# An Automated Analyzer for Financial Security of Ethereum Smart Contracts

Wansen Wang[1] Wenchao Huang[1] Zhaoyi Meng[1] Yan Xiong[1] Fuyou Miao[1] Xianjin Fang[2]
Caichang Tu[1] Renjie Ji[1]
[1]University of Science and Technology of China, Anhui, China
[2]Anhui University of Science and Technology, Anhui, China

## ABSTRACT

At present, millions of Ethereum smart contracts are created per year and attract financially motivated attackers. However, existing analyzers do not meet the need to precisely analyze the financial security of large numbers of contracts. In this paper, we propose and implement FASVERIF, an automated analyzer for fine-grained analysis of smart contracts' financial security. On the one hand, FASVERIF automatically generates models to be verified against security properties of smart contracts. On the other hand, our analyzer automatically generates the security properties, which is different from existing formal verifiers for smart contracts. As a result, FASVERIF can automatically process source code of smart contracts, and uses formal methods whenever possible to simultaneously maximize its accuracy.

We evaluate FASVERIF on a vulnerabilities dataset by comparing it with other automatic tools. Our evaluation shows that FASVERIF greatly outperforms the representative tools using different technologies, with respect to accuracy and coverage of types of vulnerabilities.

## KEYWORDS

Smart Contract; Formal Verification

## 1 INTRODUCTION

Smart contracts on Ethereum have been applied in many fields such as financial industry [5], public sector [6], *etc.*, while the market cap of the Ethereum cryptocurrency, *i.e.*, ethers, grows up to $177 billions on July 27, 2022 [19]. Unfortunately, widely usage of smart contracts and an increase in the value of ethers make smart contracts become attractive targets for attackers. The infamous vulnerability in the DAO contract led to losses of $150M in June 2016 [3]. In July 2017, $30M worth of ethers were stolen from Parity wallet due to a wrong function [7]. Most recently, there were $27M worth of ethers stolen from the Poly Network contract in August 2021 [18]. It is therefore necessary to guarantee the security of smart contracts to avoid financial losses.

Current security analyzers for smart contracts can be divided into the following three categories: automation-oriented, expressivity-oriented and hybrid-oriented. The automation-oriented approaches [47][35][41] support analysis on a great amount of smart contracts without manual intervention, motivated by the fact that 10.7 million contracts are created in 2020 [44]. However, the analysis is not accurate enough to avoid financial loss since smart contracts manage assets worth millions of dollars [62]. The expressivity-oriented approaches [59][50] tend to achieve great accuracy and high coverage of security properties by using formal verification on manually defined security properties. The requirement for manual intervention makes it difficult for them to analyze a large number of contracts. The hybrid-oriented approaches can automatically analyze smart contracts using formal verification. To the best of our knowledge, there are two hybrid-oriented tools eThor [56] and ZEUS [43]. Nonetheless, as the literature of eThor states, eThor can only detect reentrancy [13] automatically. ZEUS defines policies for detecting vulnerabilities based on known attack patterns, thus it may miss new types of vulnerabilities. In summary, existing analyzers are not sufficient to analyze the financial security of massive contracts precisely.

We propose and implement FASVERIF, a system of automated inference [55][34], a static reasoning mechanism where the properties are expected to be automatically derived, for achieving full automation on fine-grained security analysis of Ethereum smart contracts. Firstly, FASVERIF can verify security properties of smart contracts automatically. Secondly, besides automated verification, our system of automated inferencealso automatically generates finance-related security properties and the corresponding models used for the verification. In general, the goal of FASVERIF is to analyze the financial security of massive contracts accurately, whereby the security properties are generated automatically based on our statistical analysis, and the rest of the process, including the modeling and verification, are formally implemented. Moreover, FASVERIF generates properties according to the financial losses caused by vulnerabilities instead of the attack patterns for known vulnerabilities, thus it covers various types of vulnerabilities.

We collect a vulnerabilities dataset consisting of 548 contracts from other works [41][58][45], and evaluate FASVERIF on it with other automatic tools. Our evaluation shows that FASVERIF greatly outperforms the representative tools using different technologies, in which it achieves higher accuracy and F1 values in detection of various types of vulnerabilities. We also evaluate FASVERIF on 1413 contracts randomly selected from a real-world dataset. FASVERIF finds 13 contracts deployed on Ethereum with zero-day bugs, including 10 contracts with vulnerabilities of transferMint [11] that can evade the detection of current automatic tools to the best of our knowledge. Here we consider a vulnerability zero-day when it is exploitable by attackers [28].

In summary, this paper makes the following contributions:

1) We propose a novel framework for achieving automated inference. Given the source code of a smart contract, the framework generates an independent model and the finance-related security properties, which are then used for generating customized models.

2) We propose the method of property generation, according to the statistical analysis on 30577 smart contracts. Specifically, we design 2 types of properties, invariant properties and equivalence

properties, which correspond to various finance-related vulnerabilities, *e.g.*, gasless send [9].

3) We implement FASVERIF for supporting modeling and verification, where we modify the source code of Tamarin prover [48], which is the state-of-the-art tool for verifying security protocols, to support both trace properties of reachability and numerical constraint solving for finance-related properties.

4) We prove the soundness of verifying the invariant properties and equivalence properties using our translated model based on a custom semantics of Solidity [42].

5) We evaluate the effectiveness of FASVERIF and find 13 contracts with zero-day vulnerabilities by FASVERIF.

## 2 PRELIMINARIES

### 2.1 Smart contracts on Ethereum

Ethereum is a blockchain platform that supports two types of accounts: contract accounts, and external accounts. Each account has an ether balance and a unique address. The contract account is associated with a piece of code called a smart contract, which controls the behaviors of the account, and a storage that stores global variables denoting the state of the account. External accounts are controlled by humans without associated code or global variables.

A function in the smart contract can be invoked by transactions sent by external accounts. A transaction is packed into a block by the miner and when that block is published into the blockchain, the function invoked by the transaction is executed. Functions can also be invoked by internal transactions sent by contract accounts and the sending of an internal transaction can only be triggered by another transaction or internal transaction.

### 2.2 Solidity programming language

$$\begin{aligned}
\texttt{Contract C} &:= \texttt{contract}\, n_c \left\{\mathsf{d}, \texttt{func}^{+}\right\} \\
\texttt{Func func} &:= \texttt{function}\, f(\mathsf{d})\ \{\texttt{stmt}\} \\
\texttt{Stmt stmt} &:= \texttt{stmt}_{\mathsf{A}} \mid \texttt{if}\ \mathsf{e_b}\ \texttt{then stmt else stmt} \mid \texttt{stmt; stmt} \\
\texttt{Atom Stmt stmt}_{\mathsf{A}} &:= v \leftarrow \mathsf{e} \mid \tau\ v \leftarrow \mathsf{e} \mid \texttt{require}\ \mathsf{e_b} \mid \mathsf{e_c} \mid \texttt{return} \\
\texttt{Bool Expr e}_{\mathsf{b}} &:= \mathsf{e} == \mathsf{e} \mid \mathsf{e} < \mathsf{e} \mid \mathsf{e} > \mathsf{e} \mid \mathsf{e}! = \mathsf{e} \\
\texttt{Call Expr e}_{\mathsf{c}} &:= v.\texttt{transfer}(v) \mid v.\texttt{send}(v) \mid \\
&\quad\ v.\texttt{call}().\texttt{value}(v) \mid n_c(v).f(\mathsf{p}) \\
\texttt{Expr e} &:= v \mid v + \mathsf{e} \mid v - \mathsf{e} \mid v * \mathsf{e} \mid v/\mathsf{e} \mid v\%\mathsf{e} \mid \\
&\quad\ v * * \mathsf{e} \\
\texttt{Type}\ \tau &:= \tau_B \mid \tau_B \mapsto \tau \mid n_c \\
\texttt{BasicType}\ \tau_{\mathsf{B}} &:= \texttt{uint} \mid \texttt{bool} \mid \texttt{address} \\
\texttt{Var Declare d} &:= \tau\ v \mid \tau\ v; \mathsf{d} \\
\texttt{Param p} &:= v \mid v, \mathsf{p}
\end{aligned}$$

**Figure 1: Core subset of Solidity**

The most popular programming language for smart contracts is Solidity [25]. We take the smart contracts written in Solidity as the object of study in this paper. For brevity, we focus on a core subset of Solidity as shown in Fig. 1. A contract C consists of a name $n_c$, global variable declarations d and functions $\texttt{func}^{+}$. $\texttt{func}^{+}$ represents a sequence of functions, and each function func is denoted as a function name $f$, parameters declarations d and a body statement s. A statement stmt can be an atomic statement $\texttt{stmt}_{\mathsf{A}}$, a conditional statement or a sequence of statements. An atom statement $\texttt{stmt}_{\mathsf{A}}$ is an assignment of a variable $v \leftarrow e$, a declaration and assignment of a new variable $\tau\ v \leftarrow e$, a require statement, a call expression $\mathsf{e_c}$ or a return statement. A call expression $\mathsf{e_c}$ stands for an expression used to invoke official functions transfer, send, call of Solidity or custom function of contracts. The variables used in stmt fall into the following types: 1) basic types $\tau_B$, *i.e.*, uint,bool and address. 2) $\tau_B \mapsto \tau$ denoting a mapping from variables of type $\tau_B$ to variables of type $\tau$. 3) $n_c$ denoting a contract. Additionally, there are some special built-in variables of Solidity which cannot be assigned: 1) block.timestamp denoting the timestamp of the block that contains the transaction that triggers the execution of stmt. 2) $c$.balance expressing the ether balance of contract in address $c$. 3) msg.sender denoting the address of the sender of the transaction that triggers the execution of stmt. There are four types of visibilities for functions in smart contracts: public, private, external and internal. Since the functions of different visibilities are handled in similar ways, we only introduce how to process public functions in this paper for brevity while FASVERIF supports analysis of all kinds of them.

### 2.3 Multiset rewriting system

When a protocol is modeled as a multiset rewriting system, the nodes in the system represent the states that a run of the protocol may reach, and the paths represent the possible protocol runs. Each state in the system is a multiset of facts, denoted as $F(t_1, \ldots, t_n)$, where $F$ is a fact symbol, and $t_1, \ldots, t_n$ are terms. Facts model resources that can be consumed, which is applied by labeled rewriting rules in Tamarin prover. A labeled rewriting rule is denoted as $l - [a] \rightarrow r$, where $l$, $a$ and $r$ are three parts called premise, action, and conclusion, respectively. The rule is applicable to state $s$, if the multiset of facts in $l$ is included in $s$. To obtain the successor state $s'$, the consumed facts are removed, and the facts in $r$ are added. The action $a$ is also a multiset of facts representing the label of the rule. Meanwhile, global restrictions on facts in $a$ can be made such that the execution of the protocol can be further restrained. In conclusion, an execution of the protocol is the sequence consisting of the states, and the instances of applied rules along a path, where a trace of the execution is the sequence of the labels in the instances of applied rules. FASVERIF leverages the multiset rewriting system in Tamarin prover [48] to model smart contracts and attackers flexibly. We can adjust a model to cover new types of attacks by only modifying or adding several multiset rewriting rules without targeted to specified contracts. In comparison, existing tools, *e.g.*, ZEUS and eThor do not model the behaviors of attackers. Thus, to detect new types of attacks, these tools may need the users to provide contract-specific properties for different contracts, which is costly. Considering the diversification of attackers' methods [16] in Ethereum, the flexible modelling of FASVERIF makes it more practical.

## 3 OVERVIEW

### 3.1 Design of FASVERIF

As shown in Fig. 2, FASVERIF contains 4 modules:

(1) **Independent modeling**: given the source code of a smart contract as input, the module generates a partial model

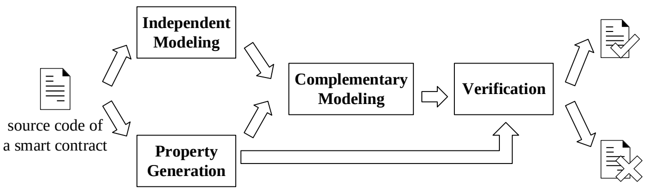

**Figure 2: Design of FASVERIF.**

of the contract, which gives the initial state of the running contract and general rules for state transitions. It translates the contract, as well as the possible behaviors of adversaries, into the model, which is independent of specific security properties. Note that this partial model cannot be verified directly.

(2) **Property generation**: FASVERIF then generates a set of security properties that the smart contract should satisfy. Since some vulnerabilities may exist only if the specified functions, *e.g.*, `call`, are used in the smart contract, the properties are generated on-demand.
(3) **Complementary modeling**: the module outputs additional rules for each property to complement the partial model, and tries to minimize the size of the model for different security properties.
(4) **Verification**: we finally design the method of verification to determine whether the properties are valid. We also modify the source code of Tamarin prover for supporting the verification where numerical constraint solving is additionally required.

## 3.2 Adversary model

We assume that the adversaries can launch attacks by leveraging the abilities of three types of entities: external accounts, contract accounts, miners. The concerned attacks on a smart contract are processes that affect the variables related to the smart contract and thus the results of the smart contract executions. The variables that can be manipulated by the adversary fall into two categories: global variables of contracts and `block.timestamp`. An external account or a contract account needs to invoke functions in victim contracts to change the values of their global variables, while a miner can manipulate `block.timestamp` in a range [4][24]. In summary, we assume that the adversary can perform the following operations:

C1. Sending a transaction to invoke any function in victim contracts with any parameters.
C2. Implementing a fallback function to send an internal call message. This message can invoke any function in victim contracts with any parameters.
C3. Increasing the timestamp of a block by up to 15 seconds [4][24].

# 4 INDEPENDENT MODELING

Given a smart contract, the module of independent modeling automatically outputs general rules of modeling the behaviors of external accounts, the account who owns the contract, and the adversaries. The rules in the multiset rewriting system correspond to the transitions of the configurations of Solidity. Therefore, we firstly define terms used in the rules, and sequences using the terms for describing the configurations. Then, we show the processes of modeling the behaviors using the terms. Finally, a comprehensive example is given to illustrate the usage of the rules, and discussions are made on technical challenges of property generation and complementary modeling based on the independent modeling.

## 4.1 Terms and sequences

The terms in multiset rewriting system are translated from the names in Solidity language. There are two types of terms: constant terms and variable terms. Correspondingly, as shown in Fig 1, a name $v$ in Solidity may represent a contract, a function, a variable or a constant. Therefore, given a name $v$, we compute a tuple $\langle name, type, range, ether\rangle$. Here, $name$ is a term used in multiset rewriting system, which corresponds to $v$. Term $type \in \{\mathsf{T_v}, \mathsf{T_c}\}$. If $v$ is a variable, $type = \mathsf{T_v}$; otherwise, $type = \mathsf{T_c}$. Term $range \in \{\mathsf{R_g}, \mathsf{R_l}, \mathsf{R_o}\}$. If $v$ is a global variable and a local variable, *i.e.*, a variable defined inside a function, $range = \mathsf{R_g}$ and $\mathsf{R_l}$, respectively; otherwise $range = \mathsf{R_o}$. If $v$ is a global variable representing an account's ether, $ether = \mathsf{E_y}$; otherwise $range = \mathsf{E_n}$. Since value of $v$ is unchanged if $type = \mathsf{T_c}$, in this case $name$ is assigned with the value of $v$; otherwise, $name = v$.

Denote $[\![e_1, e_2, ..., e_n]\!]$ as a sequence, where each element $e_i$ has the same type, *i.e.*, a term, a name, or the aforementioned tuple. $T_1 \cdot T_2$ represents the concatenation of sequence $T_1$ and $T_2$. $T|\frac{t}{t'}$ is a sequence obtained by replacing element $t$ of sequence $T$ with another element $t'$. $T_1 \backslash T_2$ represents a new sequence by removing all the elements in sequence $T_1$ that are the same as those in sequence $T_2$. We additionally define operations for a tuple sequence $\omega$. Here, $\omega[j]$ indicates $name$ of the $j$th tuple in $\omega$. $\sigma(\omega)$ outputs a term sequence consisting of all the $name$ in $\omega$. $g(\omega)$ outputs a term sequence by obtaining the $name$ of all tuples where $range = \mathsf{R_g}$ in $\omega$. Similarly, $l(\omega)$ and $e(\omega)$ are defined for obtaining sequences with $name$ of tuples in $\omega$ whose $range = \mathsf{R_l}$ and $ether = \mathsf{E_y}$, respectively. The order of $\sigma(\omega), g(\omega), l(\omega), e(\omega)$, are in accordance of the order of $\omega$. For instance, if the $i$th and $j$th name in $g(\omega)$ corresponds to the $a$th and $b$th tuple in $\omega$, respectively, $i < j \Leftrightarrow a < b$.

Furthermore, to translate names into terms, we define and implement two functions $\sigma_v, \sigma_a$. $\sigma_v$ translates a variable name into a variable term, and $\sigma_a$ translates a name that represents a contract, a function, or a constant into a constant term.

## 4.2 Modeling the behaviors

Based on the above notations, we propose to model the initialization of contracts and transitions of Solidity's configurations caused by the actions of external accounts, contract accounts, and adversaries. Specifically, given a contract account of address $c$, we will introduce how to model the executions of functions in the contract codes of the account. For brevity, we will refer to the account of address $c$ as account $c$ in the following text.

**Modeling the initialization.** Assume that the contract of account $c$ is deployed on blockchain and the following data will be initialized: 1) the ether balances of account $c$; 2) the global variables of account $c$. Besides, the ether balances of other accounts also need to be initialized since they may be modified during the executions of codes of account $c$. We use $\omega_0$ to denote the context, *i.e.*, the model of the configuration of Solidity, after initialization of account $c$. There are three kinds of tuples in $\omega_0$ in order: 1) $\langle\sigma_a(c), \mathsf{T_c}, \mathsf{R_o}\rangle$ that represents the address of account $c$; 2) tuple sequence $g(\omega_0)\backslash e(\omega_0)$ denoting the global variables of account $c$; 3) tuple sequence $e(\omega_0)$ denoting the ether balances of all accounts. Therefore, $\omega_0[1] = \sigma_a(c)$. The tuples in the context are then used to determine the order of parameters of facts in generated rules. Therefore, we define the following rules to model the initialization:

$$[\mathsf{FR}(e(\omega_0))]-[\mathsf{Init_E}()] \rightarrow [\mathsf{Evar}(e(\omega_0))] \qquad (\texttt{init\_evars})$$

$$[\mathsf{FR}(g(\omega_0)\backslash e(\omega_0))] - [\mathsf{Init_G}(\omega_0[1])] \rightarrow$$
$$[\mathsf{Gvar}([\![\omega_0[1]]\!] \cdot g(\omega_0)\backslash e(\omega_0))] \qquad (\texttt{init\_gvars})$$

Here, $\mathsf{Evar}(e(\omega_0))$ represents the current ether balances of all accounts on blockchain in initialization. Gvar represents the current global variables of account $c$. For brevity, we use $\mathsf{FR}(e(\omega_0))$ to denote a sequence that consists of $\mathsf{Fr}(t)$ for all elements $t$ in $e(\omega_0)$. $\mathsf{Fr}(t)$ here is a built-in fact of Tamarin prover [48] that denotes a freshly generated name. We use this fact to denote a term with arbitrary initial values. In practical scenarios, the ether balances of all accounts can be only initialized once and the global variables can be initialized once for every contract account. To ensure that the application of these rules accords with practice, we associate restrictions with the above rules. For instance, the restriction *All #i #j*.$\mathsf{Init_E}()$@*i*& $\mathsf{Init_E}()$@*j* => $i = j$ is added for `init_evars`. Here, $i$ and $j$ denote time points. The restriction can be understood as "the action $\mathsf{Init_E}()$ cannot occur at two different time points".

**Translation of invocation of functions.** After initialization, external accounts can send transactions to invoke any function in the contract of $c$. To model the invocation of functions, we define $\mathcal{R}$ to recursively translate a function in the contract into rules as shown in Fig. 3. Generally, in each recursive step, $\mathcal{R}$ translates a fragment of codes into a rule or multiple rules and leaves the translation of the rest in the next steps. The first argument of $\mathcal{R}$ represents the codes as shown in Fig. 3. If the first argument is a sequence of statements, the second argument $i$ is a string encoding the position of the sequence in its function and $i \circ a$ denotes a string obtained by concatenating $i$ and a character $a$; otherwise, if the first argument is a function, the second argument is an empty string $\varnothing$. The third argument is a tuple sequence $\omega$ representing the context.

For each function $f$ in the contract of account $c$, $\mathcal{R}(\texttt{function}\, f(\mathsf{d})$ $\{\texttt{stmt}\}, \varnothing, \omega_0)$ is applied in the first recursive step, which outputs three rules: `ext_call`, `recv_ext` and `recv_in`. A function can be invoked by a transaction from an external account or an internal transaction from a contract account. We introduce a variable $c_b$ to denote the address of the account sending the transaction or internal transaction. The rule `ext_call` represents an event that an external account sends a transaction. Fact $\mathsf{Call_e}$ represents the results after the event, where $seq(\mathsf{d})$ represents the parameters in function $f$. According to assumption C1 in adversary model, $c_b$ and $seq(\mathsf{d})$ are initialized by using $\mathsf{Fr}(c_b)$, $\mathsf{FR}(\sigma(seq(\mathsf{d})))$. Correspondingly, fact $\mathsf{Call_{in}}$ represents the results after a contract account sends the internal transaction. The rules for fact $\mathsf{Call_{in}}$ are generated in the process of translating statements. Rule `recv_ext` and `recv_in` denote the reception of a transaction and an internal transaction, respectively. The $\mathsf{Var_1}$ fact represents all the values required in executing $f$, whereby terms in $\mathsf{Call_e}/\mathsf{Call_{in}}$, Gvar and Evar are merged into terms in $\mathsf{Var_1}$. Therefore, $\mathcal{R}$ also updates $\omega_0$ with a sequence of the corresponding tuples. Here, $calltype \in \{\mathsf{EXT}, \mathsf{IN}\}$ indicating whether the account who invokes function $f$ is an external account or a contract account, depth denotes current call depth. Specially, if a statement in function $f$ uses `block.timestamp`, rule `ext_call_bvar` is generated instead of `ext_call`, which is shown in Appendix A.1.

