University of Houston

CONTENTS

Formalisms Report
Multiple Interacting Smart Contracts with VeriSolid

I. Supported Solidity Subset

Here, we include formal definitions for the subset of Solidity statements that VeriSolid supports [1]. Our work builds on and extends this set (see Section **??**). VeriSolid uses the following notation:

- $\mathbb{T}$ is the set of Solidity types;
- $\mathbb{I}$ is the set of valid Solidity identifiers;
- $\mathbb{D}$ is the set of Solidity event and custom type definitions;
- $\mathbb{E}$ is the set of Solidity expressions;
- $\mathbb{C}$ is the set of Solidity expressions without side effects (i.e., expression whose evaluation does not change storage, memory, balances, etc.);
- $\mathbb{S}$ is the set of supported Solidity statements.

Before we get into the details of the formalisms, we brief on some breaking changes in Solidity v0.5.0 [2] since the VeriSolid framework was developed in accordance with Solidity v0.4.0 [1]. In this paper, we describe updated grammar and operational semantics that are compatible with the changes.

First, the new version of Solidity does not support arithmetic operations on binary expressions anymore. Second, the unary (+) operator is no longer supported. Both of these changes are enforced in the definitions of Solidity expressions.

VeriSolid defines the set of supported event ($\langle event \rangle$), custom type ($\langle struct \rangle$), and modifier ($\langle modifier \rangle$) definitions $\mathbb{D}$ as follows:

$$\langle event \rangle ::= \texttt{event}\ @identifier\ \texttt{(}\ \big(@type\ @identifier\ (\texttt{,}\ @type\ @identifier)*\big)?\ \texttt{);}$$

$$\langle struct \rangle ::= \texttt{struct}\ @identifier\ \texttt{\{}\ (@type\ @identifier\ \texttt{;})+\ \texttt{\}}$$

VeriSolid uses $\mathbb{E}$ to denote the set of Solidity expressions. Further, it uses $\mathbb{C}$ to denote the following subset of Solidity expressions, which do not have any side effects:

$$
\begin{aligned}
\langle pure \rangle ::= &| \ \langle variable \rangle \\
&| \ @constant \\
&| \ \texttt{(} \ \langle pure \rangle \ \texttt{)} \\
&| \ \langle unary \rangle \ \langle pure \rangle \\
&| \ \langle pure \rangle \ \langle operator \rangle \ \langle pure \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle boolean \rangle ::= &| \ \langle variable \rangle \\
&| \ @constant \\
&| \ \texttt{(} \ \langle boolean \rangle \ \texttt{)} \\
&| \ \langle unary\_logical \rangle \ \langle boolean \rangle \\
&| \ \langle boolean \rangle \ \langle logical\_operator \rangle \ \langle boolean \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle variable \rangle ::= &| \ @identifier \\
&| \ \langle variable \rangle \ \texttt{.} \ @identifier \\
&| \ \langle variable \rangle \ \texttt{[} \ \langle pure \rangle \ \texttt{]} \ \texttt{[} \ \langle boolean \rangle \ \texttt{]}
\end{aligned}
$$

$$\langle arithmetic\_operator \rangle ::= \ \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{/} \mid \texttt{\%}$$

$$\langle logical\_operator \rangle ::= \ \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{<=} \mid \texttt{\&\&} \mid \texttt{||}$$

$$\langle operator \rangle ::= \ \langle arithmetic\_operator \rangle \mid \langle logical\_operator \rangle$$

$$\langle unary\_arithmetic \rangle ::= \ \texttt{-}$$

$$\langle unary\_logical \rangle ::= \ \texttt{!}$$

$$\langle unary \rangle ::= \ \langle unary\_arithmetic \rangle \mid \langle unary\_logical \rangle$$

VeriSolid supports the following types of statements:

- variable declarations (e.g., `int32 value = 0;` and `address from = msg.sender;`),
- expressions (e.g., `amount = balance[msg.sender];` or `msg.sender.transfer(amount);`),
- event statements (e.g., `emit Deposit(amount, msg.sender);`),
- `return` statements (e.g., `return;` and `return amount;`),
- `if`, `if ... else` and `switch` selection statements (including `if ... else if ...` and so on),
- `for`, `while`, and `do ... while` loop statements,
- compound statements (i.e., { *statement1 statement2* ... }).

The formal grammar of the subset of supported Solidity statements $\mathbb{S}$ is as follows:

$$
\begin{aligned}
\langle statement \rangle ::= \\
&\mid \langle declaration \rangle \texttt{ ;} \\
&\mid @expression \texttt{ ;} \\
&\mid \texttt{emit } @identifier \texttt{(} \left(@expression \right. \\
&\quad \left. \texttt{, } @expression \right) * \left. \right) ? \texttt{);} \\
&\mid \texttt{return } (@pure) ? \texttt{ ;} \\
&\mid \texttt{if (} @expression \texttt{)} \langle statement \rangle \\
&\quad (\texttt{else } \langle statement \rangle) ? \\
&\mid \texttt{for (} \langle declaration \rangle \texttt{ ;} @expression \texttt{ ;} \\
&\quad @expression \texttt{)} \langle statement \rangle \\
&\mid \texttt{while (} @expression \texttt{)} \langle statement \rangle \\
&\mid \texttt{do } \langle statement \rangle \texttt{ while (} @expression \texttt{);} \\
&\mid \texttt{\{} (\langle statement \rangle) * \texttt{\}}
\end{aligned}
$$

$$
\langle declaration \rangle ::= @type\ @identifier\ (\texttt{=} @expression) ?
$$

where $@expression \in \mathbb{E}$ is a primary Solidity expression, which may include function calls, transfers, etc., while $@pure \in \mathbb{C}$ is a Solidity expression without side effects.

A transition-system *smart contract* is a tuple $(D, S,\ S_F,\ s_0,\ a_0, a_F, V, T)$, where

- $D \subset \mathbb{D}$ is a set of custom event and type definitions;
- $S \subset \mathbb{I}$ is a finite set of states;
- $S_F \subset S$ is a set of final states;
- $s_0 \in S$, $a_0 \in \mathbb{S}$ are the initial state and action;
- $a_F \in \mathbb{S}$ is the fallback action;
- $V \subset \mathbb{I} \times \mathbb{T}$ contract variables (i.e., variable names and types);
- $T \subset \mathbb{I} \times S \times 2^{\mathbb{I} \times \mathbb{T}} \times \mathbb{C} \times (\mathbb{T} \cup \emptyset) \times \mathbb{S} \times S$ is a transition relation (i.e., transition name, source state, parameter variables, guard, return type, action, and destination state).

II. Operational Semantics of Systems of Transitions Systems

We let Ψ denote the state of the ledger, which includes account balances, values of state variables in all contracts, number and timestamp of the last block, etc. We let $\mathbf{s}$ the current states of contracts of the system. During the execution of a function, the execution state $\sigma = (\Psi, M, \mathbf{s}, \kappa)$ also includes the memory and stack state M, and the set of destroyed contracts κ. To handle return statements and exceptions, we also introduce

an execution status, which is E when an exception has been raised, $R[v]$ when a return statement has been executed with value v (i.e., `return` v), and N otherwise. Finally, we let $\text{Eval}(\sigma, \text{Exp}) \to \langle \hat{\sigma}, R[v] \rangle$ signify that the evaluation of a Solidity expression Exp in execution state σ yields value v.[1] On the other hand, $\text{Eval}(\sigma, \text{Exp}) \to \langle \hat{\sigma}, E \rangle$ signifies that the evaluation has resulted in an exception.

In our model, a transaction is triggered by providing a contract instance $i \in \mathcal{C}$ with a function name *name* $\in \mathbb{I}$ and a list of parameter values $v_1, v_2, \ldots$. The transaction invokes the function in the current ledger and contract states Ψ and $\mathbf{s}$, which results in changed ledger and contract states Ψ' and $\mathbf{s}'$ as well as a set of contracts κ that have selfdestructed during execution (see rule FUNC). The transaction changes the states of these contracts $j \in \kappa$ to a special state $s''_j =$ *destroyed*, and then makes the new ledger and contract states Ψ' and $\mathbf{s}''$ permanent. This normal execution is captured by the TRANS rule:

$$\text{TRANS} \quad \frac{\begin{array}{c}\langle (\Psi, \mathbf{s}, \emptyset), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \to \langle (\Psi', \mathbf{s}', \kappa), x \rangle, \quad x \in \{N, R[v]\} \\ \forall j \in \kappa : s''_j = \textit{destroyed}, \quad \forall j \notin \kappa : s''_j = s'_j \\ \mathbf{s}'' = s''_1, \ldots, s''_{|\mathcal{C}|}\end{array}}{\langle (\Psi, \mathbf{s}), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \to \langle (\Psi', \mathbf{s}''), N \rangle}$$

The execution of a transaction may fail due to the invoked function encountering an exception, which is captured by the TRANS-EXC rule:

$$\text{TRANS-EXC} \quad \frac{\langle (\Psi, \mathbf{s}, \emptyset), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \to \langle (\Psi', \mathbf{s}', \kappa), E \rangle}{\langle (\Psi, \mathbf{s}), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \to \langle (\Psi, \mathbf{s}), E \rangle}$$

Similar to a transaction, a function call is also triggered by providing a contract instance $i \in \mathcal{C}$ with a function name *name* $\in \mathbb{I}$ and a list of parameter values $v_1, v_2, \ldots$. The execution first checks if there exists a transition (i.e., function) t with name $t^{name} = name$, if the origin state t^{from} of this transition t is the current state s_i, and if the guard condition g_t evaluates to `true` with the execution state σ initialized using the parameter values $v_1, v_2, \ldots$. A normal execution passes the above tests and executes the action a_t of the transition, resulting in a new execution state σ' and keeping the execution status normal N. Finally, it sets the current state of the contract s''_i to the destination state t^{to} of the transition, and yields the new ledger, contract, and destroyed states Ψ', $\mathbf{s}''$, and κ' as well as normal execution status N. This normal execution is captured by the FUNC rule:

[1] Note that the correctness of our transformations does not depend on the exact semantics of Eval for expressions that do not include interactions with other contracts.

$$\text{FUNC} \quad \frac{\begin{array}{c} s_1, \ldots, s_{|\mathcal{C}|} = \mathbf{s}, \quad t \in T_i, \quad \textit{name} = t^{name}, \quad s_i = t^{\textit{from}} \\ M = \textit{Params}(t, v_1, v_2, \ldots), \quad \sigma = (\Psi, M, \mathbf{s}, \kappa) \\ \textit{Eval}(\sigma, g_t) \rightarrow \langle \sigma, R[\texttt{true}] \rangle \\ \langle (\sigma, N), a_t \rangle \rightarrow \langle (\sigma', N), \cdot \rangle, \quad \sigma' = (\Psi', M', \mathbf{s}', \kappa'), \\ s'_1, \ldots, s'_{|\mathcal{C}|} = \mathbf{s}', \quad s''_i = t^{to}, \quad s'_1, \ldots, s''_i, \ldots, s'_{|\mathcal{C}|} = \mathbf{s}'' \end{array}}{\langle (\Psi, \mathbf{s}, \kappa), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \rightarrow \langle (\Psi', \mathbf{s}'', \kappa'), N \rangle}$$

A function may also return a value, which is captured by the FUNC-RET rule:

$$\text{FUNC-RET} \quad \frac{\begin{array}{c} s_1, \ldots, s_{|\mathcal{C}|} = \mathbf{s}, \quad t \in T_i, \quad \textit{name} = t^{name}, \quad s_i = t^{\textit{from}} \\ M = \textit{Params}(t, v_1, v_2, \ldots), \quad \sigma = (\Psi, M, \mathbf{s}, \kappa) \\ \textit{Eval}(\sigma, g_t) \rightarrow \langle \sigma, R[\texttt{true}] \rangle \\ \langle (\sigma, N), a_t \rangle \rightarrow \langle (\sigma', R[v]), \cdot \rangle, \quad \sigma' = (\Psi', M', \mathbf{s}', \kappa'), \\ s'_1, \ldots, s'_{|\mathcal{C}|} = \mathbf{s}', \quad s''_i = t^{to}, \quad s'_1, \ldots, s''_i, \ldots, s'_{|\mathcal{C}|} = \mathbf{s}'' \end{array}}{\langle (\Psi, \mathbf{s}, \kappa), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \rightarrow \langle (\Psi', \mathbf{s}'', \kappa'), R[v] \rangle}$$