**Translation of statements.** After a function is invoked, it starts to be executed, thereby $\mathcal{R}$ translates the statements in the function into rules for modeling the execution of the function in account $c$.

Similar to $\mathsf{Var_1}$, fact $\mathsf{Var}_i$ represents the configuration of Solidity when its PC is at a certain point, denoted as index $i$. After a recursive step of $\mathcal{R}$, if the step is to translate successive codes of a conditional statement, the index $i$ is updated to $i \circ a$, where $a$ indicates which branch of codes the step chooses to translate; otherwise, $a = 1$. The translated rules serve two purposes: **1)** modeling the branches of executions. **2)** modeling the modification of values of the variables. For example, the rule for an assignment statement can result in replacement of variable terms in the facts.

The function $\theta_e(e)$ translates mathematical expressions in functions into terms in rules as follows:

$$\theta_e(e) = \begin{cases} \sigma_a(e) & \text{if } e \text{ is a constant} \\ \sigma_v(e) & \text{if } e \text{ is a variable} \\ \theta_o(e_1, e_2, \Diamond) & \text{if } e \text{ is } e_1 \Diamond e_2 \\ \mathsf{EqNum}(\theta_e(e_1), \theta_e(e_2)) & \text{if } e \text{ is } e_1 = e_2 \\ \mathsf{NeNum}(\theta_e(e_1), \theta_e(e_2)) & \text{if } e \text{ is } e_1 \neq e_2 \\ \mathsf{LessNum}(\theta_e(e_1), \theta_e(e_2)) & \text{if } e \text{ is } e_1 < e_2 \end{cases}$$

Here, conditional facts EqNum, NeqNum, LessNum denote the relationships between numeric variables. $\Diamond \in \{+, -, *, /, \%, **\}$ represents an operator in expressions. $\theta_o(e_1, e_2, \Diamond)$ represents the term translated from $e_1 \Diamond e_2$. Conditional facts are processed in the verification module of FASVERIF, which will be introduced in Section 6.2. Similarly, we define $\theta_{ne}(e)$ to translate the negation of $e$.

We only introduce the translation of statements in three categories here, the complete translation is shown in Appendix A.1.

**1)** Assignment statements. Taking an assignment statement $v_1 \leftarrow v_2$ as an example, rule `var_assign` is generated such that term $\sigma_v(v_1)$ is replaced by $\sigma_v(v_2)$ when applying the rule. If the variable $v_1$ in an assignment statement is newly declared, *e.g.*, $\tau\, v_1 \leftarrow v_2$, rule `var_declare` is generated. Rule `var_declare` is along with an additional term $\sigma_v(v_2)$ and $\omega$ will be added with a new tuple, which means that a new variable is introduced.

**2)** Return statements. Two rules `ret_ext` and `ret_in` are translated from the statement `return`. When the `return` statement of function $f$ is executed, the local variables will no longer be used and not be maintained. If the function is invoked by an external account, the statement is translated into `ret_ext`. The terms denoting global variables of account $c$ and the ether balances of all accounts are put into Gvar and Evar facts respectively. If $f$ is called by an internal transaction, the statement `return` is translated into `ret_in`, where term $\omega[4]$ denoting the address of the contract who has $f$, term $\omega[1]$ denoting function name $f$, term $\omega[2]$ representing the address of the contract who invokes $f$, and $\omega[4]$ denoting current call depth are set as parameters in fact Return. Pred_eq here is a fact provided by [48] to denote equality between terms. We use it

$$
\begin{aligned}
&\mathcal{R}(\mathtt{function}\ f(\mathsf{d})\{\mathtt{stmt}\}, \varnothing, \omega_0) = \mathcal{R}(\mathtt{stmt}, 1, \llbracket \langle \sigma_a(f), \mathsf{T_c}, \mathsf{R_o}, \mathsf{E_n}\rangle, \langle \sigma_v(c_b), \mathsf{T_v}, \mathsf{R_l}, \mathsf{E_n}\rangle, \langle \sigma_v(calltype), \mathsf{T_v}, \mathsf{R_l}, \mathsf{E_n}\rangle, \langle \sigma_v(\mathsf{depth}), \mathsf{T_v}, \mathsf{R_l}, \mathsf{E_n}\rangle \rrbracket \\
&\quad \cdot \omega_0 \cdot \boldsymbol{seq}(\mathsf{d})) \cup \{[\mathsf{Fr}(\sigma_v(c_b)), \mathsf{FR}(\sigma(\boldsymbol{seq}(\mathsf{d})))] - [] \rightarrow [\mathsf{Call_e}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b)\rrbracket \cdot \sigma(\boldsymbol{seq}(\mathsf{d})))], \qquad (\mathtt{ext_call}) \\
&[\mathsf{Call_e}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b)\rrbracket \cdot \sigma(\boldsymbol{seq}(\mathsf{d}))), \mathsf{Evar}(e(\omega_0)), \mathsf{Gvar}(\llbracket \omega_0[1]\rrbracket \cdot g(\omega_0)\backslash e(\omega_0))] - [] \rightarrow [\mathsf{Var}_1(\llbracket \sigma_a(f), \sigma_v(c_b), \mathsf{EXT}, 0\rrbracket \cdot \sigma(\omega_0) \cdot \sigma(\boldsymbol{seq}(\mathsf{d})))], \qquad (\mathtt{recv_ext}) \\
&[\mathsf{Call_{in}}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b), \sigma_v(\mathsf{depth})\rrbracket \cdot \sigma(\boldsymbol{seq}(\mathsf{d}))), \mathsf{Evar}(e(\omega_0)), \mathsf{Gvar}(\llbracket \omega_0[1]\rrbracket \cdot g(\omega_0)\backslash e(\omega_0))] - [] \rightarrow \\
&\quad [\mathsf{Var}_1(\llbracket \sigma_a(f), \sigma_v(c_b), \mathsf{IN}, \sigma_v(\mathsf{depth})\rrbracket \cdot \sigma(\omega_0) \cdot \sigma(\boldsymbol{seq}(\mathsf{d})))]\} \qquad (\mathtt{recv_in}) \\
&\mathcal{R}(v_1 \leftarrow v_2; \mathtt{stmt}, i, \omega) = \mathcal{R}(\mathtt{stmt}, i \circ 1, \omega) \cup \{[\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 1}(\sigma(\omega)|\frac{\sigma_v(v_1)}{\sigma_v(v_2)})]\} \qquad (\mathtt{var_assign}) \\
&\mathcal{R}(\tau\ v_1 \leftarrow v_2; \mathtt{stmt}, i, \omega) = \mathcal{R}(\mathtt{stmt}, i \circ 1, \omega \cdot \llbracket \langle \sigma_v(v_1), \mathsf{T_v}, \mathsf{R_l}, \mathsf{E_n}\rangle \rrbracket) \cup \{[\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 1}(\sigma(\omega) \cdot \llbracket \sigma_v(v_2)\rrbracket)]\} \qquad (\mathtt{var_declare}) \\
&\mathcal{R}(\mathtt{return}, i, \omega) = \{[\mathsf{Var}_i(\sigma(\omega))] - [\mathsf{Pred_eq}(\omega[3], \mathsf{EXT})] \rightarrow [\mathsf{Gvar}(\llbracket \omega[5]\rrbracket \cdot g(\omega)\backslash e(\omega)), \mathsf{Evar}(e(\omega))], \qquad (\mathtt{ret_ext}) \\
&\quad [\mathsf{Var}_i(\sigma(\omega))] - [\mathsf{Pred_eq}(\omega[3], \mathsf{IN})] \rightarrow [\mathsf{Return}(\llbracket \omega[5], \omega[1], \omega[2], \omega[4]\rrbracket, \mathsf{Gvar}(\llbracket \omega[5]\rrbracket \cdot g(\omega)\backslash e(\omega)), \mathsf{Evar}(e(\omega)))]\} \qquad (\mathtt{ret_in}) \\
&\mathcal{R}(c_x.\mathtt{call}().\mathtt{value}(v_1); \mathtt{stmt}, i, \omega) = \mathcal{R}(\mathtt{stmt}, i \circ 1, \omega) \cup \mathcal{R}(\mathtt{stmt}, i \circ 2, \omega) \cup \mathcal{R}(\mathtt{stmt}, i \circ 3 \circ 1, \omega) \cup \\
&\quad \{[\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 1}(\sigma(\omega)|\frac{\sigma_v(c_x)}{\sigma_v(c_x) \oplus \sigma_v(v_1)}|\frac{\sigma_v(c)}{\sigma_v(c) \ominus \sigma_v(v_1)})], \qquad (\mathtt{ether_succ}) \\
&\quad [\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 2}(\sigma(\omega))], \qquad (\mathtt{ether_fail}) \\
&\quad [\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 3}(l(\omega)), \mathsf{Fallback}(\llbracket \omega[5], \omega[1]\rrbracket), \mathsf{Gvar}(\llbracket \omega[5]\rrbracket \cdot g(\omega)\backslash e(\omega)), \mathsf{Evar}(e(\omega)|\frac{\sigma_v(c_x)}{\sigma_v(c_x) \oplus \sigma_v(v_1)}|\frac{\sigma_v(c)}{\sigma_v(c) \ominus \sigma_v(v_1)})], \qquad (\mathtt{fb_call}) \\
&\quad [\mathsf{ReturnFallback}(\llbracket \omega[5], \omega[1]\rrbracket), \mathsf{Gvar}(\llbracket \omega[5]\rrbracket \cdot g(\omega)\backslash e(\omega)), \mathsf{Evar}(e(\omega)), \mathsf{Var}_{i\circ 3}(l(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 3\circ 1}(\sigma(\omega))]\} \qquad (\mathtt{recv_fb_ret})
\end{aligned}
$$

**Figure 3: Translation of statements.**

to determine whether $\omega[3]$, *i.e.*, $\sigma_v(calltype)$ is equal to EXT or IN, corresponding to the above two cases respectively.

**3)** Ether transfer statements. The misuse of ether transfer statements using `call` is one of the reasons that cause attacks. Consider a statement $c_x$`.call().value(`$v_1$`)`, which means that the account $c$ who invokes the `call()` is to transfer ether $v_1$ to the account $c_x$, where $v_1$ is assumed as a local variable. There are three cases for the execution of the statement: **a)** the transfer succeeds. The ether balance of $c$ is reduced by $v_1$ and the ether balance of $c_x$ is increased by $v_1$. **b)** the transfer fails and the ether balances of $c$ and $c_x$ are not modified. **c)** the transfer succeeds with ether balances changed in the same way as case **a)**, but the fallback function is called probably in an unexpected way. The rules `ether_succ`, `ether_fail`, `fb_call` are generated for the 3 cases respectively. In these rules, we use $\sigma_v(c)$ and $\sigma_v(c_x)$ to denote the ether balances of $c$ and $c_x$. In rule `fb_call`, fact Fallback indicates that the fallback function is called. According to assumption C2 in the adversary model, the global variables of $c$ and ether balances of all accounts may be modified due to execution of the fallback function, thereby the terms denoting these variables are put into Gvar and Evar facts, while terms representing local variables are maintained in $\mathsf{Var}_{i\circ 3}$. In rule `recv_fb_ret`, ReturnFallback implies that the fallback function finishes executing and a return message is sent to $c$. The terms in $\mathsf{Var}_{i\circ 3}$, Gvar and Evar are merged back into terms in $\mathsf{Var}_{i\circ 3\circ 1}$, which indicates that the function $f$ continues executing.

**Adversaries.** The following rules are generated for modeling the capability C1, C2 of adversaries as mentioned in Section 3.2. The modeling of capability C3 will be introduced in Section 6.1.

**C1:** The operation that an adversary, besides normal participants, sends a transaction can also be modeled by rule `ext_call`. Therefore, no additional rules for the operation are provided.

**C2:** Since the fallback function of the adversary will only be triggered by ether transfer statements using `call`, the following rules are generated for each function $f$ in the contract of account $c$:

$$
\begin{aligned}
&[\mathsf{Fallback}(\llbracket \sigma_a(c), \sigma_a(f)\rrbracket)] - [] \rightarrow \\
&\quad [\mathsf{Call_{in}}(\llbracket \sigma_a(c), \sigma_a(f'), \sigma_a(c_{adv})\rrbracket \cdot \sigma(seq(d')))] \qquad (\mathtt{fb_in_call})
\end{aligned}
$$

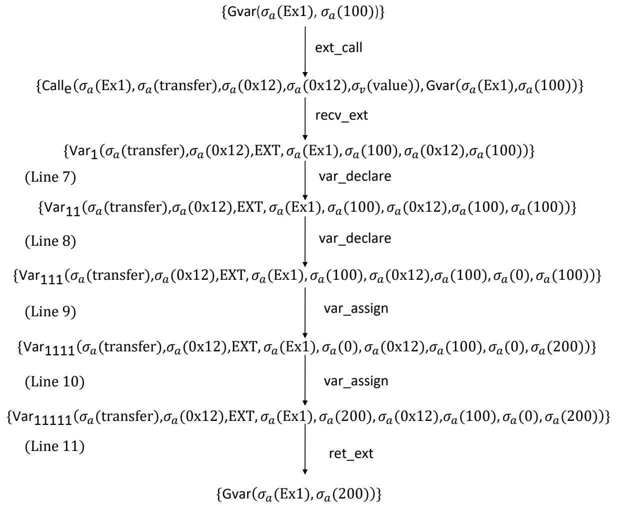
{Gvar($\sigma_a$(Ex1), $\sigma_a$(100))}
ext_call
{Call$_e$($\sigma_a$(Ex1), $\sigma_a$(transfer),$\sigma_a$(0x12),$\sigma_a$(0x12),$\sigma_v$(value)), Gvar($\sigma_a$(Ex1),$\sigma_a$(100))}
recv_ext
{Var$_1$($\sigma_a$(transfer),$\sigma_a$(0x12),EXT, $\sigma_a$(Ex1), $\sigma_a$(100), $\sigma_a$(0x12),$\sigma_a$(100))}
(Line 7)
var_declare
{Var$_{11}$($\sigma_a$(transfer),$\sigma_a$(0x12),EXT, $\sigma_a$(Ex1), $\sigma_a$(100), $\sigma_a$(0x12),$\sigma_a$(100), $\sigma_a$(100))}
(Line 8)
var_declare
{Var$_{111}$($\sigma_a$(transfer),$\sigma_a$(0x12),EXT, $\sigma_a$(Ex1), $\sigma_a$(100), $\sigma_a$(0x12),$\sigma_a$(100), $\sigma_a$(0), $\sigma_a$(100))}
(Line 9)
var_assign
{Var$_{1111}$($\sigma_a$(transfer),$\sigma_a$(0x12),EXT, $\sigma_a$(Ex1), $\sigma_a$(0), $\sigma_a$(0x12),$\sigma_a$(100), $\sigma_a$(0), $\sigma_a$(200))}
(Line 10)
var_assign
{Var$_{11111}$($\sigma_a$(transfer),$\sigma_a$(0x12),EXT, $\sigma_a$(Ex1), $\sigma_a$(200), $\sigma_a$(0x12),$\sigma_a$(100), $\sigma_a$(0), $\sigma_a$(200))}
(Line 11)
ret_ext
{Gvar($\sigma_a$(Ex1), $\sigma_a$(200))}

**Figure 4: The execution that models an attack on Ex1.**

$$
\begin{aligned}
&[\mathsf{Return}(\llbracket \sigma_a(c), \sigma_a(f'), \sigma_a(c_{adv})\rrbracket)] - [] \rightarrow \\
&\qquad [\mathsf{ReturnFallback}(\llbracket \sigma_a(c), \sigma_a(f)\rrbracket)] \qquad (\mathtt{ret_fb})
\end{aligned}
$$

Here, $c_{adv}$ represents the address of the contract account owned by the adversary. $f'$ denotes an arbitrary function in the contract of account $c$, and $d'$ denotes the parameters of $f'$. After the fallback function is triggered, an internal transaction is sent which invokes function $f'$. Therefore, the rule `fb_in_call` indicates that the fallback function of account $c_{adv}$ is triggered by function $f$ in the contract of account $c$. The rule `ret_fb` indicates that the adversary gets a return message after the execution of $f'$ and sends a message denoting that the fallback function finishes executing.

## 4.3 An illustrative example

Fig. 5 shows a simplified version of a practical smart contract, which is with a vulnerability of transferMint [11]. Here, transferMint is a new type of vulnerability, which could be exploited by attackers to obtain unlimited tokens through manipulating the parameters of address type. We use the contract name `Ex1` to denote the address

```
contract Ex1{
    mapping(address=>uint) balances;
    constructor() public{
        balances[0x12] = 100;
    }
    function transfer (address to,uint value) public{
        uint val1 = balances[msg.sender] - value;
        uint val2 = balances[to] + value;
        balances[msg.sender] = val1;
        balances[to] = val2;
        return;
    }
}
```

**Figure 5: Illustrative example Ex1: a simplified version of a practical smart contract.**

of the account who owns this contract. The function `constructor` is used to initialize the global variable `balances` denoting the token balances of accounts. After initialization, the account of address $0x12$ has 100 tokens. The function `transfer(to,value)` can be called by an account to send a certain amount of tokens, *i.e.*, `value`, to the account of address `to`.

An execution of the model that corresponds to the attack is shown in Fig. 4. Since function `transfer` does not modify the ether balance of any account, we omit Evar fact in the figure. Hence, in the execution, the initial state is $\{\mathsf{Gvar}(\sigma_a(\mathtt{Ex1}), \sigma_a(100))\}$ where $\omega_0[1] = \sigma_a(\mathtt{Ex1})$ and $g(\omega_0)\backslash e(\omega_0) = [\![\sigma_a(100)]\!]$. Next, an external account invokes `transfer` whereby the rule `ext_call` is applied such that $\mathsf{Call_e}(\sigma_a(\mathtt{Ex1}), \sigma_a(\mathtt{transfer}), \sigma_v(c_b), \sigma_v(\mathtt{to}), \sigma_v(\mathtt{value}))$ is added to the new state. Since $\sigma_v(c_b)$, $\sigma_v(\mathtt{to})$ and $\sigma_v(\mathtt{value})$ can be arbitrary values, in this execution, they can be instantiated as $\sigma_a(0x12)$, $\sigma_a(0x12)$ and $\sigma_c(100)$ respectively. In the following steps, the state is updated in similar ways. When the transaction invoking `transfer` finishes, the state is $\mathsf{Gvar}(\sigma_a(\mathtt{Ex1}), \sigma_a(200))$, which implies that $\mathtt{balances}[0x12]$ changes into 200 in an unexpected way. Note that the numerical instantiation cannot be supported by original Tamarin prover. Moreover, the independent model cannot be verified directly to find an attack as shown in Fig. 4, since several technical challenges need to be addressed.

## 4.4 Technical challenges and main solutions

Since the module of independent modeling only provides a framework that automatically generates models of smart contracts partially, we have to address the following technical challenges to complement the model for the downstream verification.

**Challenge 1: recognizing security requirements.** Given an execution shown in Fig. 4, a corresponding property is still needed for the verifier to recognize this execution as an instance of some vulnerabilities. However, smart contracts are designed for different purposes and without a uniform standard for security requirements in practical scenarios, which makes the precise generation of security properties difficult. The existing automation-oriented analyzers summarized security patterns according to known vulnerabilities [65] [41]. However, the vulnerabilities that can be detected by these analyzers are limited to known ones, and just a variant of a known vulnerability may evade their detection [53]. There are also expressivity-oriented tools that require manually-defined specifications of contracts as input [50] [59], which cannot be applied to verification for a large number of smart contracts.

To deal with this challenge, we perform statistical analysis on 30577 real-world smart contracts and obtain a key observation: most of the smart contracts (91.11%) are finance-related, *i.e.*, the executions of these contracts may change the cryptocurrencies of themselves and others. Meanwhile, the attacks [33] that occurred on Ethereum also indicate that one of the main goals of the adversaries is to obtain cryptocurrencies. For these reasons, we divide the smart contracts into different categories according to the cryptocurrencies that they use and propose security properties to check whether the cryptocurrencies may be lost in unexpected ways.

**Challenge 2: contract-oriented automated reasoning.** Given an independent model, the rule `ext_call` can be applied repeatedly, which is corresponding to the practical scenarios that a function in contracts can be invoked any times. This may lead to non-termination of verification. Besides, the independent model may also be insufficient for verifying properties that require comparison between normal executions and abnormal ones, e.g., [31].

We address the challenge based on the fact that a transaction is atomic and cannot be interfered by other transactions. Therefore, the independent model can be reduced for different types of properties: (1) the properties that should be maintained for a single transaction; (2) the properties that may be affected by other transactions. For the first type, we propose to automatically generate invariant properties and the corresponding reduced model that the behaviors of other transactions are ignored. For the second type, since a transaction is atomic, the rest way to trigger an attack is to leverage different results of a sequence of transactions caused by different orders of the transactions or different block variables. Hence, we propose the equivalence properties and also the modeling method to achieve effective automated reasoning. We also modify the source code of Tamarin prover for supporting the verification where numerical constraint solving is additionally required.

# 5 PROPERTY GENERATION

To address Challenge 1, we divide finance-related smart contracts into three categories according to the type of cryptocurrencies they use: ether-related, token-related and indirect-related. An ether-related contract may transfer or receive ethers, *i.e.*, the official cryptocurrency of Ethereum, in a transaction. Similarly, a token-related contract may send or receive tokens, *i.e.*, the cryptocurrency implemented by the contract itself. An indirect-related contract is used in the former two contracts to provide additional functionality, *e.g.*, permission management. Hence, to check whether the cryptocurrencies may be lost in unexpected ways, we focus on generating security properties for ether-related and token-related contracts. We propose to recognize the category and key variables related to cryptocurrencies from the codes of smart contracts, and use the information to generate the security properties. Note that we analyze the indirect-related contracts in an indirect way and thus do not generate properties for the indirect-related contracts. For example, given a token contract $C_1$ and an indirect-related contract $C_2$, assume that $C_1$ implements authentication by invoking functions in $C_2$. In this case, we can specify $C_1$ to be analyzed and then generate a model for it, which takes into account the interaction between $C_1$ and $C_2$.

## 5.1 Recognizing categories and key variables

**Ether-related contracts.** The ethers can be transferred by using official functions, *e.g.*, `transfer`, `send` and `call`. The modifier `payable` is only used in ether-related contracts for receiving ethers. Therefore, FASVERIF recognize an ether-related contract by determining if there are keywords, *i.e.*, `transfer`, `send`, `call` or `payable` in the contract. If a contract is recognized as an ether-related contract, then we use the built-in variable `balance` as the key variable, which denotes the ether balance of an account.

**Token-related contracts.** The token-related contracts can be divided into token contracts and token managing contracts. A token contract is used to implement a kind of customized cryptocurrency, *i.e.*, tokens, which can be traded and have financial value. A token managing contract, *e.g*, an ICO contract [8], is used to manage the distribution or sale of tokens.

We propose a method to recognize token contracts based on another observation from our statistical result in Section 7.2: developers tend to use similar variable names to represent the token balance of an account. Therefore, a contract is identified as a token contract, if there is a variable of type `mapping(address=>uint)` with a name similar to two commonly used names: `balances` or `ownedTokenCount`. Specifically, we calculate the similarity of names using fuzzywuzzy[12], a Python package that uses Levenshtein Distance to calculate the similarity between strings. When the similarity calculated is larger than 85, we consider two names similar. The threshold 85 is set based on our evaluation on real-world contracts, which will be introduced in Section 7.2. For the contracts using uncommon variable names for token balances, FASVERIF also supports the users to provide their own variable names. In addition to `balances`, we observe that some token-related contracts define a variable of `uint` type to record the total number of tokens. Similarly, we use the most common variable name `totalSupply` to match the variables representing the total amount of tokens. However, this kind of variables are not used to recognize the token contract, but rather for the subsequent generation of properties.

After the recognition of token contracts, we search for contracts that instantiate and deploy token contracts and regard them as token managing contracts.

```
function f()
{
    require(!transfered);
    msg.sender.call.value(10);
    transfered = True;
}
```

**Figure 6: An example with reentrancy vulnerability.**

## 5.2 Generating security properties with reduction and restrictions

As mentioned in Section 4.4, we propose two kinds of properties: invariant properties and equivalence properties.

**Invariant properties.** The invariant property requires that for any transaction a proposition (a statement that denotes the relationship between values of variables) $\phi$ holds when the transaction finishes, if $\phi$ holds when the transaction starts executing. Since a transaction is atomic, FASVERIF checks invariant properties in single transactions instead of the total executions to achieve effective automated reasoning. Here, we design the invariants to ensure that the token balances in token-related contracts are calculated in an expected way. Note that we do not design invariant properties for ether-related contracts, since the calculation of the ether balance is performed by the EVM and there are conditional checks throughout the process to ensure correctness [26].

For a token contract with key variable balances, the following invariant is generated:

$$\sum_{a \in A_1} \text{balances}(a) = C_1 \qquad \texttt{(token\_inv1)}$$

Since a transaction can only affect a limited number of accounts, $A_1$ is the set of addresses of the accounts whose token balances may be modified in the transaction. $C_1$ is an arbitrary constant value and the invariant implies that the sum of token balances of all accounts should be unchanged after a transaction. If the invariant is broken, it indicates an error in the process of recording token balances, which would make this kind of tokens worthless [10]. Here, balances can be replaced by any variable name denoting the token balances. If there are multiple variables denoting token balances of different types, all of them will be used. For instance, given two variable names balances denoting the balance of tokens that can be used and freezebalances denoting the balance of tokens that cannot be withdrawn, respectively, the following invariant is generated:

$$\sum_{a \in A_1} (\text{balances}(a) + \text{freezebalances}(a)) = C_1 \quad \texttt{(token\_inv2)}$$

The invariant indicates that the sum of normal tokens and frozen tokens of all accounts is unchanged. Specially, if there is a key variable totalSupply denoting the total amount of tokens in the token contract, the constant $C_1$ in `token_inv1` and `token_inv2` will be replaced by totalSupply. For a token managing contract, the invariant `token_inv1` or `token_inv2` is generated for the token contract that it manages. FASVERIF also supports the users to provide customized invariants to check the security of contracts.

**Equivalence properties.** We define the equivalence property as follows: The equivalence of a global variable $v$ holds for a transaction sequence $T$, if the value of $v$ after $T$'s execution is always the same. Here we study the equivalence of the token or ether balance of the adversary. Given two sequences $T_A$ and $T_B$ that have the same transactions, we propose the following property:

$$\begin{aligned} &\text{balances}_A(c_{adv}) = \text{balances}_B(c_{adv}) \\ &\qquad \& \ \text{balance}_A(c_{adv}) = \text{balance}_B(c_{adv}) \quad \texttt{(equivalence)} \end{aligned}$$

Here, denote $\text{balances}_A(c_{adv})$ and $\text{balance}_A(c_{adv})$ as the token balance and ether balance of the adversary after execution of $T_A$, respectively. Similarly, $\text{balances}_B(c_{adv})$ and $\text{balance}_B(c_{adv})$ represent the corresponding balances for $T_B$. `equivalence` requires that the adversary cannot change its own balances by changing the orders of transactions or other conditions; otherwise, the difference of the balances may be the illegal profit of the adversary.

## 5.3 Relationship between properties and common vulnerabilities

The properties of FASVERIF are designed with a basic idea: leveraging the phenomenon that the loss of ethers and tokens is one of the popular intentions of attackers [33]. This idea is also shared by another existing work [30] that detects economic attacks, *e.g.*, arbitrage. FASVERIF generates properties based on key variables that denote the token balances or ether balances. Our properties aim to cover vulnerabilities that cause financial losses. As a result, FASVERIF covers 6 types of vulnerabilities, including *transferMint* that is not supported by existing automatic tools, through two properties.

To explain the usage of our properties, we provide examples of several common vulnerabilities, detailing how contracts with these vulnerabilities violate the two properties mentioned above.

**Gasless send.** During the executions of official functions `send` and `call`, if the gas is not enough, the transaction will not be reverted but a result returns. If a contract does not check the execution results of `send` or `call`, it may mistakenly assume that the execution was successful. Given two sequences of same transactions that invoke functions with gasless send vulnerability, one with sufficient gas and one without sufficient gas, the results of them will be different. Therefore, the equivalence property is violated.

**Reentrancy.** Taking the contract in Fig. 6 as an example, suppose that the adversary sends a transaction to invoke `f` and the statement on line 4 sends ethers to the adversary. According to Section 3.2, the adversary can then send an internal transaction through the fallback function to call `f` again, and since the code on line 5 is not executed, the checking on line 3 will still pass, allowing the adversary to get ethers one more time. Assume that there are two sequences $T_1$ and $T_2$, $T_1$ consisting of two transactions invoking `f` and $T_2$ consisting of one transaction invoking `f` and one internal transaction invoking `f` through a reentrancy vulnerability. We treat the internal transaction sent by the adversary as a transaction and consider $T_1$ and $T_2$ as consisting of the same transactions. In this case, the ether balances of adversary after $T_1$ and $T_2$ are different, which means that the equivalence property is broken.

**TD&TOD.** When some statements are control dependent on the `block.timestamp`, the adversary can control the execution of these statements by modifying `block.timestamp` in a range, which is called TD. Given two same sequences of transactions that invoke a function with TD, the execution results may be different with different `block.timestamp`, which violates the equivalence property. Similarly, the contracts with TOD vulnerability violate our equivalence property when the order of transactions changes.

**Overflow/underflow.** Overflow/underflow is a kind of arithmetic error. Since the goal of FASVERIF is to analyze the financial security of contracts, FASVERIF detects overflow/underflow vulnerabilities that can change the number of tokens. For the remaining overflow/underflow vulnerabilities, FASVERIF can also support them through custom invariants as mentioned in Section 5.2.

# 6 COMPLEMENTARY MODELING AND VERIFICATION

In this section, we introduce how we address Challenge 2 using our contracts modeler and verifier. According to different properties of a contract, we propose the method of complementary modeling to automatically generate customized models. The customized models are built upon the independent model with rules replaced or new rules added. Furthermore, we propose a stepwise solution to check whether a customized model satisfies the corresponding property.

## 6.1 Complementary modeling

The goal of complementary modeling is to generate a customized model, which satisfies that a property is not valid, if and only if there exists an execution in the model that breaks the property. Besides, to support automated verification, the model is added with more constraints such that each execution that reaches a certain state breaks the property. Then, the property is not valid if and only if the state is reachable. Hence, we design the method for invariant property and equivalence property as follows.

**Invariant properties.** The generated model for invariant properties has the following features:

i) The invariant holds at the beginning of any execution.
ii) An execution simulates the execution of one transaction.
iii) The invariant is assumed to be broken at the end of any execution, which corresponds to the state that breaks the property.

To make the generated model conform to feature i), we first replace the rule `init_gvars` with the following rule:

$$\begin{aligned}&[\mathsf{FR}(g(\omega_0)\backslash e(\omega_0))] - [\mathsf{Init_G}(\omega_0[1]), \theta_e(\phi)] \rightarrow\\ &\qquad [\mathsf{Gvar}(\llbracket \omega_0[1] \rrbracket \cdot g(\omega_0)\backslash e(\omega_0))] \qquad (\texttt{init\_gvars\_inv})\end{aligned}$$

Here, $\phi$ can be any invariant in Section 5.2. We also use the function $\theta_e$ defined in Section 4.2 to transform $\phi$ into terms in rules. The fact $\theta_e(\phi)$ denotes that the invariant $\phi$ holds after the initialization.

Then we replace the rule `ext_call` with a new rule:

$$\begin{aligned}&[\mathsf{Fr}(\sigma_v(c_b)), \mathsf{FR}(\sigma(seq(\mathsf{d})))] - [\mathsf{Start}()] \rightarrow\\ &\qquad [\mathsf{Call_e}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b) \rrbracket \cdot \sigma(seq(\mathsf{d})))] \quad (\texttt{ext\_call\_inv})\end{aligned}$$

Different from rule `ext_call`, an action Start() is added into rule `ext_call_inv`. The action is associated with the restriction:

$$All\ \#i\ \#j.\mathsf{Start}()@i\ \&\ \mathsf{Start}()@j\ \ =>\ \ i = j$$

This restriction requires that Start() occurs only once in an execution of the model. The action and restriction achieve feature ii).

Finally, we modify the rule `ret_ext` into `ret_ext_inv`:

$$\begin{aligned}&[\mathsf{Var}_i(\sigma(\omega))] - [\mathsf{Pred\_eq}(\omega[3], \mathsf{EXT}), \theta_{ne}(\phi), \mathsf{End}()] \rightarrow\\ &\qquad [\mathsf{Gvar}(\llbracket \omega[4] \rrbracket \cdot g(\omega)\backslash e(\omega)), \mathsf{Evar}(e(\omega))] \quad (\texttt{ret\_ext\_inv})\end{aligned}$$

where `ret_ext_inv` has additional facts $\theta_{ne}(\phi)$ and End() compared to `ret_ext`, which together achieve feature iii). Fact End() serves as an indicator that an execution of the model reaches the end of the transaction if rule `ret_ext_inv` is applied, and fact $\theta_{ne}(\phi)$ means that invariant $\phi$ is broken at the same time.

**Equivalence properties.** The generated model for equivalence properties has the following features:

i) An execution of the model simulates the executions of two sequences $T_A$ and $T_B$ consisting of the same transactions but possibly with different orders.
ii) Before the executions of $T_A$ and $T_B$, the values of global variables and ether balances of all accounts are the same.
iii) The ether or token balances of the adversary are assumed to be different at the end of any execution, which corresponds to the state that breaks the equivalence property.

Firstly, we replace rules `init_evars` and `init_gvars` with the following rules to achieve feature ii):

$$[\mathsf{FR}(e(\omega_0))] - [\mathsf{Init}_{\mathsf{E}}()] \rightarrow [\mathsf{Evar}_{\mathsf{A}}(e(\omega_0)), \mathsf{Evar}_{\mathsf{B}}(e(\omega_0))] \tag{init\_evars\_AB}$$

$$[\mathsf{FR}(g(\omega_0)\backslash e(\omega_0))] - [\mathsf{Init}_{\mathsf{G}}(\omega_0[1])] \rightarrow [\mathsf{Gvar}_{\mathsf{A}}(\llbracket\omega_0[1]\rrbracket\cdot g(\omega_0)\backslash e(\omega_0)) \\ , \mathsf{Gvar}_{\mathsf{B}}(\llbracket\omega_0[1]\rrbracket \cdot g(\omega_0)\backslash e(\omega_0))] \tag{init\_gvars\_AB}$$

Similar to rule `init_evars_AB`, the $\mathsf{Call}_{\mathsf{e}}$ facts in `ext_call_AB` and `ext_call_bvar_AB` are duplicated into $\mathsf{Call}_{\mathsf{Ae}}$ and $\mathsf{Call}_{\mathsf{Be}}$, which indicates that two transactions with same parameters and same sender are sent. The $\mathsf{Bvar}(bt)$ fact in `ext_call_bvar_AB` is also duplicated into $\mathsf{Bvar}_{\mathsf{A}}(bt_A)$ and $\mathsf{Bvar}_{\mathsf{B}}(bt_B)$, which represents the timestamp of $T_A$ and $T_B$, respectively. This rule indicates that the adversary can modify the timestamp of blocks containing transactions in $T_A$ and $T_B$, which corresponds to C3 in Section 3.2. Except rules `init_evars`, `init_gvars`, `ext_call` and `ext_call_bvar`, each of the remaining rules in the model is replicated into two rules, and the facts of the two rules are added with different subscripts $A$ and $B$ to represent the execution of transactions in sequences $T_A$ and $T_B$, respectively. For example, the rule `recv_ext` is replaced with `recv_ext_A` and `recv_ext_B`. The complete form of the above rules can be found in Appendix A.2. There are two actions added in `recv_ext_A` and `recv_ext_B` respectively: $\mathsf{Exc}_{\mathsf{A}}(\sigma_v(c_b)), \sigma_a(f))$, $\mathsf{Exc}_{\mathsf{B}}(\sigma_v(c_b)), \sigma_a(f))$. These actions are associated with the following restrictions to achieve feature i):

$$All\ \#i\ c_b.\mathsf{Exc}_{\mathsf{A}}(\sigma_v(c_b)), \sigma_a(f))@i => Ex\ \#j.\mathsf{Exc}_{\mathsf{B}}(\sigma_v(c_b)), \sigma_a(f))@j$$
$$All\ \#i\ c_b.\mathsf{Exc}_{\mathsf{B}}(\sigma_v(c_b)), \sigma_a(f))@i => Ex\ \#j.\mathsf{Exc}_{\mathsf{A}}(\sigma_v(c_b)), \sigma_a(f))@j$$

Finally, to achieve feature iii), we add the following rule to compare the ether balances and token balances of the adversary:

$$[\mathsf{Gvar}_{\mathsf{A}}(\llbracket\omega[4]\rrbracket\cdot g(\omega)\backslash e(\omega)), \mathsf{Evar}_{\mathsf{A}}(e(\omega)), \mathsf{Gvar}_{\mathsf{B}}(\llbracket\omega[4]\rrbracket\cdot g(\omega)\backslash e(\omega)), \\ \mathsf{Evar}_{\mathsf{B}}(e(\omega))] - [\theta_{ne}(\phi_{equ}), \mathsf{End}()] \rightarrow []$$

Facts $\theta_{ne}(\phi_{equ})$ and $\mathsf{End}()$ are added for subsequent verification where $\phi_{equ}$ is property `equivalence` in Section 5.2.

## 6.2 Verification

The verification module is implemented by modifying the source code of Tamarin prover [48] to achieve modeling using multiset rewriting rules with additional support for numerical constraint solving by Z3 [27]. Taken a generated property and the corresponding model as input, the workflow of this module is as follows: 1) Search for an execution that reaches the fact $\mathsf{End}()$ without considering the numerical constraints. 2) If the search fails, the module terminates and outputs that the property is valid; otherwise, go to step 3). 3) Collect the numerical constraints that the execution must satisfy, and solve the constraints by Z3. 4) If the set of constraints is satisfied according to Z3, which indicates that the execution that violates the property exists, the module terminates and outputs the execution as a counterexample; otherwise, add a constraint to the model that the execution does not exist, and go to step 1).

In step 3), FASVERIF additionally generates numerical constraints related to timestamps of blocks, which cannot be modeled in multiset rewriting system. Specifically, according to adversary model C3, the adversary can only increase the timestamp by a maximum of 15 seconds. Moreover, the timestamp of a block must be greater than the timestamp of previous ones, according to the validation algorithm of blocks [24]. Thus, if a contract uses `block.timestamp`, when verifying the equivalence property, FASVERIF processes $\mathsf{Bvar}_{\mathsf{A}}(bt_A)$ and $\mathsf{Bvar}_{\mathsf{B}}(bt_B)$ in the execution with the following additional steps: i) assign indices to each $bt_A$ and $bt_B$ in the order they appear in the execution, starting with 0; ii) generate numerical constraint $bt_{Ai} > bt_{Ai-1}$ and $bt_{Bi} > bt_{Bi-1}$ for every index i except for 0. ii) generate numerical constraint $bt_{Bi} <= bt_{Ai} + 15$ for $bt_{Ai}$ and $bt_{Bi}$ with same index $i$. Note that FASVERIF can also handle cases that the adversary increases block.timestamp by up to more than or less than 15 seconds by modifying this constraint.

To support processing of numerical constraints, the modified Tamarin supports parsing of the conditional facts $\mathsf{EqNum}$, $\mathsf{NeNum}$, $\mathsf{LessNum}$ and operators $\Diamond \in \{+, -, *, /, \%, **\}$. The conditional facts are transformed into expressions which can be computed by Z3. Specifically, $\mathsf{EqNum}$ facts are transformed into equalities, and $\mathsf{NeNum}$ and $\mathsf{LessNum}$ are transformed into inequalities.

## 6.3 Formal guarantee

Currently there is no official formal semantics of Solidity to the best of our knowledge. Instead, we prove the correctness of translation from Solidity language to our models based a custom semantics of Solidity, named KSolidity [42], which is claimed to completely cover the supported high-level core language features specified by the official Solidity documentation and be consistent with the official Solidity compiler. Note that the proofs can improve the faithfulness of FASVERIF, but it still does not mean that the results of FASVERIF are completely reliable, due to the informal parts of FASVERIF and the gap between Solidity and EVM bytecode. Specifically, for a contract that can be modeled by FASVERIF, *e.g.*, with bounded loops, we prove the following theorem (informal description):

Theorem 6.1. *(Soundness). If an invariant property (or equivalence property) holds in FASVERIF, it holds in real-world transactions interpreted by KSolidity semantics.*

Proof. See Appendix A.3. □

# 7 EVALUATION

In this section, we firstly make preparations on the experimental setup, including the types of vulnerabilities, datasets, and representative tools that we choose. Then, we report the experimental results and analyze the effectiveness of FASVERIF on datasets of existing works. Finally, we verify real-world contracts using FASVERIF and demonstrate the zero-day bugs that FASVERIF finds.

## 7.1 Experimental setup

**Types of Vulnerabilities.** First, we introduce the vulnerabilities that FASVERIF currently targets. According to SWC Registry [14], a library that consists of categories and test cases of smart contracts' vulnerabilities, there are 37 types of smart contracts' vulnerabilities. These vulnerabilities can be divided into three categories: a) vulnerabilities that can be detected through syntax checking, *e.g.*, outdated compiler version. b) vulnerabilities that do not have clear consequences, *e.g.*, dangerous delegatecall. c) vulnerabilities that can actually cause losses of ethers or tokens. FASVERIF does not detect

**Table 1: A comparison of representative automated analyzers for smart contracts. (Acc and F1 outside brackets correpsond to the finance-vulnerable contracts, while those inside brackets correpsond to the vulnerable contracts)**

| Types of | Osiris | | SECURIFY | | Mythril | | OYENTE | | VERISMART | | SmartCheck | | Slither | | eThor* | | FASVERIF * | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Vulnerabilities | Acc(%) | F1 | Acc(%) | F1 | Acc(%) | F1 | Acc(%) | F1 | Acc(%) | F1 | Acc(%) | F1 | Acc(%) | F1 | Acc(%) | F1 | Acc(%) | F1 | U |
| TOD-eth | / | / | 96.43 | 0.98 | / | / | 42.86 | 0.6 | / | / | / | / | / | / | / | / | 100 | 1 | 10 |
| TOD-token | / | / | / | / | / | / | / | / | / | / | / | / | / | / | / | / | 100 | 1 | 0 |
| TD | 71.60<br>(70.37) | 0.83<br>(0.82) | / | / | 45.68<br>(44.44) | 0.62<br>(0.62) | 76.54<br>(75.31) | 0.87<br>(0.86) | / | / | / | / | 16.05<br>(14.81) | 0.26<br>(0.25) | / | / | 95.06<br>(93.83) | 0.97<br>(0.96) | 33 |
| reentrancy | 66.67<br>(69.05) | 0.79<br>(0.81) | 78.57<br>(76.19) | 0.85<br>(0.84) | 71.42<br>(69.04) | 0.81<br>(0.8) | 73.81<br>(76.19) | 0.85<br>(0.86) | / | / | 73.81<br>(76.19) | 0.85<br>(0.86) | 85.71<br>(83.33) | 0.91<br>(0.90) | 83.72<br>(86.05) | 0.92<br>(0.93) | 90.48<br>(88.10) | 0.94<br>(0.93) | 2 |
| gasless send | / | / | 92.19 | 0.95 | 82.35 | 0.67 | / | / | / | / | 92.19 | 0.95 | 85.94 | 0.91 | / | / | 100 | 1 | 7 |
| overflow/underflow | 81.20<br>(81.20) | 0.89<br>(0.89) | / | / | 95.30<br>(95.30) | 0.97<br>(0.97) | 90.27<br>(90.27) | 0.95<br>(0.95) | 98.99<br>(98.99) | 0.99<br>(0.99) | / | / | / | / | / | / | 99.33<br>(99.33) | 0.99<br>(0.99) | 4 |
| transferMint | / | / | / | / | / | / | / | / | / | / | / | / | / | / | / | / | 100 | 1 | 0 |

the vulnerabilities in category a) and b) although other automatic tools can cover some of them, since they do not affect the financial security of contracts. Instead, FASVERIF targets the vulnerabilities in category c) as they can cause financial loss and are difficult to detect. There are 6 types of vulnerabilities that FASVERIF currently supports: 1) *transaction order dependency (TOD)*; 2) *timestamp dependency (TD)*; 3) *reentrancy*; 4) *gasless send*; 5) *overflow/underflow*; 6) *transferMint* [11]. The relationship between these vulnerabilities and our properties has been mentioned in Section 5.3. We divide the *TOD* vulnerabilities into two groups: *TOD* that changes ether balances of accounts (*TOD-eth*), *TOD* that changes token balances of accounts (*TOD-token*), since SECURIFY and OYENTE only support the detection of the former.