The execution of a function might encounter an exception for a number of reasons. First, the contract might be in the wrong state, which is captured by the FUNC-WRO rule:

$$\text{FUNC-WRO} \quad \frac{s_1, \ldots, s_{|\mathcal{C}|} = \mathbf{s}, \quad t \in T_i, \quad \textit{name} = t^{name}, \quad s_i \neq t^{\textit{from}}}{\langle (\Psi, \mathbf{s}, \kappa), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \rightarrow \langle (\Psi, \mathbf{s}, \kappa), E \rangle}$$

A function may also fail is the guard condition is not satisfied, which is captured by the FUNC-GRD rule:

$$\text{FUNC-GRD} \quad \frac{\begin{array}{c} s_1, \ldots, s_{|\mathcal{C}|} = \mathbf{s}, \quad t \in T_i, \quad \textit{name} = t^{name}, \quad s_i = t^{\textit{from}} \\ M = \textit{Params}(t, v_1, v_2, \ldots), \quad \sigma = (\Psi, M, \mathbf{s}, \kappa) \\ \textit{Eval}(\sigma, g_t) \rightarrow \langle \sigma, R[\texttt{false}] \rangle \end{array}}{\langle (\Psi, \mathbf{s}, \kappa), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \rightarrow \langle (\Psi, \mathbf{s}, \kappa), E \rangle}$$

Alternatively, the evaluation of the guard condition may result in an exception, which is captured by the FUNC-EXC1 rule:

$$\text{FUNC-EXC1} \quad \frac{\begin{array}{c} s_1, \ldots, s_{|\mathcal{C}|} = \mathbf{s}, \quad t \in T_i, \quad \textit{name} = t^{name}, \quad s_i = t^{\textit{from}} \\ M = \textit{Params}(t, v_1, v_2, \ldots), \quad \sigma = (\Psi, M, \mathbf{s}, \kappa) \\ \textit{Eval}(\sigma, g_t) \rightarrow \langle \sigma, E \rangle \end{array}}{\langle (\Psi, \mathbf{s}, \kappa), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \rightarrow \langle (\Psi, \mathbf{s}, \kappa), E \rangle}$$

Finally, if the execution passes the guard condition, it may still encounter an exception in the execution of the action, which is captured by the FUNC-EXC2 rule:

$$\text{FUNC-EXC2} \quad \frac{\begin{array}{c} s_1, \ldots, s_{|\mathcal{C}|} = \mathbf{s}, \quad t \in T_i, \quad \textit{name} = t^{name}, \quad s_i = t^{\textit{from}} \\ M = \textit{Params}(t, v_1, v_2, \ldots), \quad \sigma = (\Psi, M, \mathbf{s}, \kappa) \\ \textit{Eval}(\sigma, g_t) \to \langle \sigma, R[\texttt{true}] \rangle \\ \langle (\sigma, N), a_t \rangle \to \langle (\sigma', E), \cdot \rangle \end{array}}{\langle (\Psi, \mathbf{s}, \kappa), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \to \langle (\Psi, \mathbf{s}, \kappa), E \rangle}$$

If there is no function corresponding to the provided name, then the fallback action is performed, which is captured by the FUNC-FALL rule:

$$\text{FUNC-FALL} \quad \frac{\begin{array}{c} \forall t \in T_i : \textit{name} \neq t^{name} \\ \sigma = (\Psi, \emptyset, \mathbf{s}, \kappa) \\ \langle (\sigma, N), a_F \rangle \to \langle (\sigma', x), \cdot \rangle, \quad x \neq E, \quad \sigma' = (\Psi', M', \mathbf{s}', \kappa'), \end{array}}{\langle (\Psi, \mathbf{s}, \kappa), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \to \langle (\Psi', \mathbf{s}', \kappa'), N \rangle}$$