**Datasets.** We use two datasets of smart contracts to evaluate FASVERIF, which are uploaded as supplementary material. The first dataset, called *vulnerability dataset*, is used to test the performance of FASVERIF in detecting different types of vulnerabilities compared with other automated tools. We collect 610 smart contracts with vulnerabilities in category c) mentioned above from public dataset of other works [41][58][45][37]. We filter out 6 smart contracts whose codes are incomplete and 56 smart contracts that FASVERIF does not support. We illustrate the number of contracts that FASVERIF cannot support by types of vulnerabilities in the last column of Table 1, and the reasons that FASVERIF does not support them will be introduced in Section 9.1. Finally we get *vulnerability dataset* with 548 contracts. The second dataset, called *real-world dataset*, is used to evaluate the effectiveness of FASVERIF in detecting real-world smart contracts. We use a crawler to crawl 46453 Solidity source code files from Etherscan [23], and then filter the contracts to remove duplicates. We calculate the similarity of two files using difflib [21] package of Python, and considered two contracts as duplicates when their similarity is larger than 90%. Finally, we obtained 17648 Solidity files containing 30577 contracts as *real-world dataset*. We also add 11 smart contracts with vulnerability of transferMint from the *real-world dataset* to the *vulnerability dataset*, since the previous datasets have not gathered this type of contracts.

**Tools.** We compare FASVERIF with the following representative automatic tools using different technologies: OYENTE [47], Mythril[35], SECURIFY[65], ContractFuzzer [41], Osiris [63], Slither [38], SmartCheck [60], VERISMART [58] and eThor [56]. OYENTE and Mythril are based on symbolic execution, where Mythril is a tool recommended by Ethereum. Osiris is a tool that extends OYENTE to detect overflow. Slither is a framework converting smart contracts into IR, which can also detect vulnerabilities. SmartCheck is a static analysis tool that checks contracts against XPath patterns. SECURIFY is an analyzer searching for specific patterns in bytecode of contracts. Here we use its newest version called SECURIFY2. VERISMART verifies arithmetic safety of smart contracts by SMT solving. ContractFuzzer uses fuzzing testing to find vulnerabilities in smart contracts. eThor is a sound static analyzer that abstracts the semantics of EVM bytecode into Horn clauses. eThor and FASVERIF are hybrid-oriented and we indicate this by asterisks in TABLE 1.

We do not compare FASVERIF with ZEUS [43], another hybrid-oriented tool, since it is not publicly available. Besides, We do not compare FASVERIF with expressivity-oriented tools for the following reasons: **1)** The main obstacle to the comparison is that all existing expressivity-oriented tools require manual input of properties (for each vulnerability) of smart contracts, which require certain expertise and would be labor-intensive when evaluating hundreds of contracts in *vulnerability dataset*. Meanwhile, how to express the properties in different specification languages equivalently becomes a problem and may affect the fairness of comparison. **2)** We also try to use our method to generate properties for expressivity-oriented tools that can verify properties automatically. To the best of our knowledge, SmartPulse is the only expressivity-oriented tool that can verify properties automatically and publicly available [17]. However, we fail to make SmartPulse work by following the instructions on its page.

**Experimental Environment.** We experiment on a server with Intel Xeon Gold 5215 2.50GHz CPU, 128G memory and Ubuntu 16.04 (64-bit). FASVERIF is implemented based on Tamarin 1.6.0.

## 7.2 Statistical analysis

We first perform statistical analysis on *real-world dataset*. We manually classify the contract as finance-related or others taking the following parts of contracts into account and try our best to avoid misclassification: 1) Contract names. The usage of some contracts can be shown in their name, e.g, PoolToken, so the contract names can be information that helps us to determine the types of contracts. 2) Contract annotations. The annotations of contracts can provide us with some information like the contracts' usage, the related contracts and so on. 3) Inheritance of contracts. When a contract is identified as a token contract, its children contracts can possibly be a token contract. 4) Contract instantiation statements. When a contract instantiates a token contract, it can possibly be a token managing contract. 5) Ether transfer statements. When a contract uses official functions to transfer ethers, it is an ether-related contract. Note that we determine the types of contracts according

**Table 2: The effectiveness of our method for identifying token contracts.**

| threshold | 70 | 75 | 80 | 85 | 90 |
|---|---|---|---|---|---|
| Acc(%) | 98.31 | 98.32 | 98.32 | 98.50 | 98.46 |
| F1(%) | 98.13 | 98.14 | 98.14 | 98.31 | 98.27 |

to their codes and annotations, and contracts that are difficult to distinguish their usage from both parts are classified as others.

After the above classification, we find 27858 finance-related contracts, including 6307 ether-related contracts (20.63%), 7661 token-related contracts (25.05%), 5994 contracts both ether-related and token-related (19.60%) and 7896 indirect-related contracts in total (25.82%). The remaining contracts account for 8.89%. Hence, finance-related contracts make up a major portion (91.11%) of the real-world contracts, which validates the goal of generating properties aiming to protect cryptocurrencies shown in Section 5.2.

During the classification, we find that since the official ERC20 [15] token standard of Ethereum recommends using variable name `balances` to represent token balances, most token contracts use names similar to `balances` to denote token balances. Besides, there are also token contracts that use names similar to `ownedTokenCount` to represent token balances due to ERC721 [22] standard.

To validate our observation and evaluate the effectiveness of our methods to identify token contracts, we perform an evaluation on *real-world dataset.* We search for contracts with variables of type `mapping(address=>uint)` that have names similar to `balances` or `ownedTokenCount`, while the similarity of two names is calculated based on fuzzywuzzy[12], and the thresholds are set to 70, 75, 80, 85 and 90, respectively. We collect the following data under different thresholds: 1) TP: the number of token contracts correctly identified. 2) FN: the number of token contracts that are missed. 3) FP: the number of contracts misclassified as token contracts. 4) TN: the number of contracts that are not token contracts correctly classified. 5) Accuracy: $Acc = \frac{TP+TN}{TP+TN+FP+FN}$. 6) F1: $F1 = \frac{2TP}{2TP+FP+FN}$. We only show 5) and 6) in Table 2 due to the page limit. According to Table 2, our method for identifying token contracts achieves Accuracy and F1-score higher than 98% under different thresholds. Since the effectiveness of our method reaches its highest value at a threshold of 85, we choose 85 as our final threshold.

## 7.3 Comparison

Unlike other automatic tools, FASVERIF detects the effect of the vulnerabilities, *i.e.*, whether causing the financial loss. To fairly compare FASVERIF and other automatic tools, we run the tools on *vulnerability dataset*, and collect two sets of results as shown in TABLE 1. Here we call a contract with a vulnerability as a vulnerable contract, and call a contract with a vulnerability that can cause financial loss as a finance-vulnerable contract. We regard the number of contracts that the tool correctly recognizes as a finance-vulnerable / vulnerable contract as $TP$, and regard the number of contracts that the tool correctly recognizes as a contract that is not finance-vulnerable / vulnerable as $TN$. The calculation formulas of accuracy and F1 are the same as the ones mentioned above. Due to the page limit, we only show the accuracy and F1 of tools in TABLE 1. Note that the vulnerabilities of *TOD-eth*, *TOD-token*, *gasless send* and *transferMint* always cause financial loss, thus the two sets of results for them are the same, and we only show one set of results.

Totally, FASVERIF outperforms the representative tools that it achieves higher accuracy and F1 values in the detection of vulnerable and finance-vulnerable contracts in *vulnerability dataset*. Meanwhile, FASVERIF is the only one that is able to detect all the types of vulnerabilities in TABLE 1 among the automated tools mentioned above. Note that we try to run ContractFuzzer but fail to make this tool report any findings. Though being in contact with the authors, we are unable to fix the issue and both sides eventually give up. SECURIFY can output alerts at a coarse-grained level for all contracts using timestamp, but is not targeted to detect *TD*, so we do not compare its ability to detect *TD* with FASVERIF.

We analyze the reason for the false results produced by different automatic tools shown in Table 1.

***TOD-eth,TD, gasless send***: The above automatic tools detect these types of vulnerabilities based on their pre-defined patterns and their accuracy depends on the patterns. On one hand, progressive patterns can result in false negatives. For example, SECURIFY decides if a contract is secure against *gasless send* by matching the pattern whether each return value of `send` is checked. However, a contract in the dataset checks the result of `send` but does not handle the exception raised by `send`, which evades the detection of SECURIFY. On the other hand, conservative patterns can lead to false positives. For example, OYENTE and SECURIFY detect *TOD-eth* according to the pattern that when the order of transactions changes, the recipient of ethers may also change. A contract returns ethers to their senders and the first sender will be the first receiver. For this case, both OYENTE and SECURIFY falsely report *TOD-eth* vulnerability. However, all senders will eventually receive ethers, *i.e.*, the result is not changed whenever the order of transactions changes, whereas our equivalence property holds. Additionally, the tools using symbolic execution, *e.g.*, OYENTE and Mythril, may produce false negatives since they only explore a subset of contracts' behaviors.

***reentrancy***: EThor defines a property: an internal transaction can only be initiated by the execution of a `call` instruction, which over-approximates the property that a contract free from *reentrancy* should satisfy. Therefore, eThor gets more false positives than FASVERIF in detection of *reentrancy*. The reasons for the false reports of the other tools in the detection of *reentrancy* are still inaccurate patterns.

***Overflow/underflow***: OYENTE, Mythril, Osiris assume that the values of all the variables are arbitrary and thus output false positives for this category. Differently, FASVERIF and VERISMART consider additional constraints of variables, e.g., for the variables whose values cannot be changed after initialization, their values should be equal to the initial values. VERISMART outputs 2 false positives due to its assumption: every function can be accessed.

FASVERIF also produces 9 false negatives due to the error of property generation. Specifically, FASVERIF fails to detect 2 contracts with vulnerabilities of *overflow*. In these two contracts, the variable `allowance` may overflow. We currently do not design the invariants for this variable. Instead, we manually define a new invariant according to the codes of these contracts. Eventually, FASVERIF successfully discovers the vulnerabilities. FASVERIF also misses 3 contracts with vulnerabilities of *TD* and misses 4 contracts with vulnerabilities of *reentrancy*. These contracts use uncommon variable names to denote the number of tokens. We manually specify the key variable names in the verification of equivalence properties and finally find out the missed vulnerabilities.

To compare the efficiency of the above tools, we calculate the average time taken by them to analyze one contract in *vulnerability dataset* as follows: Slither (2.16 s), SmartCheck (4.93 s), eThor (11.95 s), OYENTE (20.81 s), Mythril (55.00 s), VERISMART (63.45 s), Osiris (73.52 s), SECURIFY (222.99 s), FASVERIF (829.61 s). As the results show, the accuracy of FASVERIF comes at the expense of efficiency, and we leave the optimization of efficiency in future works.

## 7.4 Security analysis of real-world smart contracts

To evaluate the effectiveness of FASVERIF in real-world contracts, we conduct an experiment on randomly-selected 1413 contracts from *real-world dataset*. FASVERIF reports 14 contracts with vulnerabilities, of which 11 violates the invariant property and 3 violates the equivalence property. We manually check the results and find that 13 of detected contracts are with true zero-day vulnerabilities, which can still be leveraged by adversaries online, while the rest one has been destroyed. Among the zero-day bugs, there are 10 of *transferMint* vulnerabilities, which cannot be detected by existing automatic tools as shown in Table 1. To avoid abusing the vulnerabilities, we do not provide the addresses of vulnerable smart contracts. Instead, we choose the destroyed contract and another typical vulnerable contract as examples, and illustrate the vulnerabilities according to their simplified versions as follows:

**Example 1.** The contract Ex1 shown in Fig. 5 is a contract with a zero-day bug. Note that the contract shown in Fig. 5 is simplified, and in the practical contract there are conditional statements to ensure that every adding or subtracting operation will not cause *overflow/underflow*. FASVERIF recognizes Ex1 as a token-related contract and chooses the invariant token_inv1. Specifically, assume that the sum of token balances of to and msg.sender before the transaction, *i.e.*, $\texttt{balances}_0\texttt{[to]+balances}_0\texttt{[msg.sender]}$ is value $C_1$, FASVERIF checks whether the sum after the transaction, i.e., $\texttt{balances}_1\texttt{[to]} + \texttt{balances}_1\texttt{[msg.sender]}$ can be different from $C_1$. In the verification, FASVERIF successfully finds an execution that reaches End() and has a constraint Pred_eq(to, msg.sender) in the modified Tamarin prover. According to the constraint, the modified Tamarin unifies the names by replacing msg.sender in all expressions with to. Moreover, since the rules var_declare and var_assign are in the execution, $\texttt{balances}_1\texttt{[to]}$ is replaced by $\texttt{balances}_1\texttt{[to]} + \texttt{value}$. Hence, the constraints $\texttt{balances}_0\texttt{[to]} + \texttt{balances}_0\texttt{[to]} = C_1$, $\texttt{balances}_0\texttt{[to]} + \texttt{value} + \texttt{balances}_0\texttt{[to]} + \texttt{value} \neq C_1$ are added to Z3. As a result, The constraints are satisfied with $\texttt{value} \neq 0$, which indicates the invariant token_inv1 is broken and FASVERIF decides this contract as a vulnerable case. Comparatively, SECURIFY, OYENTE and Mythril, which are implemented based on pattern matching, fail to recognize this new type vulnerability with an unknown pattern. VERISMART also fails to detect this vulnerability because it only analyze arithmetic safety and this vulnerability does not cause *overflow/underflow*.

```
contract Ex2{
    mapping(address=>uint) balances;
    function dice(uint bet) public{
        uint prize = bet * 9 / 10;
        if (block.timestamp % 2 == 1){
          balances[msg.sender] = balances[msg.sender]+prize;
        }
        if (block.timestamp % 2 == 0){
          balances[msg.sender] = balances[msg.sender]-bet;
        }
    }
}
```

**Figure 7: Example Ex2: a simplified version of a vulnerable smart contract.**

**Example 2.** Fig. 7 shows a simplified version of a practical contract that violates the equivalence property. The function dice is used to play a game. If block.timestamp is odd, msg.sender will get a prize, *i.e.*, his token balances will be increased; otherwise his token balances will be decreased. The contract is recognized as a token contract since it defines a variable balances of mapping(address=>uint) type. An invariant property and an equivalence property are both generated. In verification of equivalence property, FASVERIF can find an execution that simulates two sequences $T_A$ and $T_B$ consisting of a same transaction which invokes dice. Correspondingly, since this two sequences only use block.timestamp once, the terms $bt_A$ and $bt_B$ are transformed into $bt_{A0}$ and $bt_{B0}$ and a numerical constraint $bt_{B0} <= bt_{A0} + 15$ is generated as mentioned in Section 6.2. The token balances after $T_A$ and $T_B$ are obviously different as $bt_A$ and $bt_B$ could be in different parity, which violates the equivalence property. Therefore, Ex2 is regarded vulnerable by FASVERIF. The vulnerability in Ex2 is a kind of *TD*. Although this is a known type of vulnerabilities, FASVERIF still finds 3 contracts with *TD* that are deployed on Ethereum.

FASVERIF runs for up to 5 hours for each contract in the 1413 randomly-selected contracts. In verification of invariant properties, 50% of the contracts are verified within 2044 seconds, and 80% are verified within 4158 seconds. In verification of equivalence properties, 50% of the contracts are verified within 4371 seconds, 80% are verified within 8201 seconds.

# 8 RELATED WORK

## 8.1 Automation-oriented analyzers for smart contracts

Automation-oriented analyzers fall into two categories: analyzers using symbolic execution and analyzers using other technologies.

The analyzers using symbolic execution represent each program trace as a propositional formula and recognize traces that match given patterns or satisfy given constraints. For instance, OYENTE [47] executes EVM bytecode symbolically and checks for various vulnerability patterns in execution traces. Consensys [20] proposes Mythril using taint analysis and symbolic execution to find vulnerability patterns. Osiris [64] is specially designed for detecting arithmetic bugs. Compared with the above tools using symbolic execution, there are differences between FASVERIF and them: 1) FASVERIF uses the formal methods whenever possible in its design of automated inference and provides a proof of our translation. 2) FASVERIF generates security properties on demand instead of using fixed patterns or constraints.

There are also automation-oriented analyzers using other technologies. ContractFuzzer [41] defines test oracles for vulnerabilities and instruments EVM to search for executions that match test oracles. SECURIFY [65] transforms bytecode of smart contracts into DataLog facts and devises compliance and violation patterns to detect safe or unsafe contracts. SmartCheck [61] flags potential vulnerabilities by searching for specific patterns in the XML syntax trees transformed from contracts. VERISMART [58] generates and checks invariants to find the overflow in smart contracts. The vulnerabilities that these tools can detect are either in a particular category or dependent on pre-defined known patterns.

## 8.2 Expressivity-oriented analyzers for smart contracts

Expressivity-oriented tools use formal verification methods for the analysis of contracts to obtain more accurate and reliable results. SMARTPULSE [59] is a tool used to automatically check given temporal properties of smart contracts, including liveness. Similarly, VerX [50] performs a semi-automatic verification of temporal safety specifications written in PastLTL. KEVM [40] provides an executable formal specification of the EVM's bytecode stack-based language built with the K Framework. ConCert [29] proposes a proof framework in Coq for functional smart contract languages. These tools can be used to verify functional properties of smart contracts, which are not currently supported by FASVERIF. However, these tools need human involvement to produce results. Differently, FASVERIF concentrates on security analysis for a large amount of finance-related smart contracts and can generate security properties on-demand automatically. Besides, according to the literature of the above tools, they do not support our equivalence properties.

## 8.3 Hyrid-oriented analyzers for smart contracts

To the best of our knowledge, there are two hybrid-oriented analyzers for smart contracts: eThor [56] and ZEUS [43]. eThor is a sound static analyzer for EVM bytecode, abstracts the semantics of EVM bytecode into a set of Horn clauses and expresses properties in terms of reachability queries, which are solved using Z3. As the literature of eThor states, it can detect *reentrancy* automatically through a pre-defined property. However, as our evaluation in Section 7.3 shown, eThor gets more false positives than FASVERIF in detection of *reentrancy* due to its conservative property. ZEUS [43] transforms Solidity smart contracts into an intermediate language and later into LLVM bitcode which allows for using existing symbolic model checkers. ZEUS uses pre-defined policies, which are designed based on known vulnerability patterns, for bug detection. Thus, ZEUS may miss unknown vulnerabilities or variants of known vulnerabilities, *e.g.*, *cross-function reentrancy* , *TOD-token* and *transferMint* which are supported by FASVERIF.

# 9 DISCUSSIONS AND FUTURE WORK

## 9.1 Limitations

We summarize several limitations of FASVERIF as follows:

**Solidity language is not fully supported.** Due to the Turing-completeness of Solidity [25], it is challenging to fully support its features. Therefore, we add the following restrictions to define a simplified fragment of Solidity supported by FASVERIF.

1) Loops. FASVERIF supports unrolling of bounded loops, i.e., the execution times of loops are constant, where the loop statement is replaced by equivalent statements without loops. The unbounded loops, whose execution times cannot be determined statically, are not supported. As the unbounded loops are costly and are regarded as anti-patterns [60], we recommend the developers not to use them and omit contracts with unbounded loops in our analysis.

2) Revert. FASVERIF verifies the properties under the assumption that all transactions can be executed to completion. For transactions where a revert occurs, we assume that the executions of the transactions do not result in modification of any variables.

3) Contract creation. FASVERIF supports the case of static creation of contracts in the constructor functions. To trade-off efficiency and coverage for the solidity features, we omit the contracts that create contracts via function call since modelling of the behavior would increase the state space of models. Even with this omission, FASVERIF can still cover most real-world contracts, since we only find 9.66%(2954/30577) of contracts in *real-world dataset* that create contracts via function calls.

4) Function call. Given a set of contracts with Solidity codes, FASVERIF requires them not to invoke functions in contracts outside the set whose codes are unknown. This is because FASVERIF can only analyze codes that are given beforehand, which is an inherent defect of static analyzers [49].

**Misclassification of token contracts.** Although our method of identifying token contracts achieves accuracy higher than 98% in *real-world dataset*, it still may 1) miss recognizing token contracts, or 2) falsely recognize non-token contracts as token contracts. For the first kind of contracts, FASVERIF supports custom key variable names to cover them. For the second kind of contracts, which are a small proportion (0.8%, 252/30577, threshold=85) of our *real-world dataset*, FASVERIF does not support identifying them temporarily.

## 9.2 Future works

There are several interesting directions for our future works. Firstly, we plan to optimize our method identifying contracts by taking more information, *e.g.*, control flow, into account. Secondly, We plan to study the automatic generation of exploits based on the outputs of FASVERIF, to evaluate the practicality of our discovered vulnerabilities. Lastly, we are interested in the analysis of contracts written in languages other than Solidity or the analysis of bytecode.

# 10 CONCLUSION

We propose and implement FASVERIF, a system of automated inference for achieving full automation on fine-grained security analysis of Ethereum smart contracts. FASVERIF can automatically generate finance-related security properties on-demand and the corresponding models for the verification, and verify the properties automatically. FASVERIF outperforms other automatic tools in detecting finance-related vulnerabilities with respect to accuracy and coverage of types of vulnerabilities, and it successfully finds 13 contracts with zero-day bugs, including 10 contracts that evade the detection of current automatic tools to the best of our knowledge.

# REFERENCES

[1] 1997. Formalizing and securing relationships on public networks. https://firstmonday.org/ojs/index.php/fm/article/view/548/469. (1997).

[2] 2003. Formal proof sketch. (2003). https://www.cs.ru.nl/F.Wiedijk/pubs/sketches2.pdf

[3] 2016. The DAO hack. https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/. (2016).

[4] 2016. Ethereum Smart Contract Best Practices. https://consensys.github.io/smart-contract-best-practices/development-recommendations/solidity-specific/timestamp-dependence/. (2016).

[5] 2017. Blockchain is empowering the future of insurance. https://techcrunch.com/2016/10/29/blockchain-is-empowering-the-future-of-insurance/. (2017).

[6] 2017. ETHLance. http://ethlance.com/. (2017).

[7] 2017. The Parity bug. http://hackxingdistributed.com/2017/07/22/deep-dive-parity-bug. (2017).

[8] 2018. A Closer Look at ICO Smart Contracts. https://tokeny.com/a-closer-look-at-ico-smart-contracts/. (2018).

[9] 2018. DASP — Top 10. (2018). https://www.dasp.co/

[10] 2018. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts. https://peckshield.medium.com/new-multioverflow-bug-identified-in-multiple-erc20-smart-contracts-cve-2018-10706-8e55946c252c. (2018).

[11] 2019. Fatal TransferMint Bug in Multiple TRC20 Smart Contracts. https://twitter.com/peckshield/status/1115226918855401479?cxt=HHwWjoCj5aq-ivoeAAAA. (2019).

[12] 2020. fuzzywuzzy 0.18.0. https://pypi.org/project/fuzzywuzzy/. (2020).

[13] 2020. SWC-107: Reentrancy. https://swcregistry.io/docs/SWC-107. (2020).

[14] 2020. SWC Registry. https://swcregistry.io/. (2020).

[15] 2021. ERC20 standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md. (2021).

[16] 2021. How to defend against attacks on DeFi. (2021). https://biz.crast.net/the-evolution-of-the-blockchain-industry-and-how-to-defend-against-attacks-on-defi/

[17] 2021. Main repository for SmartPulse. (2021). https://github.com/utopia-group/SmartPulseTool

[18] 2021. The Poly Network attack. https://en.wikipedia.org/wiki/Poly_Network_exploit. (2021).

[19] 2022. AICoin — information about Ethereum. (2022). https://www.aicoin.com/currencies/ethereum.html?lang=en

[20] 2022. ConsenSys. https://www.consensys.net/. (2022).

[21] 2022. difflib — Helpers for computing deltas. (2022). https://docs.python.org/3/library/difflib.html

[22] 2022. ERC721 standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md. (2022).

[23] 2022. Etherscan. https://etherscan.io/. (2022).

[24] 2022. go-ethereum. https://github.com/ethereum/go-ethereum/blob/4e474c74dc2ac1d26b339c32064d0bac98775e77/consensus/ethash/consensus.go. (2022).

[25] 2022. Solidity documentation. : https://solidity.readthedocs.io/en/latest. (2022).

[26] 2022. Source codes of EVM. https://github.com/ethereum/go-ethereum/tree/master/core/vm/evm.go. (2022).

[27] 2022. Z3: An efficient SMT solver. https://www.microsoft.com/en-us/research/project/z3-3/. (2022).

[28] 2022. Zero-day_(computing). (2022). https://en.wikipedia.org/wiki/Zero-day_(computing)

[29] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. 2020. ConCert: a smart contract certification framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 215–228. https://doi.org/10.1145/3372885.3373829

[30] Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. 2021. Clockwork Finance: Automated Analysis of Economic Security in Smart Contracts. *CoRR* abs/2109.04347 (2021). arXiv:2109.04347 https://arxiv.org/abs/2109.04347

[31] David A. Basin, Jannik Dreier, and Ralf Sasse. 2015. Automated Symbolic Proofs of Observational Equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 1144–1155. https://doi.org/10.1145/2810103.2813662

[32] David A. Basin, Ralf Sasse, and Jorge Toro-Pozo. 2021. The EMV Standard: Break, Fix, Verify. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1766–1781. https://doi.org/10.1109/SP40001.2021.00037

[33] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses. *ACM Comput. Surv.* 53, 3 (2020), 67:1–67:43. https://doi.org/10.1145/3391195

[34] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2012. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.* 77, 9 (2012), 1006–1036. https://doi.org/10.1016/j.scico.2010.07.004

[35] ConsenSys. 2022. Mythril. https://github.com/ConsenSys/mythril. (2022).

[36] Cas Cremers, Benjamin Kiesl, and Niklas Medinger. 2020. A Formal Analysis of IEEE 802.11's WPA2: Countering the Kracks Caused by Cracking the Counters. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1–17. https://www.usenix.org/conference/usenixsecurity20/presentation/cremers

[37] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47, 587 Ethereum smart contracts. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 530–541. https://doi.org/10.1145/3377811.3380364

[38] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. IEEE / ACM, 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[39] Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David A. Basin. 2020. A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1857–1874. https://www.usenix.org/conference/usenixsecurity20/presentation/girol

[40] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 204–217. https://doi.org/10.1109/CSF.2018.00022

[41] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 259–269. https://doi.org/10.1145/3238147.3238177

[42] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanán, Yang Liu, and Jun Sun. 2020. Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1695–1712. https://doi.org/10.1109/SP40000.2020.00066

[43] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf

[44] Harith Kamarul. 2020. Ethereum in 2020: the View from the Block Explorer. https://medium.com/etherscan-blog/ethereum-in-2020-the-view-from-the-block-explorer-2f9a1db2ee15. (2020).

[45] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2019. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 363–373. https://doi.org/10.1145/3293882.3330560

[46] Steve Kremer and Robert Künnemann. 2014. Automated Analysis of Security Protocols with Global State. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 163–178. https://doi.org/10.1109/SP.2014.18

[47] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 254–269. https://doi.org/10.1145/2976749.2978309

[48] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 696–701. https://doi.org/10.1007/978-3-642-39799-8_48

[49] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. 2019. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version). *ACM Trans. Priv. Secur.* 22, 2 (2019), 14:1–14:34. https://doi.org/10.1145/3313391

[50] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1661–1677. https://doi.org/10.1109/SP40000.2020.00024

[51] Grigore Roşu. 2017. Matching logic. *Logical Methods in Computer Science* 13, 4 (December 2017), 1–61. https://doi.org/abs/1705.06312

[52] Grigore Roşu and Andrei Ştefănescu. 2012. Checking Reachability using Matching Logic. In *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. ACM, 555–574. https://doi.org/citation.cfm?doid=2384616.2384656

[53] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/sereum-protecting-existing-smart-contracts-against-re-entrancy-attacks/

[54] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

[55] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (2002), 217–298. https://doi.org/10.1145/514188.514190

[56] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 621–640. https://doi.org/10.1145/3372297.3417250

[57] Cheng Shi and Kazuki Yoneyama. 2020. Formal Verification of Fair Exchange Based on Bitcoin Smart Contracts. In *International Conference on Cryptology in India*. Springer, 89–106.

[58] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1678–1694. https://doi.org/10.1109/SP40000.2020.00032

[59] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *42nd IEEE Symposium on Security and Privacy*. IEEE. https://www.microsoft.com/en-us/research/publication/smartpulse-automated-checking-of-temporal-properties-in-smart-contracts/

[60] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018*, Roberto Tonelli, Giuseppe Destefanis, Steve Counsell, and Michele Marchesi (Eds.). ACM, 9–16. https://doi.org/10.1145/3194113.3194115

[61] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018*. ACM, 9–16. http://ieeexplore.ieee.org/document/8445052

[62] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2020. A Survey of Smart Contract Formal Specification and Verification. *CoRR* abs/2008.02712 (2020). arXiv:2008.02712 https://arxiv.org/abs/2008.02712

[63] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 664–676. https://doi.org/10.1145/3274694.3274737

[64] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 664–676. https://doi.org/10.1145/3274694.3274737

[65] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 67–82. https://doi.org/10.1145/3243734.3243780

[66] Xiaobing Wang, Xiaoyu Yang, and Chunyi Li. 2020. A formal verification method for smart contract. In *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 31–36.

# A APPENDIX

## A.1 Complete definition of function $\mathcal{R}$

As mentioned in Section 4.2, if a statement in function $f$ uses `block.timestamp`, the rule `ext_call_bvar` is generated instead of `ext_call`:

$$\begin{aligned}&[\mathsf{Fr}(\sigma_v(c_b)), \mathsf{FR}(\sigma(seq(\mathsf{d}))), \mathsf{Fr}(bt)] - [] \rightarrow [\mathsf{Bvar}(bt),\\ &\qquad \mathsf{Call_e}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b) \rrbracket \cdot \sigma(seq(\mathsf{d})))] \quad (\texttt{ext\_call\_bvar})\end{aligned}$$

where $bt$ is a term denoting `block.timestamp`. The rule implies that a transaction can be sent from an external account and the timestamp of the block containing the transaction is arbitrary.

The complete definition of function $\mathcal{R}$ in Section 4.2 is shown in Fig. 8. In rule `send_succ_assign` and `send_succ_assign`, `TRUE` and `FALSE` are constant terms denoting boolean values true and false, respectively. The function $\mathcal{R}$ translates five categories of statements into rules, since the translation of assignment statements, return statements and ether transfer statements are introduced previously, here we introduce the translation of statements in the remaining two categories:

**1)** Conditional statements. Two rules `if_true` and `if_false` are generated for statement `if` $e_b$ `then stmt`$_1$ `else stmt`$_2$. Here, $\theta_e(e_b)$ is a conditional fact representing that the value of $e_b$ is true. Similarly, $\theta_{ne}(e_b)$ denotes that the value of $e_b$ is false. $\mathsf{Var}_{i\circ1}$, $\mathsf{Var}_{i\circ2}$ correspond to the states when `stmt`$_1$, `stmt`$_2$ start to be executed, respectively.

**2)** Internal call statements. Besides external accounts, contract account $c$ can also invoke the function $f_x$ of another account $c_x$ by executing the statement $x(c_x).f_x(p)$, where $p$ is a sequence of parameters and $x$ is name of the contract of account $c_x$. In this case, the execution of this statement can be divided into the following steps: **a)** An internal transaction is sent by $c$ to invoke $f_x$. To denote this step, rule `in_call` is generated. Because the ether balances may be modified during executions of $f_x$, while local variables will not change, terms denoting local variables in rule `in_call` are maintained in $\mathsf{Var}_{i\circ1}$ fact, while terms representing the ether balances of all accounts are put into Evar and terms denoting global variables of account $c$ are put into Gvar. **b)** The codes in $f_x$ are executed, which have been recursively modeled as shown in Fig. 3. **c)** $f_x$ returns, which has been modeled in return statement category. **d)** The next statement of $f_x$ is prepared to be executed. Rule `recv_ret` is generated for this step, indicating that $c$ receives the return message from $c_x$ and $f$ is ready to continue executing.

## A.2 The complete form of rules in Section 6.1

The complete form of rules mentioned in Section 6.1 is shown in Fig. 9. Two restrictions are associated with rule `recv_ext_A` and `recv_ext_B`:

$$\begin{aligned}&\mathit{All}\ \#i\ \#j\ c_b.\mathsf{Exc_A}(\sigma_v(c_b)), \sigma_a(f))@i\ \&\ \mathsf{Exc_A}(\sigma_v(c_b)), \sigma_a(f))@j\\ &\qquad\qquad\qquad\qquad\qquad\qquad => i = j\end{aligned}$$

$$\begin{aligned}&\mathit{All}\ \#i\ \#j\ c_b.\mathsf{Exc_B}(\sigma_v(c_b)), \sigma_a(f))@i\ \&\ \mathsf{Exc_B}(\sigma_v(c_b)), \sigma_a(f))@j\\ &\qquad\qquad\qquad\qquad\qquad\qquad => i = j\end{aligned}$$

## A.3 Correctness of the translations

To prove correctness of the translation from Solidity language to our models, we firstly introduce the custom semantics of Solidity [42], namely KSolidity. We also briefly explain the relations between the modeling in FASVERIF and KSolidity, including the configurations

$$
\begin{aligned}
&\mathcal{R}(\texttt{function}\ f(\mathsf{d})\{\texttt{stmt}\}, \varnothing, \omega_0) = \mathcal{R}(\texttt{stmt}, 1, \llbracket \langle \sigma_a(f), \mathsf{T_c}, \mathsf{R_o}, \mathsf{E_n}\rangle, \langle \sigma_v(c_b), \mathsf{T_v}, \mathsf{R_l}, \mathsf{E_n}\rangle, \langle \sigma_v(calltype), \mathsf{T_v}, \mathsf{R_l}, \mathsf{E_n}\rangle, \langle \sigma_v(\mathsf{depth}), \mathsf{T_v}, \mathsf{R_l}, \mathsf{E_n}\rangle \rrbracket \\
&\qquad \cdot \omega_0 \cdot seq(\mathsf{d})) \cup \{[\mathsf{Fr}(\sigma_v(c_b)), \mathsf{FR}(\sigma(seq(\mathsf{d})))] - [] \rightarrow [\mathsf{Call_e}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b)\rrbracket \cdot \sigma(seq(\mathsf{d})))], && \texttt{(ext\_call)} \\
&\qquad [\mathsf{Call_e}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b)\rrbracket \cdot \sigma(seq(\mathsf{d}))), \mathsf{Evar}(e(\omega_0)), \mathsf{Gvar}(\llbracket \omega_0[1]\rrbracket \cdot g(\omega_0)\backslash e(\omega_0))] - [] \rightarrow \\
&\qquad\qquad [\mathsf{Var}_1(\llbracket \sigma_a(f), \sigma_v(c_b), \mathsf{EXT}, 0\rrbracket \cdot \sigma(\omega_0) \cdot \sigma(seq(\mathsf{d})))], && \texttt{(recv\_ext)} \\
&\qquad [\mathsf{Call_{in}}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b), \sigma_v(\mathsf{depth})\rrbracket \cdot \sigma(seq(\mathsf{d}))), \mathsf{Evar}(e(\omega_0)), \mathsf{Gvar}(\llbracket \omega_0[1]\rrbracket \cdot g(\omega_0)\backslash e(\omega_0))] - [] \rightarrow \\
&\qquad\qquad [\mathsf{Var}_1(\llbracket \sigma_a(f), \sigma_v(c_b), \mathsf{IN}, \sigma_v(\mathsf{depth})\rrbracket \cdot \sigma(\omega_0) \cdot \sigma(seq(\mathsf{d})))]\} && \texttt{(recv\_in)}
\end{aligned}
$$

$$
\mathcal{R}(v_1 \leftarrow v_2; \texttt{stmt}, i, \omega) = \mathcal{R}(\texttt{stmt}, i \circ 1, \omega) \cup \{[\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 1}(\sigma(\omega)|\frac{\sigma_v(v_1)}{\sigma_v(v_2)})]\} \qquad \texttt{(var\_assign)}
$$

$$
\mathcal{R}(\tau\ v_1 \leftarrow v_2; \texttt{stmt}, i, \omega) = \mathcal{R}(\texttt{stmt}, i \circ 1, \omega \cdot \llbracket \langle \sigma_v(v_1), \mathsf{T_v}, \mathsf{R_l}, \mathsf{E_n}\rangle \rrbracket) \cup \{[\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 1}(\sigma(\omega) \cdot \llbracket \sigma_v(v_2)\rrbracket)]\} \qquad \texttt{(var\_declare)}
$$

$$
\begin{aligned}
&\mathcal{R}(\texttt{if}\ e_b\ \texttt{then}\ \texttt{stmt}_1\ \texttt{else}\ \texttt{stmt}_2; \texttt{stmt}_3, i, \omega) = \mathcal{R}(\texttt{stmt}_1; \texttt{stmt}_3, i \circ 1, \omega) \cup \mathcal{R}(\texttt{stmt}_2; \texttt{stmt}_3, i \circ 2, \omega) \cup \\
&\qquad \{[\mathsf{Var}_i(\sigma(\omega))] - [\theta_e(e_b)] \rightarrow [\mathsf{Var}_{i\circ 1}(\sigma(\omega))], && \texttt{(if\_true)} \\
&\qquad [\mathsf{Var}_i(\sigma(\omega))] - [\theta_{ne}(e_b)] \rightarrow [\mathsf{Var}_{i\circ 2}(\sigma(\omega))]\} && \texttt{(if\_false)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{R}(\texttt{require}\ e_b\ \texttt{stmt}_1, i, \omega) = \mathcal{R}(\texttt{stmt}_1, i \circ 1, \omega) \cup \{[\mathsf{Var}_i(\sigma(\omega))] - [\theta_e(e_b)] \rightarrow [\mathsf{Var}_{i\circ 1}(\sigma(\omega))], && \texttt{(require\_true)} \\
&\qquad [\mathsf{Var}_i(\sigma(\omega))] - [\theta_{ne}(e_b)] \rightarrow []\} && \texttt{(require\_false)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{R}(\texttt{return}, i, \omega) = \{[\mathsf{Var}_i(\sigma(\omega))] - [\mathsf{Pred\_eq}(\omega[3], \mathsf{EXT})] \rightarrow [\mathsf{Gvar}(\llbracket \omega[5]\rrbracket \cdot g(\omega)\backslash e(\omega)), \mathsf{Evar}(e(\omega))], && \texttt{(ret\_ext)} \\
&\qquad [\mathsf{Var}_i(\sigma(\omega))] - [\mathsf{Pred\_eq}(\omega[3], \mathsf{IN})] \rightarrow [\mathsf{Return}(\llbracket \omega[5], \omega[1], \omega[2], \omega[4]\rrbracket, \mathsf{Gvar}(\llbracket \omega[5]\rrbracket \cdot g(\omega)\backslash e(\omega)), \mathsf{Evar}(e(\omega)))]\} && \texttt{(ret\_in)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{R}(x(c_x).f_x(p); \texttt{stmt}, i, \omega) = \mathcal{R}(\texttt{stmt}, i \circ 1 \circ 1, \omega) \cup \{[\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Call_{in}}(\llbracket \sigma_a(c_x), \sigma_a(f_x), \omega[5], \omega[4] \oplus 1\rrbracket \cdot \sigma_s(p)), \mathsf{Var}_{i\circ 1}(\sigma(\omega)\backslash g(\omega)), \\
&\qquad\qquad \mathsf{Gvar}(\llbracket \omega[5]\rrbracket \cdot g(\omega)\backslash e(\omega)), \mathsf{Evar}(e(\omega))], \\
& && \texttt{(in\_call)} \\
&\qquad [\mathsf{Return}(\llbracket \sigma_a(c_x), \sigma_a(f_x), \omega[5], \sigma_v(\mathsf{r\_depth})\rrbracket), \mathsf{Var}_{i\circ 1}(\sigma(\omega)\backslash g(\omega)), \mathsf{Gvar}(\llbracket \omega[5]\rrbracket \cdot g(\omega)\backslash e(\omega)), \mathsf{Evar}(e(\omega))] \\
&\qquad\qquad - [\mathsf{Pred\_eq}(\sigma_v(\mathsf{r\_depth}), \omega[4] \oplus 1)] \rightarrow [\mathsf{Var}_{i\circ 1\circ 1}(\sigma(\omega))]\} \\
& && \texttt{(recv\_ret)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{R}(c_x.\texttt{transfer}(v_1); \texttt{stmt}, i, \omega) = \mathcal{R}(\texttt{stmt}, i \circ 1, \omega) \cup \\
&\quad \{[\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 1}(\sigma(\omega)|\frac{\sigma_v(c_x)}{\sigma_v(c_x) \oplus \sigma_v(v_1)}|\frac{\sigma_v(c)}{\sigma_v(c) \ominus \sigma_v(v_1)})], && \texttt{(transfer\_succ)} \\
&\quad [\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [], && \texttt{(transfer\_fail)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{R}(c_x.\mathsf{send}(v_1); \texttt{stmt}, i, \omega) = \mathcal{R}(\texttt{stmt}, i \circ 1, \omega) \cup \mathcal{R}(\texttt{stmt}, i \circ 2, \omega) \cup \\
&\quad \{[\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 1}(\sigma(\omega)|\frac{\sigma_v(c_x)}{\sigma_v(c_x) \oplus \sigma_v(v_1)}|\frac{\sigma_v(c)}{\sigma_v(c) \ominus \sigma_v(v_1)})], && \texttt{(send\_succ)} \\
&\quad [\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 2}(\sigma(\omega))], && \texttt{(send\_fail)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{R}(c_x.\texttt{call}().\mathsf{value}(v_1); \texttt{stmt}, i, \omega) = \mathcal{R}(\texttt{stmt}, i \circ 1, \omega) \cup \mathcal{R}(\texttt{stmt}, i \circ 2, \omega) \cup \mathcal{R}(\texttt{stmt}, i \circ 3 \circ 1, \omega) \cup \\
&\quad \{[\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 1}(\sigma(\omega)|\frac{\sigma_v(c_x)}{\sigma_v(c_x) \oplus \sigma_v(v_1)}|\frac{\sigma_v(c)}{\sigma_v(c) \ominus \sigma_v(v_1)})], && \texttt{(ether\_succ)} \\
&\quad [\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 2}(\sigma(\omega))], && \texttt{(ether\_fail)} \\
&\quad [\mathsf{Var}_i(\sigma(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 3}(l(\omega)), \mathsf{Fallback}(\llbracket \omega[5], \omega[1]\rrbracket), \mathsf{Gvar}(\llbracket \omega[5]\rrbracket \cdot g(\omega)\backslash e(\omega)), \mathsf{Evar}(e(\omega)|\frac{\sigma_v(c_x)}{\sigma_v(c_x) \oplus \sigma_v(v_1)}|\frac{\sigma_v(c)}{\sigma_v(c) \ominus \sigma_v(v_1)})], && \texttt{(fb\_call)} \\
&\quad [\mathsf{ReturnFallback}(\llbracket \omega[5], \omega[1]\rrbracket), \mathsf{Gvar}(\llbracket \omega[5]\rrbracket \cdot g(\omega)\backslash e(\omega)), \mathsf{Evar}(e(\omega)), \mathsf{Var}_{i\circ 3}(l(\omega))] - [] \rightarrow [\mathsf{Var}_{i\circ 3\circ 1}(\sigma(\omega))]\} && \texttt{(recv\_fb\_ret)}
\end{aligned}
$$