The fallback action may also encounter an exception, which is captured by the FUNC-EXC3 rule:

$$\text{FUNC-EXC3} \quad \frac{\begin{array}{c} \forall t \in T_i : \textit{name} \neq t^{name} \\ \sigma = (\Psi, \emptyset, \mathbf{s}, \kappa) \\ \langle (\sigma, N), a_F \rangle \to \langle (\sigma', E), \cdot \rangle \end{array}}{\langle (\Psi, \mathbf{s}, \kappa), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \to \langle (\Psi, \mathbf{s}, \kappa), E \rangle}$$

Functions is invoked either by an external actor through a transaction, or by another function when evaluating an expression that contains a function call, which is captured by the EVAL-CALL rule:

$$\text{EVAL-CALL} \quad \frac{\begin{array}{c} \sigma = (\Psi, M, \mathbf{s}, \kappa) \\ \langle (\Psi, \mathbf{s}, \kappa), i.\textit{name}\,(v_1, v_2, \ldots) \rangle \to \langle (\Psi', \mathbf{s}', \kappa'), x \rangle \\ \sigma' = (\Psi', M, \mathbf{s}', \kappa') \end{array}}{\textit{Eval}\,(\sigma, i.\textit{name}(v_1, v_2, \ldots)) \to \langle \sigma', x \rangle}$$

Besides invoking a function, a contract may also delegate execution to it using our custom, high-level delegation, which is captured by the DELEG rule:

$$\text{DELEG} \quad \frac{\begin{array}{c}\sigma = (\Psi, M, \mathbf{s}, \kappa), \quad s_1, \ldots, s_{|\mathcal{C}|} = \mathbf{s}, \quad s_i \neq \textit{destroyed} \\ t \in T_i, \quad \textit{name} = t^{name} \\ \textit{Eval}(\sigma, g_t) \to \langle \sigma, R[\texttt{true}] \rangle \\ \langle (\sigma, N), a_t \rangle \to \langle (\sigma', x), \cdot \rangle, \quad x \neq E\end{array}}{\textit{Eval}\,(\sigma, i.\texttt{delegate}.\textit{name}(v_1, v_2, \ldots)) \to \langle \sigma', N \rangle}$$

If the target contract has been destroyed, then our custom, high-level delegation will finish without any effects, which is captured the DELEG-DES rule:

$$\text{DELEG-DES} \quad \frac{\sigma = (\Psi, M, \mathbf{s}, \kappa), \quad s_1, \ldots, s_{|\mathcal{C}|} = \mathbf{s}, \quad s_i = \textit{destroyed}}{\textit{Eval}\,(\sigma, i.\texttt{delegate}.\textit{name}(v_1, v_2, \ldots)) \to \langle \sigma, N \rangle}$$

Finally, exceptions are "rethrown," which is captured by the DELEG-EXC rule:

$$\text{DELEG-EXC} \quad \frac{\begin{array}{c}\sigma = (\Psi, M, \mathbf{s}, \kappa), \quad s_1, \ldots, s_{|\mathcal{C}|} = \mathbf{s}, \quad s_i \neq \textit{destroyed} \\ t \in T_i, \quad \textit{name} = t^{name} \\ \textit{Eval}(\sigma, g_t) \to \langle \sigma, R[\texttt{true}] \rangle \\ \langle (\sigma, N), a_t \rangle \to \langle (\sigma', E), \cdot \rangle\end{array}}{\textit{Eval}\,(\sigma, i.\texttt{delegate}.\textit{name}(v_1, v_2, \ldots)) \to \langle \sigma', E \rangle}$$

For the semantics of the supported statements (i.e., specifying $\langle (\sigma, x), a \rangle \to \langle (\sigma', x'), \cdot \rangle$), we use the standard Solidity semantics, which have been formalized using small-step operational semantics in [1].

REFERENCES

[1] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-design smart contracts for Ethereum," in *Proceedings of the 23rd International Conference on Financial Cryptography and Data Security (FC)*, February 2019.

[2] Solidity Documentation, "Breaking changes," https://solidity.readthedocs.io/en/v0.5.7/050-breaking-changes.html, 2018, accessed on 9/24/2019.