**Figure 8: Complete definition of function $\mathcal{R}$.**

$$
\begin{aligned}
&[\mathsf{Fr}(\sigma_v(c_b)), \mathsf{FR}(\sigma(seq(\mathsf{d})))] - [] \rightarrow [\mathsf{Call_{Ae}}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b)\rrbracket \cdot \sigma(seq(\mathsf{d}))), \mathsf{Call_{Be}}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b)\rrbracket \cdot \sigma(seq(\mathsf{d})))] && \texttt{(ext\_call\_AB)} \\
&[\mathsf{Fr}(\sigma_v(c_b)), \mathsf{FR}(\sigma(seq(\mathsf{d}))), \mathsf{Fr}(bt_A), \mathsf{Fr}(bt_B)] - [] \rightarrow [\mathsf{Bvar_A}(bt_A), \mathsf{Bvar_B}(bt_B), \mathsf{Call_{Ae}}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b)\rrbracket \cdot \sigma(seq(\mathsf{d}))), \\
&\qquad \mathsf{Call_{Be}}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b)\rrbracket \cdot \sigma(seq(\mathsf{d})))] && \texttt{(ext\_call\_bvar\_AB)} \\
&[\mathsf{Call_{Ae}}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b)\rrbracket \cdot \sigma(seq(\mathsf{d}))), \mathsf{Evar_A}(e(\omega_0)), \mathsf{Gvar_A}(\llbracket \omega_0[1]\rrbracket \cdot g(\omega_0)\backslash e(\omega_0))] - [\mathsf{Exc_A}(\sigma_v(c_b)), \sigma_a(f))] \rightarrow \\
&\qquad [\mathsf{Var}_{A1}(\llbracket \sigma_a(f), \sigma_v(c_b), \mathsf{EXT}\rrbracket \cdot \sigma(\omega_0) \cdot \sigma(seq(\mathsf{d})))] && \texttt{(recv\_ext\_A)} \\
&[\mathsf{Call_{Be}}(\llbracket \omega_0[1], \sigma_a(f), \sigma_v(c_b)\rrbracket \cdot \sigma(seq(\mathsf{d}))), \mathsf{Evar_B}(e(\omega_0)), \mathsf{Gvar_B}(\llbracket \omega_0[1]\rrbracket \cdot g(\omega_0)\backslash e(\omega_0))] - [\mathsf{Exc_B}(\sigma_v(c_b)), \sigma_a(f))] \rightarrow \\
&\qquad [\mathsf{Var}_{B1}(\llbracket \sigma_a(f), \sigma_v(c_b), \mathsf{EXT}\rrbracket \cdot \sigma(\omega_0) \cdot \sigma(seq(\mathsf{d})))] && \texttt{(recv\_ext\_B)}
\end{aligned}
$$

**Figure 9: Complete form of four rules in Section 6.1.**

$$
\left\langle
\begin{array}{c}
\langle \texttt{\$PGM:SourceUnit} \rangle_{k} \\
\left\langle \begin{array}{c} \langle \text{ListItem(-1)} \rangle_{\mathit{contractStack}} \ \langle \text{.List} \rangle_{\mathit{functionStack}} \\ \langle \text{.List} \rangle_{\mathit{newStack}} \ \langle \text{.List} \rangle_{\mathit{blockStack}} \end{array} \right\rangle_{\mathit{controlStacks}} \\
\left\langle \begin{array}{c} \langle \text{0:Int} \rangle_{\mathit{cntContractDefs}} \\ \langle \langle \text{.K} \rangle_{\mathit{cName}} \ \langle \text{.List} \rangle_{\mathit{stateVars}} \ \langle \text{false} \rangle_{\mathit{Constructor}} \ \ldots \rangle_{\mathit{contract}^*} \end{array} \right\rangle_{\mathit{contracts}} \\
\left\langle \begin{array}{c} \langle \text{0:Int} \rangle_{\mathit{cntFuctions}} \\ \left\langle \begin{array}{c} \langle \text{0:Int} \rangle_{\mathit{fId}} \ \langle \text{.K} \rangle_{\mathit{fName}} \ \langle \text{.K} \rangle_{\mathit{inputParameters}} \\ \langle \text{.K} \rangle_{\mathit{returnParameters}} \ \langle \text{.K} \rangle_{\mathit{Body}} \ \langle \text{.K} \rangle_{\mathit{FunQuantifiers}} \end{array} \ldots \right\rangle_{\mathit{function}^*} \end{array} \right\rangle_{\mathit{functions}} \\
\left\langle \begin{array}{c} \langle \text{0:Int} \rangle_{\mathit{cntContracts}} \\ \left\langle \begin{array}{c} \langle \text{(-1):Int} \rangle_{\mathit{ctId}} \ \langle \text{.K} \rangle_{\mathit{ctName}} \ \langle \text{.Map} \rangle_{\mathit{ctContext}} \\ \langle \text{.Map} \rangle_{\mathit{globalContext}} \ \langle \text{.Map} \rangle_{\mathit{ctType}} \ \langle \text{.Map} \rangle_{\mathit{ctLocation}} \\ \langle \text{.Map} \rangle_{\mathit{ctStorage}} \ \langle \text{.Map} \rangle_{\mathit{Memory}} \\ \langle \text{0:Int} \rangle_{\mathit{slotNum}} \ \langle \text{0:Int} \rangle_{\mathit{Balance}} \end{array} \ldots \right\rangle_{\mathit{contractInstance}^*} \end{array} \right\rangle_{\mathit{contractInstances}} \\
\left\langle \begin{array}{c} \langle \text{1:Int} \rangle_{\mathit{cntTrans}} \ \langle \text{0 |-> "Main"} \rangle_{\mathit{tranComputation}} \ \langle \text{.K} \rangle_{\mathit{Msg}} \ \langle \text{.List} \rangle_{\mathit{msgStack}} \\ \langle \text{0:Int} \rangle_{\mathit{gasConsumption}} \ \langle \text{.List} \rangle_{\mathit{gasStack}} \end{array} \right\rangle_{\mathit{transactions}}
\end{array}
\right\rangle_{T}
$$

**Figure 10: An example of configuration in KSolidity semantics**

and rules for the configurations. Then, we explain the notations and adopted theories preparing for the proofs. Finally, we prove the soundness of FASVERIF.

### *A.3.1* ***A glance at K-framework and KSolidity [42].***

KSolidity is defined using K-framework [54], a rewrite-based executable semantic framework. The definition of a language consists of 3 parts: language syntax, the runtime configuration, and a set of rules constructed based on the syntax and the configuration.

Configurations organize the state in units called cells, which are labeled and can be nested. The cells' contents can be various semantic data, such as trees, lists, maps, etc. As shown in Figure 10, the configuration of Solidity is composed of six main cells: (1) *k*, the rest of programs to be executed, (2) *controlStacks*, a collection of runtime stacks, (3) *contracts*, a set of function definitions, (4) *functions*, a set of contract definitions (5) *contractInstances*, a set of contract instances and (6) *transactions*, information for the runtime transactions.

When an event occurs in the blockchain, e.g., a statement in a function of a smart contract is executed, the contents in the configuration are updated, i.e., a new configuration is yielded. The rules in the K-framework describe how the configuration is yielded upon different events. For example, a configuration $c$ can be yielded to a new configuration by applying rule ReadAddress-LocalVariables, if the first fragment of code in $c$'s $k$ cell matches pattern `readAddress( Addr:Int,String2Id("Local"))` and the top of $c$ *contractStack* records integer value `N`, which corresponds to `ctId` of $c$'s *contractInstance* cell that records `Addr |-> V` in sub-cell *Memory*. As a result, the first fragment in $c$ is substituted by `gasCal(#read,String2Id("Local"))` $\curvearrowright$ `V`, while the rest of the configuration stays unchanged. In practice, the rule means that when `readAddress` is executed, the value `V` is read from contract `N`'s local variable whose address is `Addr`, before which some gas has been consumed. The notation $\curvearrowright$ is a list constructor (read "followed by"). The detailed explanations on the configurations and the rules can be found in [42]. We will introduce the notations and theories, which will then be used together with the k-framework for the proofs.

RULE ReadAddress-LocalVariables

$$
\left\langle \cfrac{\texttt{readAddress(Addr:Int,String2Id("Local"))}}{\texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright \texttt{V:Value}} \cdots \right\rangle_{k}
$$
$$
\langle \texttt{ListItem(N:Int)} \ \ldots \rangle_{\mathit{contractStack}}
$$
$$
\left\langle \begin{array}{c} \langle \texttt{N} \rangle_{\mathit{ctId}} \\ \langle \ldots \ \texttt{Addr |-> V} \ \ldots \rangle_{\mathit{Memory}} \end{array} \cdots \right\rangle_{\mathit{contractInstance}}
$$

**Relationship between facts and configurations**: The configurations In FASVERIF are mainly represented by the multiset of facts. We give examples of the relationship between the facts in FASVERIF and configurations in the K-framework, as well as the relationship between the corresponding rules, to facilitate the understanding of proofs in Appendix A.3.3.

Suppose that a function is currently executing, and the PC is currently at a certain point, denoted as index $i$. As mentioned in Section 4, fact $\mathsf{Var}_i$ represents the current configuration in the model generated by FASVERIF. We demonstrate the corresponding configuration in KSolidity as follows:

- The sequence of the first three terms in $\mathsf{Var}_i$, denoted as $[\![\sigma_a(f), \sigma_v(c_b), \mathit{calltype}]\!]$, represents the current executed function $f$, contract account $c_b$ who invokes the function, and the type *calltype* indicating whether $c_b$ is an external account or a contract account. Correspondingly in KSolidity, the name $f$ and $c_b$ is statically recorded in sub-cell *fName* and *cName*, respectively; and the runtime information about $f$ and $c_b$ is stored in cell *controlStacks*. In detail, the *fId* of $f$ is stored on top of the stack *functionStack* , and the *cId* of $c_b$ is the second from the top of the stack *contractStack*.
- The 4th term in $\mathsf{Var}_i$, denoted as $\sigma_a(c)$, represents the contract account whose function $f$ is being executed. In KSolidity, the *cId* of $c$ is top of the stack *contractStack*.
- The sequence after the 4th terms in $\mathsf{Var}_i$ represents the concatenation of three sequences: the global variables of account $c$, ether balances of all accounts, and the current local variables. In KSolidity, global variables, balances, local

variables are stored in *ctStorage*, *Balance*, *and Memory*, respectively.

Consider the case of initializations in the independent model of FASVERIF, which prepares for a new transaction. Assume that the contract account $c$'s function $f$ is to be invoked by $c_b$ at the beginning of the transaction. Note that the contract accounts, e.g., $c$, have been created, and it is also assumed that some transactions may have finished execution before the initialization. The facts $\mathsf{EVar}(e(\omega_0))$ and $\mathsf{GVar}(\llbracket\sigma_a(c)\rrbracket \cdot g(\omega_0)\backslash e(\omega_0))$ are generated by applying the rules init_evars and init_gvars, which means the initial values of balances and global variables of account $c$ are set arbitrarily. The initial configuration can be represented by $\llbracket\sigma_a(f)$, $\sigma_v(c_b)$, $\mathit{calltype}{=}\mathsf{EXT}, \sigma_a(c)\rrbracket \cdot g(\omega_0)\backslash e(\omega_0) \cdot e(\omega_0)$, since there are no local variables.

The first configuration in KSolidity corresponds to the one when the blockchain starts running. For instance, the number of created contract accounts is 0, as counted in *cntContracts*. Hence, the initial configuration in FASVERIF corresponds to a certain configuration, not its first configuration, in KSolidity. In the derivation of the configuration in the K-framework, contract accounts have been created, i.e., rule NEW-CONTRACT-INSTANCE-CREATION has been applied for each creation, and transactions may also be executed, i.e, multiple corresponding rules may be applied as well. Finally, the application of the rules results in changing the value of the cells, e.g., *cntContracts*, in the configuration.

*A.3.2* ***Notations and theories***. We recall definitions and theories in Tamarin prover [46, 48], K-framework [51] and matching logic [51] necessarily needed for proofs in Appendix A.3.3 as follows.

**Tamarin [46, 48]**: Given a set $S$ we denote by $S^*$ the set of finite sequences of elements from $S$ and by $S^\#$ the set of finite multisets of elements from $S$. We use the superscript # to annotate usual multiset operation, e.g., $S1 \cup^\# S_2$ denotes the multiset union of multisets $S_1, S_2$. Set membership modulo $E$ is denoted by $\in_E$ and defined as $e \in_E S$ if $\exists e' \in S.e' =_E e$. Define the set of facts as the set $\mathcal{F}$ consisting of all facts $F(t_1, \ldots, t_k)$, where $t_i$ are the terms. Denote $\mathit{names}(F)$ as the multiset of signature names of the facts in $\mathcal{F}$. For a fact $f$ we denote by $\mathit{ginsts}(f)$ the set of ground instances, i.e. instances that do not contain variables, of $f$.

*Definition A.1.* (Multiset rewrite rule). *A labelled multiset rewrite rule ri is a triple $(l, a, r)$, $l, a, r \in \mathcal{F}^*$, written $l - [a] \to r$. We call $l = \mathit{prems}(ri)$ the premises, $a = \mathit{actions}(ri)$ the actions, and $r = \mathit{conclusions}(ri)$ the conclusions of the rule.*

*Definition A.2.* (Labelled multiset rewriting system). *A labelled multiset rewriting system is a set of labelled multiset rewrite rules R, such that each rule $l - [a] \to r \in R$ satisfies the following conditions:*

- *$l, a, r$ do not contain fresh names*
- *$r$ does not contain* Fr*-facts*

We define one distinguished rule Fresh which is the only rule allowed to have Fr-facts on the right-hand side

$$[] - [] \to [\mathsf{Fr}(x : fresh)] \qquad (\mathtt{Fresh})$$

*Definition A.3.* (Labelled transition relation). *Given a multiset rewriting system R, define the labeled transition relation $\to_R \subseteq \mathcal{G}^\# \times \mathcal{P}(\mathcal{G})^\# \times \mathcal{G}^\#$ as*

$$S \xrightarrow{a}_R ((S\backslash^\# \mathit{lfacts}(l)) \cup^\# r)$$

*if and only if $l - [a] \to r \in_E \mathit{ginsts}(R \cup \mathtt{Fresh})$, $\mathit{lfacts}(l) \subseteq^\# S$ and $\mathit{pfacts}(l) \subseteq S$.*

Here, we denote $\mathcal{G}$ as the set of all ground facts, i.e., facts that do not contain variables. Given a sequence or set of facts $S$ we denote by $\mathit{lfacts}(S)$ the multiset of all linear facts in $S$ and $\mathit{pfacts}(S)$ the set of all persistent facts in $S$. Since the persistent facts are not used in FASVERIF, we trivially conclude the following lemma.

LEMMA A.4. *[Simplified labelled transition relation] Given a multiset rewriting system R that does not have persistent facts,*

$$S \xrightarrow{a}_R ((S\backslash^\# l) \cup^\# r)$$

*if and only if $l - [a] \to r \in_E \mathit{ginsts}(R \cup \mathtt{Fresh})$, $l \subseteq^\# S$.*

*Definition A.5.* (MSR-executions) *Given a multiset rewriting system R we define its set of executions as*

$$\begin{aligned} \mathit{exec}^{msr}(R) = \{ & \emptyset \xrightarrow{r_1}_R \ldots \xrightarrow{r_n}_R S_n \mid \forall a, i, j : 0 \leq i \neq j < n. \\ & (S_{i+1}\backslash^\# S_i) = \{\mathrm{Fr}(a)\} \Rightarrow (S_{j+1}\backslash^\# S_j) \neq \{\mathrm{Fr}(a)\}\} \end{aligned}$$

The definition indicates that the rule Fresh is at most fired once for each name in the transition sequence. If the $i$th step is executed by applying a key rule, $r_i$ is the rule's name; otherwise, $r_i$ is an empty string $\emptyset$. The key rules are in the set KEYRULES = {recv_ext, recv_in, var_assign, var_declare, if_true, if_false, require_true, require_false, ret_ext, ret_in, in_call, ether_succ, ether_fail, fb_call, transfer_succ, transfer_fail, send_succ, send_fail}, which is also shown in Table 3.

*Definition A.6.* (MSR-traces) *The set of traces is defined as*

$$\begin{aligned} \mathit{traces}^{msr}(R) = \{ & [r_1, \ldots, r_n] \mid \forall 0 \leq i \leq n. r_i \neq \emptyset \\ & \mathit{and}\ \emptyset \overset{r_1}{\Longrightarrow}_R \ldots \overset{r_n}{\Longrightarrow}_R S_n \in \mathit{exec}^{msr}(R)\} \end{aligned}$$

*where* $\overset{r_1}{\Longrightarrow}_R$ *is defined as* $\xrightarrow{\emptyset}{}^*_R \xrightarrow{r_1}_R \xrightarrow{\emptyset}{}^*_R$.

**K-framework [54] and matching logic [51]**:

A signature $\Sigma$ is a pair $(S, F)$ where $S$ is a set of sorts and $F$ is a set of operations $f : w \to s$, where $f$ is an operation symbol, $w \in S^*$ is its arity, and $s \in S$ is its result sort. If $w$ is the empty word $\epsilon$ then $f$ is a constant. The universe of terms $T_\Sigma$ associated to a signature $\Sigma$ contains all the terms which can be formed by iteratively applying symbols in $F$ over existing terms (using constants as basic terms, to initiate the process), matching the arity of the symbol being applied with the result sorts of the terms it is applied to. Given an S-sorted set of variables $\mathcal{X}$, the universe of terms $T_\Sigma(\mathcal{X})$ with operation symbols from $F$ and variables from $\mathcal{X}$ consists of all terms in $T_\Sigma(\mathcal{X})$, where $\Sigma(\mathcal{X})$ is the signature obtained by adding the variables in $\mathcal{X}$ as constants to $\Sigma$, each to its corresponding sort. One can associate a signature to any context-free language (CFG), so that well-formed words in the CFG language are associated corresponding terms in the signature.

A *substitution* is a mapping yielding terms (possibly with variables) for variables. Any substitution $\psi : \mathcal{X} \to T_\Sigma(\mathcal{Y})$ naturally extends to terms, yielding a homonymous mapping $\psi : T_\Sigma(\mathcal{X}) \to T_\Sigma(\mathcal{Y})$. When $\mathcal{X}$ is finite and small, the application of substitution

$\psi : \{x_1, \ldots, x_n\} \rightarrow T_\Sigma(y)$ to term $t$ can be written as $t[\psi(x_1)/x_1, ..., \psi(x_n)/x_n] ::= \psi(t)$. This notation allows one to use substitutions by need, without formally defining them.

Given an ordered set of variables, $\mathcal{W} = \{\square_1, \ldots, \square_n\}$, named *context variables*, or *holes*, a $\mathcal{W}$-context over $\Sigma(\mathcal{X})$ (assume that $\mathcal{X} \cap \mathcal{W} = \emptyset$) is a term $C \in T_\Sigma(\mathcal{X} \cup \mathcal{W})$ which is linear in $W$ (i.e., each hole appears exactly once). The instantiation of a $\mathcal{W}$-context C with an n-tuple $\bar{t} = (t_1, \ldots, t_n)$,written $C[\bar{t}]$ or $C[t_1, \ldots, t_n]$, is the term $C[t_1/\square_1, \ldots, t_n/\square_n]$. One can alternatively regard $\bar{t}$ as a substitution $\bar{t} : \mathcal{W} \rightarrow T_\Sigma(X)$, defined by $\bar{t}(\square_i) = t_i$, in which case $C[\bar{t}] = \bar{t}(C)$. A $\Sigma$-context is a $\mathcal{W}$ context over $\Sigma$ where $\mathcal{W}$ is a singleton.

A *rewrite system* over a term universe $T_\Sigma$ consists of rewrite rules, which can be locally matched and applied at different positions in a $\Sigma$-term to gradually transform it. For simplicity, we only discuss unconditional rewrite rules. A $\Sigma$-rewrite rule is a triple $(X, l, r)$, written $(\forall \mathcal{X})l \rightarrow r$, where $\mathcal{X}$ is a set of variables and $l$ and $r$ are $T_\Sigma(X)$-terms, named the *left-hand-side (lhs)* and the *right-hand-side (rhs)* of the rule, respectively. A rewrite rule $(\forall \mathcal{X})l \rightarrow r$ matches a $\Sigma$-term $t$ using $\mathcal{W}$-context $C$ and substitution $\theta$, iff $t = C[\theta[l]]$. If that is the case, then the term t rewrites to $C[\theta[r]]$. A ($\Sigma$-)rewrite-system $\mathcal{R} = (\Sigma, R)$ is a set $R$ of $\Sigma$-rewrite rules.

*Definition A.7.* (K rule, K-system). *A **K-rule** $\rho : (\forall \mathcal{X})p[\frac{L}{R}]$ over a signature $\Sigma = (S, F)$ is a tuple $(\mathcal{X}, p, L, R)$, where:*

- *$\mathcal{X}$ is an $S$-sorted set, called the **variables** of the rule $\rho$;*
- *$p$ is a $\mathcal{W}$-context over $\Sigma(\mathcal{X})$, called the **rule pattern**, where $\mathcal{W}$ are the holes of $p$; $p$ can be thought of as the "read-only" part of $\rho$;*
- *$L, R : \mathcal{W} \rightarrow T_\Sigma(\mathcal{X})$ associate to each hole in $\mathcal{W}$ the **original term** and its **replacement term**, respectively; $L$, $R$ can be thought of as the "read-write" part of $\rho$.*

*We may write $(\forall \mathcal{X})p[\frac{l_1}{r_1}, \ldots, \frac{l_n}{r_n}]$ instead of $\rho : (\forall \mathcal{X})p[\frac{L}{R}]$ whenever $\mathcal{W} = \{\square_1, \ldots, \square_n\}$ and $L(\square_i) = l_i$ and $R(\square_i) = r_i$; this way, the holes are implicit and need not be mentioned. A set of K rules $\mathcal{K}$ is called a **K system**.*

Recall rule ReadAddress-LocalVariables in Appendix A.3.1. When formalizing the rule as $\rho_r : (\forall \mathcal{X}_r)p_r[\frac{L_r}{R_r}]$, according to Definition A.7, we obtain

$$\rho_r : \langle \frac{\texttt{readAddress(Addr,String2Id("Local"))}}{\texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright \texttt{V}} \_\rangle_k$$
$$\langle \texttt{N}\ \_\rangle_{contractStack} \langle\ \langle \texttt{N} \rangle_{ctId} \langle \_\ \texttt{Addr} \mapsto \texttt{V}\ \_\rangle_{Memory}\ \rangle_{contractInstance}$$

If we want to identify the anonymous variables, the rule could be alternatively written as:

$$\rho_r : \langle \frac{\texttt{readAddress(Addr,String2Id("Local"))}}{\texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright \texttt{V}}\ a\rangle_k$$
$$\langle \texttt{N}\ b\rangle_{contractStack} \langle\ \langle \texttt{N} \rangle_{ctId} \langle c\ \texttt{Addr} \mapsto \texttt{V}\ d\rangle_{Memory}\ \rangle_{contractInstance}$$

Here, we have:

$$\mathcal{X}_r = \{\texttt{Addr}, \texttt{V}, a, N, b, c, d\}$$
$$\mathcal{W}_r = \{\square\}$$
$$p_r = \langle \square\ a\rangle_k \ \langle \texttt{N}\ b\rangle_{contractStack}$$
$$\langle\ \langle \texttt{N} \rangle_{ctId} \langle c\ \texttt{Addr} \mapsto \texttt{V}\ d\rangle_{Memory}\ \rangle_{contractInstance}$$
$$L_r(\square) = \texttt{readAddress(Addr,String2Id("Local"))}$$
$$R_r(\square) = \texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright \texttt{V}$$

RULE AllocateAddress-LocalVariables

$$\left\langle \frac{\begin{array}{c}\texttt{allocateAddress(N:Int, Addr:Int,}\\ \texttt{String2Id("Local"), V:Value)}\end{array}}{\texttt{gasCal(\#allocate,String2Id("Local"))} \curvearrowright \texttt{V}} \cdots \right\rangle_k$$
$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId} \\ \left\langle \frac{\texttt{MEMORY:Map}}{\texttt{MEMORY (Addr |-> V)}} \right\rangle_{Memory} \end{array} \cdots \right\rangle_{contractInstance}$$

Another example is rule AllocateAddress-LocalVariables. The difference from the first example is that there are 2 cells to be subistituted in the configuration. When formalizing the rule as $\rho_w : (\forall \mathcal{X}_w)p_w[\frac{L_w}{R_w}]$, according to Definition A.7, we obtain

$$\rho_w : \langle \frac{\texttt{allocateAddress(N, Addr,String2Id("Local"), V)}}{\texttt{gasCal(\#allocate,String2Id("Local"))} \curvearrowright \texttt{V}} \_\rangle_k$$
$$\langle \langle \texttt{N} \rangle_{ctId} \langle \frac{\texttt{MEMORY}}{\texttt{MEMORY (Addr} \mapsto \texttt{V)}} \_\rangle_{Memory} \rangle_{contractInstance}$$

If we want to identify the anonymous variables, the rule could be alternatively written as:

$$\rho_w : \langle \frac{\texttt{allocateAddress(N, Addr,String2Id("Local"), V)}}{\texttt{gasCal(\#allocate,String2Id("Local"))} \curvearrowright \texttt{V}}\ a\rangle_k$$
$$\langle \langle \texttt{N} \rangle_{ctId} \langle \frac{\texttt{MEMORY}}{\texttt{MEMORY (Addr} \mapsto \texttt{V)}}\ b\rangle_{Memory} \rangle_{contractInstance}$$

Here, we have:

$$\mathcal{X}_w = \{\texttt{N}, \texttt{Addr}, \texttt{V}, a, b, \texttt{MEMORY}\}$$
$$\mathcal{W}_w = \{\square_1, \square_2\}$$
$$p_w = \langle \square_1\ a\rangle_k\ \langle\ \langle \texttt{N} \rangle_{ctId} \langle \square_2\ b\rangle_{Memory}\ \rangle_{contractInstance}$$
$$L_w(\square_1) = \texttt{allocateAddress(N, Addr,String2Id("Local"), V)}$$
$$R_w(\square_1) = \texttt{gasCal(\#allocate,String2Id("Local"))} \curvearrowright \texttt{V}$$
$$L_w(\square_2) = \texttt{MEMORY}$$
$$R_w(\square_2) = \texttt{MEMORY (Addr} \mapsto \texttt{V)}$$

The matching logic [51] can serve as a logic foundation of K system. We give some definitions and properties that we use as follows.

*Definition A.8.* (Pattern). *A matching logic formula, or a **pattern**, is a first-order logic (FOL) formula. Let $\mathcal{T}$ denote the elements of $T_\Sigma(\mathcal{X})$ of a distinguished sort, called configurations. We define satisfaction $(\gamma, \psi) \vDash \varphi$ over configurations $\gamma \in \mathcal{T}$, valuations (can also be seen as substitutions) $\psi : T_\Sigma(\mathcal{X}) \rightarrow T_\Sigma(\mathcal{Y})$ and patterns $\varphi$ as follows (among the FOL constructs, we only show $\exists$):*

- *$(\gamma, \psi) \vDash \exists X \varphi$ iff $(\gamma, \psi') \vDash \varphi$ for some $\psi' : T_\Sigma(\mathcal{X}) \rightarrow T_\Sigma(\mathcal{Y})$ with $\psi'(y) = \psi(y)$ for all $y \in T_\Sigma(\mathcal{X}) \backslash X$.*
- *$(\gamma, \psi) \vDash \pi$ iff $\gamma = \psi(\pi)$, where $\pi \in \mathcal{T}$.*

*We write* $\vDash \varphi$ *when* $(\gamma, \psi) \vDash \varphi$ *for all* $\gamma \in \mathcal{T}$ *and all* $\psi : T_\Sigma(\mathcal{X}) \rightarrow T_\Sigma(\mathcal{Y})$.

*Example*: in rule AllocateAddress-LocalVariables, $p_w$ is an abbreviated form of FOL logic formula:

$\exists \square_1, a, N, \square_2, b. \langle\langle \square_1\ a\rangle_k \ \langle\ \langle \mathsf{N}\rangle_{ctId}\ \langle \square_2\ b\rangle_{Memory}\ \rangle_{contractInstance}\rangle_{Cfg}$

Here, $\langle \ldots \rangle_{Cfg}$ represents a configuration pattern.

Lemma A.9. *(Structural Framing) If* $\sigma \in \Sigma_{s_1,\ldots,s_n,s}$, *and* $\varphi_i, \varphi'_i \in$ Pattern$_{s_i}$ *such that* $\vDash \varphi_i \rightarrow \varphi'_i$ *for all* $i \in 1 \ldots n$, *then* $\vDash \sigma(\varphi_1, \ldots, \varphi_n) \rightarrow \sigma(\varphi'_1, \ldots, \varphi'_n)$.

*Let* $T_{\Sigma,s}(Var)$ *be the set of* $\Sigma$*-terms of sort s, and* Pattern$_s$ *be the s-sorted set of patterns. Therefore, think of* $\Sigma_{s_1,\ldots,s_n,s}$ *as the pattern* $\sigma(x_1 : s_1, \ldots, x_n : s_n)$.

*Example*: assume that

$$p = \langle \texttt{gasCal(\#allocate,String2Id("Local"))}\ \_\rangle_k\ \langle 5\rangle_{ctId}$$

Since $\vDash \langle 5\rangle_{ctId} \rightarrow \top$, where intuitively $\top$ is a pattern that is matched by all elements, we can get $\vDash p \rightarrow p'$ by Lemma A.9, where

$$p' = \langle \texttt{gasCal(\#allocate,String2Id("Local"))}\ \_\rangle_k$$

Lemma A.10. *If* $(\gamma, \psi) \vDash p$, *and* $\vDash p \rightarrow p'$, *then* $(\gamma, \psi) \vDash p'$

Following from the definition of reachability system which is based on matching logic [52], we define the transition system in K systems:

*Definition A.11.* (K-transition system) *The K system* $\mathcal{K}$ *induces a* ***K transition system*** $(\mathcal{T}, \rightarrow_{\mathcal{K}}, \gamma_0)$ *on the configuration model. Here* $\gamma_0 \in \mathcal{T}$ *is the initial configuration.* $\gamma \rightarrow_{\mathcal{K}} \gamma'$ *for* $\gamma, \gamma' \in \mathcal{T}$ *iff there is a substitution* $\psi : T_\Sigma(\mathcal{X}) \rightarrow T_\Sigma(\mathcal{Y})$ *and* $\rho : (\forall \mathcal{X})p[\frac{L}{R}]$ *in* $\mathcal{K}$ *with* $\gamma = \psi(p[L])$ *and* $\gamma' = \psi(p[R])$ *also written as* $\gamma \xrightarrow{\rho}_{\mathcal{K}} \gamma'$.

*Definition A.12.* (K-traces). Given a K transition system $(\mathcal{T}, \rightarrow_{\mathcal{K}}, \gamma_0)$, we define the set of K traces as

$$traces^{\mathcal{K}} = \{[\rho_1, \ldots, \rho_n] \mid \gamma_0 \xrightarrow{\rho_1}_{\mathcal{K}} \gamma_1 \xrightarrow{\rho_2}_{\mathcal{K}} \ldots \xrightarrow{\rho_n}_{\mathcal{K}} \gamma_n\}$$

### *A.3.3* ***Proof of Correctness***.

The framework of the proof is similar to the proof of correctness of translating SAPIC [46], a variant of the applied pi calculus, into language of Tamarin. Here, we prove the soundness of FASVERIF that if there is a trace that has a certain property according to Solidity semantics, there is also a corresponding trace that has the same property in the model of FASVERIF;

*Definition A.13.* ($\mathcal{K}_s$) *The K system of KSolidity is a set of K-rules* $\mathcal{K}_s$, *where* $\mathcal{K}_s = \{$Require, Out-of-gas, $\ldots\}$, *as shown in Figure 11,12,13,14,15,16,17. The K system* $\mathcal{K}_s$ *induces a* ***K transition system*** $(\mathcal{T}, \rightarrow_{\mathcal{K}_s}, \gamma_0)$ *according to Definition A.11.*

Note that we include a subset of KSolidity rules. For instance, rules for `while` statement and arrays are not included. To improve readability, we also remove redundant and similar rules for proving.

*Definition A.14.* ($R_s$) *Given a set of source codes in Solidity language* $S = \{c_1, c_2, \ldots, c_n\}$, *let* $R_s$ *be the independent model, i.e.,* $R_s = \mathcal{R}(c_1) \cup \mathcal{R}(c_2) \cup \ldots \mathcal{R}(c_n) \cup \{$`Init_evars`, `Init_gvars`, `fb_in_call`, `ret_fb`$\}$. *Additionally, the restriction* $\beta_{init}$ *is also added to the system, where*

$$\beta_{init} \equiv All\ \#i\ \#j.\mathsf{Init_E}()@i\&\ \mathsf{Init_E}()@j\ \ {=}{>}\ \ i = j$$

We prove the correctness of independent modeling, because the independent modeling is designed to correspond to the Solidity semantics, while the complementary modeling adds more constraints, e.g., only allowing one transaction in a trace, to achieve higher efficiency in proving. Note that we find that the K-rules corresponding to `ether_succ`, `ether_fail`, `fb_call` are not implemented, so we omit the proof for the 3 rules.

Let a runtime process of Solidity be a state transition system, named *solidity process*, that the state is yielded iff a solidity function is called or a statement in a solidity function is executed, and the next transition is prepared to be executed. A state in a solidity process is represented by a tuple $s = (\gamma, i, c)$, where $\gamma$ is a special configuration in $\mathcal{K}_s$ according to KSolidity, $i$ represents the position of a state on the syntax tree of its function, and $c$ is the sequence of codes (including statements and functions) of the current function to be executed. Assume the KSolidity semantic is correct and relatively complete as claimed, so the KSolidity rules shown in Table 3 should correctly correspond to state transition during a transaction. Specifically, the state $s = (\gamma, i, c)$ in solidity process is yielded into $s = (\gamma', i', c')$ iff the corresponding $\mathcal{K}_s$ starts deriving at $\gamma$ and stops at $\gamma'$, and $c, c'$ can be computed from the k cell in $\gamma, \gamma'$, respectively. The correspondence relation between $c$ and $\gamma$ is shown in the 5th column of Table 3. We also define the transition relation $s \Rightarrow s'$ representing that a state $s$ can be yielded to state $s'$ which is shown on the second column of the table. On the row of ID=9, the notation _ means the function of an external transaction to be executed is not concerned (can be an arbitrary one). Hence the assumption can be formalized as follows:

Proposition A.15. *Given a K transition system* $(\mathcal{T}, \rightarrow_{\mathcal{K}_s}, \gamma_0)$ *that satisfies* $\mathcal{K}_s \vDash \alpha$. *For any* $[\rho_1, \ldots, \rho_n] \in traces^{\mathcal{K}_s}(\gamma_0)$, $\lambda_0 \xrightarrow{\rho_1}_{\mathcal{K}} \lambda_1 \xrightarrow{\rho_2}_{\mathcal{K}} \ldots \xrightarrow{\rho_n}_{\mathcal{K}} \lambda_n$ *corresponds to a sequence of state transitions* $s_0 \Rightarrow s_1 \Rightarrow \cdots \Rightarrow s_m$, *and there exists a strictly monotonically increasing function* $f : \{0, \ldots, m\} \rightarrow \{0 \ldots, n\}$ *such that* $\gamma_j = \lambda_{f(j)}$, *where* $s_j = (\gamma_j, i_j, c_j)$.

In Table 3,the first column represents the type of the current state which can also be easily obtained from $c$, and we denote $T(s)$ as type ID of state $s$. Let function $F_c(i, id)$ be the key facts to be consumed when the state transits from the position $i$ and type $id$. We manually assign the value of $F_c(i, id)$ in the table which is used for latter proving. Here, the key facts are the ones that can be identified and to used build the correspondence relationship in the proof.

*Definition A.16.* (K Successors). *Let* $\rho \in \mathcal{K}_s$, $\mathbf{succ}_k(\rho)$ *represents the set of* ***K successors*** *of* $\rho$ *such that*

$$\mathbf{succ}_k(\rho) = \{\rho_1 \in \mathcal{K}_s \mid \exists \gamma_1, \gamma_2, \gamma_3. \gamma_1 \xrightarrow{\rho}_{\mathcal{K}_s} \gamma_2 \xrightarrow{\rho_1}_{\mathcal{K}_s} \gamma_3\}$$

*Definition A.17.* (Code Successors). *Let* $\rho \in \mathcal{K}_s$, $\mathbf{succ}_c(\rho)$ *represents the set of* ***code successors*** *of* $\rho$:

$$\mathbf{succ}_c(\rho) = \{\rho_1 \in \mathcal{K}_s \mid \exists \psi, \psi_1. \psi(p[R]) = \psi_1(p_1[L])\}$$

*where* $\rho : (\forall \mathcal{X})p[\frac{L}{R}]$ *and* $\rho_1 : (\forall \mathcal{X}_1)p_1[\frac{L_1}{R_1}]$

Lemma A.18. *For each* $\rho \in \mathcal{K}_s$, *we have*

$$\mathbf{succ}_k(\rho) \subseteq \mathbf{succ}_c(\rho)$$

**Table 3: Correspondence relationship of translated rules from Solidity codes.**

| ID | States of solidity process in function $f_c$ | KSolidity | | | | FASVERIF | |
|---|---|---|---|---|---|---|---|
| | | Start or key rule | Command in rule | Correspondence | Valuation $\mathcal{E}$ | Key Rule | $F_c(i, id)$ |
| 1 | $(\gamma, \phi, \texttt{function } f_c(\texttt{d})\{\texttt{stmt}\})$<br>$\Rightarrow (\gamma', 1, \texttt{stmt})$ | FUNCTION-CALL | `functionCall(C:Int;R:Int;`<br>`F:Id;Es:Values;M:Msg)` | `Es` ~ d | $\mathcal{E}(\sigma(seq(\texttt{d}))) = \texttt{d}$ | `recv_ext` | $\{\text{Call}_e, \text{GVar}$<br>$, \text{EVar}\}$ |
| 2 | local state: $(\gamma, i, x(c_x).f_x(p); \texttt{stmt}) \Rightarrow$<br>Recipient state: $(\gamma', 1, \texttt{stmt}_x)$<br>local state: ...$\Rightarrow (\gamma'', i \circ 1 \circ 1, \texttt{stmt})$ | FUNCTION-CALL | `functionCall(C:Int;R:Int;`<br>`F:Id;Es:Values;M:Msg)` | `Es` ~ p | $\mathcal{E}(\sigma(seq(\texttt{p}))) = \texttt{p}$ | `in_call` | $\{\text{Var}_i\}$ |
| 3 | $(\gamma, i, v_1 \leftarrow v_2; \texttt{stmt})$<br>$\Rightarrow (\gamma', i \circ 1, \texttt{stmt})$ | ..., WRITE | `X:Id=V:Value` | `X` $\sim v_1$<br>`V` $\sim v_2$ | $\mathcal{E}(\sigma_v(v_2)) = v_2$ | `var_assign` | $\{\text{Var}_i\}$<br>$\{\text{Var}_i\}$ |
| 4 | $(\gamma, i, \tau\ v_1 \leftarrow v_2; \texttt{stmt})$<br>$\Rightarrow (\gamma', i \circ 1, \texttt{stmt})$ | ..., VAR-DECLARATION | `T:EleType X:Id=V:Value` | `X` $\sim v_1$<br>`V` $\sim v_2$ | $\mathcal{E}(\sigma_v(v_2)) = v_2$ | `var_declare` | $\{\text{Var}_i\}$ |
| 5 | $(\gamma, i, \texttt{if } e_b \texttt{ then stmt}_1 \texttt{ else stmt}_2; \texttt{stmt}_3)$<br>$\Rightarrow (\gamma', i \circ 1, \texttt{stmt}_1; \texttt{stmt}_3)$ | ..., R5 | `if (true) S:Statement`<br>`else S1:Statement` | `true` $\sim e_b$ | $\mathcal{E}(\theta_e(e_b)) = \texttt{true}$ | `if_true` | $\{\text{Var}_i\}$ |
| 6 | $(\gamma, i, \texttt{if } e_b \texttt{ then stmt}_1 \texttt{ else stmt}_2; \texttt{stmt}_3)$<br>$\Rightarrow (\gamma', i \circ 2, \texttt{stmt}_2; \texttt{stmt}_3)$ | ..., R6 | `if (false) S:Statement`<br>`else S1:Statement` | `false` $\sim e_b$ | $\mathcal{E}(\theta_e(e_b)) = \texttt{false}$ | `if_false` | $\{\text{Var}_i\}$ |
| 7 | $(\gamma, i, \texttt{require } e_b; \texttt{stmt})$<br>$\Rightarrow (\gamma', i \circ 1, \texttt{stmt})$ | ..., REQUIRE | `require(true)` | `true` $\sim e_b$ | $\mathcal{E}(\theta_e(e_b)) = \texttt{true}$ | `require_true` | $\{\text{Var}_i\}$ |
| 8 | | | `require(false)` | `false` $\sim e_b$ | $\mathcal{E}(\theta_e(e_b)) = \texttt{false}$ | `require_false` | $\{\text{Var}_i\}$ |
| 9 | $(\gamma, i, \texttt{return i\}); \Rightarrow (\gamma', \phi, \_)$ | ..., RETURN-VALUE | `return E:Value` | `E` $\sim i$ | | `ret_ext` | $\{\text{Var}_i\}$ |
| 10 | $(\gamma, i, \texttt{return i\}); \Rightarrow ...(\textit{caller states})$ | ..., RETURN-VALUE | `return E:Value` | `E` $\sim i$ | | `ret_in` | $\{\text{Var}_i\}$ |
| 11 | $(\gamma, i, c_x.\texttt{transfer}(v_1); \texttt{stmt})$<br>$\Rightarrow (\gamma', i \circ 1, \texttt{stmt})$ | ..., TRANSFER-FUND-BEGIN | `#memberAccess(R:Id,F:Id)`<br>`↷MsgValue:Int` | `MsgValue` $\sim v_1$<br>`R` $\sim c_x$ | $\mathcal{E}(\sigma_v(v_1)) = v_1$ | `transfer_succ` | $\{\text{Var}_i\}$ |
| 12 | | | | | | `transfer_fail` | $\{\text{Var}_i\}$ |
| 13 | $(\gamma, i, c_x.\texttt{send}(v_1); \texttt{stmt})$<br>$\Rightarrow (\gamma', i \circ 1, \texttt{stmt})$ or $\Rightarrow (\gamma', i \circ 2, \texttt{stmt})$ | ..., SEND-FUND-BEGIN | `#memberAccess(R:Id,F:Id)`<br>`↷MsgValue:Int` | `MsgValue` $\sim v_1$<br>`R` $\sim c_x$ | $\mathcal{E}(\sigma_v(v_1)) = v_1$ | `send_succ` | $\{\text{Var}_i\}$ |
| 14 | | | | | | `send_fail` | $\{\text{Var}_i\}$ |

PROOF. We prove the lemma by the following sequence:

(1) Assume $\rho_1 \in \textbf{succ}_k(\rho)$.
(2) By Definition A.16, $\exists \gamma_1, \gamma_2, \gamma_3 . \gamma_1 \xrightarrow{\rho}_{\mathcal{K}_s} \gamma_2 \xrightarrow{\rho_1}_{\mathcal{K}_s} \gamma_3$.
(3) Eliminate $\exists$ in (2), i.e., $\gamma_1 \xrightarrow{\rho}_{\mathcal{K}_s} \gamma_2 \xrightarrow{\rho_1}_{\mathcal{K}_s} \gamma_3$
(4) By (3) and Definition A.11,

$$\exists \psi . \gamma_2 = \psi(p[R])$$

(5) Eliminate $\exists$ in (4), i.e., $\gamma_2 = \psi(p[R])$
(6) By (3) and Definition A.11,

$$\exists \psi_1 . \gamma_2 = \psi_1(p_1[L])$$

(7) By (5), (7),

$$\psi(p[R]) = \psi_1(p_1[L])$$

(8) By (7) and Definition A.17, $\rho_1 \in \textbf{succ}_c(\rho)$.
(9) By (1) and (8), proved.

□

*Definition A.19.* ($\Mapsto$). *Let $\gamma$ be a configuration in $\mathcal{K}_s$, and $p$ be a pattern, we write $\gamma \Mapsto p$ iff there exists a substitution $\psi$, such that $(\gamma, \psi) \vDash p$.*

*Definition A.20.* (2-tuples and sets in $\mathcal{K}_s$.) *In system $\mathcal{K}_s$, let local/global variables be $\mathbb{L}(\gamma)/\mathbb{G}(\gamma)$, i.e., the Memory/ctContext cell of the contract on top of the contractStack of configuration $\gamma$. Let balances be $\mathbb{E}(\gamma)$, i.e., union of Balance cell of all contractInstance cells. Define $\mathbb{L}_b(\gamma)/\mathbb{G}_b(\gamma)$ which collects the local/global variables from cell globalContext, respectively. Define $\mathbb{G}(\gamma, x)/\ \mathbb{G}_g(\gamma, x)$ as global variables of contract $x$ collected by using cell ctContext/globalContext, respectively, in configuration $\gamma$. Define $\mathbb{S}(\gamma)$ as the collection of all contract ID in configuration $\gamma$. Formally,*

$$\mathbb{L}(\gamma) = \{(a, v) \mid \gamma \Mapsto \langle c\ \_\rangle_{contractStack} \langle\ \langle c\rangle_{ctId} \langle\_\ a \mapsto d\ \_\rangle_{ctContext}$$
$$\langle\_\ a \mapsto Local\_\rangle_{ctLocation} \langle\_\ d \mapsto v\ \_\rangle_{Memory}\ \rangle_{contractInstance}\}$$
$$\mathbb{L}_g(\gamma) = \{(a, v) \mid \gamma \Mapsto \langle c\ \_\rangle_{contractStack} \langle\ \langle c\rangle_{ctId} \langle\_\ a \mapsto d\ \_\rangle_{globalContext}$$
$$\langle\_\ a \mapsto Local\_\rangle_{ctLocation} \langle\_\ d \mapsto v\ \_\rangle_{Memory}\ \rangle_{contractInstance}\}$$
$$\mathbb{G}(\gamma) = \{(a, v) \mid \gamma \Mapsto \langle c\ \_\rangle_{contractStack} \langle\ \langle c\rangle_{ctId} \langle\_\ a \mapsto d\ \_\rangle_{ctContext}$$
$$\langle\_\ a \mapsto Global\ \_\rangle_{ctLocation} \langle\_\ d \mapsto v\ \_\rangle_{ctStorage}\ \rangle_{contractInstance}\}$$
$$\mathbb{G}(\gamma, x) = \{(a, v) \mid \gamma \Mapsto \langle\ \langle x\rangle_{ctId} \langle\_\ a \mapsto d\ \_\rangle_{ctContext}$$
$$\langle\_\ a \mapsto Global\ \_\rangle_{ctLocation} \langle\_\ d \mapsto v\ \_\rangle_{ctStorage}\ \rangle_{contractInstance}\}$$
$$\mathbb{G}_g(\gamma) = \{(a, v) \mid \gamma \Mapsto \langle c\ \_\rangle_{contractStack} \langle\ \langle c\rangle_{ctId} \langle\_\ a \mapsto d\ \_\rangle_{globalContext}$$
$$\langle\_\ a \mapsto Global\ \_\rangle_{ctLocation} \langle\_\ d \mapsto v\ \_\rangle_{ctStorage}\ \rangle_{contractInstance}\}$$
$$\mathbb{G}_g(\gamma, x) = \{(a, v) \mid \gamma \Mapsto \langle\ \langle x\rangle_{ctId} \langle\_\ a \mapsto d\ \_\rangle_{globalContext}$$
$$\langle\_\ a \mapsto Global\ \_\rangle_{ctLocation} \langle\_\ d \mapsto v\ \_\rangle_{ctStorage}\ \rangle_{contractInstance}\}$$
$$\mathbb{E}(\gamma) = \{(a, v) \mid \gamma \Mapsto \langle\ \langle a\rangle_{ctId} \langle v\rangle_{Balance}\rangle_{contractInstance}\}$$
$$\mathbb{S}(\gamma) = \{x \mid \gamma \Mapsto \langle\langle x\rangle_{ctId}\rangle_{contractInstance}\}$$

We also define information about the contexts in cell *functionStack* as follows.

*Definition A.21.* (Contexts in function stacks).

$$\text{STACK}(\gamma) = \{s \mid \gamma \Mapsto \langle\_\ \texttt{ListItem(\#state(}s, \_, \_, \_, \_\texttt{))}\ \_\rangle_{functionStack}\}$$

$$\mathbb{L}_s(\gamma, s) = \{(a, v) \mid \exists d.(a \mapsto d\ \in s) \wedge \gamma \Mapsto$$
$$\langle\ \langle\_\ a \mapsto Local\_\rangle_{ctLocation} \langle\_\ d \mapsto v\ \_\rangle_{Memory}\ \rangle_{contractInstance}\}$$

$$\mathbb{L}_A(\gamma) = \{\mathbb{L}_s(\gamma, s) \mid s \in \text{STACK}(\gamma)\}$$

Here, an element in STACK($\gamma$) is a copy of the cell *ctContext*. It represents the addressing information for both global variables and local variables, which is used for context switching. Assume STACK($\gamma$) = $\emptyset$ just when a transaction starts.

We define formula $\alpha$ to set up the initial configuration and rules to filter out the executions that we wish to discard.

- $\alpha_{\rho_1}$: $\rho_1$ starts a transaction. It also indicates that the environment, i.e., configuration, for executing a transaction has been prepared. For example, the contract accounts, i.e., $\langle\rangle_{contractInstance}$, have been created and functions $\langle\rangle_{function}$ have been loaded.
- $\alpha_{\rho_4}$: The transaction is external.
- $\alpha_{\rho_e}$: No error is in the trace.
- $\alpha_{\rho_n}$: $\rho_n$ ends a transaction.
- $\alpha_{init}$: At the start of a transaction, local variables in all contract instances are empty, and the addressing information for global variables stored in *globalContext* and *ctContext* is the same.

$$\alpha \equiv \alpha_{\rho_1} \wedge \alpha_{\rho_4} \wedge \alpha_{\rho_e} \wedge \alpha_{\rho_n} \wedge \alpha_{init}$$

$$\alpha_{\rho_1} \equiv \forall [\rho_1, \ldots, \rho_n] \in traces^{\mathcal{K}_s}(\gamma_0).\rho_1 = \textsc{Function-Call}$$

$$\alpha_{\rho_4} \equiv \forall [\rho_1, \ldots, \rho_n] \in traces^{\mathcal{K}_s}(\gamma_0). \\ \rho_4 = \textsc{Internal-Function-Call}$$

$$\alpha_{\rho_e} \equiv \forall [\rho_1, \ldots, \rho_n] \in traces^{\mathcal{K}_s}(\gamma_0). \\ \forall \rho \in \{\rho_1, \ldots, \rho_n\}.\rho \neq \textsc{Propagate-Exception-True}$$

$$\alpha_{\rho_n} \equiv \forall [\rho_1, \ldots, \rho_n] \in traces^{\mathcal{K}_s}(\gamma_0). \\ \rho_n = \textsc{Propagate-Exception-False}$$

$$\alpha_{init} \equiv \forall x \in \mathbb{S}(\gamma_j).\mathbb{L}_g(\gamma_j, x) = \mathbb{L}(\gamma_j, x) = \emptyset \wedge \mathbb{G}_g(\gamma_j, x) = \mathbb{G}(\gamma_j, x)$$

When a rule in $R_s$ is applied (except `fresh` rule), the variable terms of facts on the right-hand side of the rule are substituted by constant terms or fresh names. Therefore, the produced multiset of facts by the rule has no variables terms, which can correspond to a configuration in $\mathcal{K}_s$, if the concrete terms and fresh names are assigned with values of data type in $\mathcal{K}_s$. Hence, we build the relationship between the terms and the corresponding values in $\mathcal{K}_s$ as follows.

*Definition A.22.* (Valuation $\mathcal{E}$). *Given a term $x$ in a fact, let valuation $\mathcal{E}(x)$ output $x$'s value of data types in $\mathcal{K}_s$. If $x$ is a fresh name or constant term, the value is assigned beforehand; otherwise, if $x$ is a variable term, after the terms in its fact are substituted (i.e., a rule, where the fact is on the right-hand side, is applied), it is evaluated according to the substituted constant terms and fresh names.*

The valuation can be seen as a process of calculating the value of a variable term, based on the assignment of the constant terms and fresh names.

*Definition A.23.* ($\leftrightarrow_v$). *Define vars($f$), gvars($f$), evars($f$) as the sequence of terms representing variables, global variables, and ether balances in fact $f$, respectively. Let $A$ be a 2-tuple derived from a configuration $\gamma$ in $\mathcal{K}_s$. Let $B$ be a sequence of terms obtained from vars, in $R_s$. We write $A \leftrightarrow_v B$, if there exists a bijection between $A$ and $B$ such that whenever $(a, v) \in A$ is mapped to $x \in B$, we have that*

$$v = \mathcal{E}(x)$$

When $A \leftrightarrow_v B$, $(a, v) \in A$ and $x \in B$, we also write $(a, v) \leftrightarrow_v x$ if this bijection maps $(a, v)$ to $x$.

Remark 1. *Note that $\leftrightarrow_v$ has the following properties.*

- *If $A_1 \leftrightarrow_v B_1$ and $A_2 \leftrightarrow_v B_2$, then $A_1 \cup A_2 \leftrightarrow_v B_1 \cup B_2$*
- *If $A_1 \leftrightarrow_v B_1$, and $a \leftrightarrow_v b$ for $a \in A_1$ and $b \in B_1$,then $A_1 \backslash \{a\} \leftrightarrow_v B_1 \backslash \{b\}$*

Recall that the first and fourth parameter in fact GVar and Var represents its contract name, respectively. To differentiate the facts, we denote GVar$^x$ and Var$^x$ as the fact for contract with *ctId* $x$ (corresponding to the first and fourth parameter of the fact, respectively) in $K_s$; we also omit the tag $x$, i.e., denote the facts as GVar and Var, if $x$ is the current running contract in the corresponding $K_s$ (in the following proofs, a state in $\mathcal{K}_s$ always corresponds to a multiset of facts). Though in most cases of the proof the tag is not used, it is useful when analyzing the case related to context switching.

Lemma A.24. *Given function (transaction) code c, define $C_r(c, i)$ as the first parameter of $\mathcal{R}$ when $\mathcal{R}$ is recursively applied on c and i is the second parameter of $\mathcal{R}$; define $C_p(c, i)$ as the third element of a state when the state is at a transition running c and i is the second parameter of the state.*

*Then we have*

$$\vDash \forall i, c.C_r(c, i) = C_p(c, i)$$

Proof. We proceed by induction over the number of state transitions.

If a function $c_0$ is running, and a sequence of state transitions on running $c_0$ is

$$s_0 \Rightarrow s_1 \Rightarrow \ldots$$

where $s_j = (\gamma_j, i_j, c_j)$.

*Base case.* For $s_0$

1. By Table 3, $T(s_0) = 1$ and $i_0 = \emptyset$.
2. By Definition of $\mathcal{R}$, $C_r(c_0, i_0) = C_r(c_0, \emptyset) = c_0$
3. By Definition of $C_p$ and Table 3, $C_p(c_0, \emptyset) = c_0$
4. By (2),(3), $C_r(c_0, \emptyset) = C_p(c_0, \emptyset)$

*Inductive Step.* Assume the invariants hold for $i_j$. We have to show that the lemma holds for the successors of $i_j$.

**Case**: $T(s_j) = 1$,

1. By Table 3, denote $C_p(c_0, i_j) = \texttt{function}\, f(\texttt{d})\{\texttt{stmt}\}$
2. By (1) and inductive hypothesis,
$$C_r(c_0, i_j) = \texttt{function}\, f(\texttt{d})\{\texttt{stmt}\}$$
3. By (1), $i_j = \emptyset$
4. By (1), (3), and Table 3, $C_p(c_0, 1) = \texttt{stmt}$
5. By (2) and Definition of $\mathcal{R}$,
$$C_r(c_0, 1) = \texttt{stmt}$$
6. By (4),(5), $C_r(c_0, 1) = C_p(c_0, 1)$

**Case**: $T(s_j) = 2 \ldots 14$, the proof of the cases is similar to the case $T(s_j) = 1$, and we omit the proof for the cases. □

Theorem A.25. *(Soundness). Let $(\mathcal{T}, \rightarrow_{\mathcal{K}_s}, \gamma_0)$ be a K transition system that satisfies $\mathcal{K}_s \vDash \alpha$. If*

$$\lambda_0 \xrightarrow{\rho_1}_{\mathcal{K}} \lambda_1 \xrightarrow{\rho_2}_{\mathcal{K}} \ldots \xrightarrow{\rho_n}_{\mathcal{K}} \lambda_n$$

*where $[\rho_1, \ldots, \rho_n] \in traces^{\mathcal{K}_s}(\gamma_0)$, and it corresponds to a sequence of state transitions according to Proposition A.15:*

$$s_0 \Rightarrow s_1 \Rightarrow \cdots \Rightarrow s_m$$

*where $s_j = (\gamma_j, i_j, c_j)$, then there are $(r_1, F_1), (r_2, F_2), \ldots, (r_{m'}, F_{m'})$, such that*

$$\emptyset \xrightarrow{r_1}{}_{R_s} F_1 \xrightarrow{r_2}{}_{R_s} \cdots \xrightarrow{r_{m'}}{}_{R_s} F_{m'}$$

*and there exists a valuation $\mathcal{E}$ and a monotonic, strictly increasing function $g : \{0, \ldots, m\} \rightarrow \{0, \ldots, m'\}$ such that $g(m) = m'$ and for all $j \in \{0, \ldots, m\}$*

- *(a)* $F_c(i_j, T(s_j)) \subseteq names(F_{g(j)})$
- *(b)* $T(s_j) = 1 \rightarrow \exists V \in^{\#} F_{g(j)}.$
  $\mathbb{G}(\gamma_j) \leftrightarrow_v gvars(V) \wedge names(\{V\}) = \{\mathsf{GVar}\}$
- *(c)* $T(s_j) > 1 \rightarrow \exists i.\exists V \in^{\#} F_{g(j)}.$
  $\mathbb{G}(\gamma_j) \leftrightarrow_v gvars(V) \wedge names(\{V\}) = \{\mathsf{Var}_i\}$
- *(d)* $T(s_j) = 1 \rightarrow \exists V \in^{\#} F_{g(j)}.$
  $\mathbb{E}(\gamma_j) \leftrightarrow_v evars(V) \wedge names(\{V\}) = \{\mathsf{EVar}\}$
- *(e)* $T(s_j) > 1 \rightarrow \exists i.\exists V \in^{\#} F_{g(j)}.$
  $\mathbb{E}(\gamma_j) \leftrightarrow_v evars(V) \wedge names(\{V\}) = \{\mathsf{Var}_i\}$
- *(f)* $T(s_j) = 1 \rightarrow \mathbb{L}(\gamma_j) = \emptyset$
- *(g)* $T(s_j) > 1 \rightarrow \exists i.\exists V \in^{\#} F_{g(j)}.$
  $\mathbb{L}(\gamma_j) \leftrightarrow_v vars(V)\backslash gvars(V)) \wedge names(\{V\}) = \{\mathsf{Var}_i\}$
- *(h)* $\forall x \in \mathbb{S}(\gamma_j).\mathbb{G}_g(\gamma, x) = \mathbb{G}(\gamma, x)$
- *(i)* $\forall x \in \mathbb{S}(\gamma_j).\mathbb{L}_g(\gamma_j, x) = \emptyset$
- *(j)* $\forall l \in \mathbb{L}_A(\gamma_j).\exists i.\exists V \in^{\#} F_{g(j)}.$
  $l \leftrightarrow_v vars(V)\backslash gvars(V)) \wedge names(\{V\}) = \{\mathsf{Var}_i\}$
- *(k)* $\forall x \in \mathbb{S}(\gamma_j).\exists V \in^{\#} F_{g(j)}.\mathbb{G}(\gamma_j, x) \leftrightarrow_v gvars(V).$

Proof. We proceed by induction over the number of state transitions $k$. Denote $\iota(\gamma)$ as the contract ID of on top of cell *contractStack* of $\gamma$. Note that to achieve readability of the proof (besides strictness), we omit the details of a derivation if the derivation is similar to one that has been illustrated for another conclusion.

*Base case.* For $k = 0$, we let $g(0) = t + 2$, if there are $t$ contracts translated by $\mathcal{R}$. Choose a function of a contract to run and let $F_1, F_2, F_3$ be the multiset obtained by using the rule `init_evars`, `init_gvars`, `ext_call` in order respectively.

(1) By Lemma A.3 and Lemma A.4

$$\begin{aligned} F_1 =& \{\mathsf{Evar}(e(\omega_0))\} \\ F_2 =& F_1 \cup^{\#} \{\mathsf{Gvar}([\![\omega_0[1]]\!] \cdot g(\omega_0)\backslash e(\omega_0))\} \\ &= \{\mathsf{Evar}(e(\omega_0)), \mathsf{Gvar}([\![\omega_0[1]]\!] \cdot g(\omega_0)\backslash e(\omega_0))\} \\ F_3 =& F_2 \cup^{\#} \{\mathsf{Call_e}([\![\omega_0[1], \sigma_a(f), \sigma_v(c_b)]\!] \cdot \sigma(seq(\mathsf{d})))\} \\ =& \{\mathsf{Evar}(e(\omega_0)), \mathsf{Gvar}([\![\omega_0[1]]\!] \cdot g(\omega_0)\backslash e(\omega_0)), \\ & \mathsf{Call_e}([\![\omega_0[1], \sigma_a(f), \sigma_v(c_b)]\!] \cdot \sigma(seq(\mathsf{d})))\} \end{aligned}$$

Similarly, rule `init_gvars` is applied to other contracts for $t - 1$ times. For instance, for contract $x$, facts $\mathsf{GVar}^x$ is added to the current multiset of facts. Finally, $F_{t+2}$ is generated.

(2) By Definition of *names*

$$\{\mathsf{EVar}, \mathsf{GVar}, \mathsf{Call_e}\} = names(F_3) \subseteq names(F_{t+2})$$

(3) By Table 3, $i_0 = \emptyset, T(s_0) = 1$,

$$F_c(i_0, T(s_0)) = \{\mathsf{EVar}, \mathsf{GVar}, \mathsf{Call_e}\}$$

(4) By (2),(3), $F_c(i_0, T(s_0)) \subseteq names(F_{t+2})$, *condition **(a)*** proved.

(5) By (3), $T(s_0) = 1$, so *condition **(c)**, **(e)**, **(g)*** hold trivially.

(6) By $\alpha_{init}$, *Condition* (***b***), (***d***), (***f***), (***j***), (***k***) hold by interpreting $\mathcal{E}$ such that

- $\mathbb{G}(\gamma_0) \leftrightarrow_v vars(\mathsf{Gvar})$
- $\mathbb{E}(\gamma_0) \leftrightarrow_v vars(\mathsf{Evar})$
- $\mathbb{L}(\gamma_0) \leftrightarrow_v \emptyset$
- $\forall x \in \mathbb{S}(\gamma_0).\mathbb{L}(\gamma_0, x) = \emptyset \leftrightarrow_v \emptyset$
- $\forall x \in \mathbb{S}(\gamma_0).\mathbb{G}(\gamma_0, x) \leftrightarrow_v gvars(\mathsf{Gvar}^x)$

(7) By $\alpha_{init}$, *condition (**h**), (**i**)* holds.

*Inductive step.* Assume the invariant holds for $k \geq 0$. We have to show that the lemma holds for $k + 1$ transitions.

$$s_0 \Rightarrow s_1 \Rightarrow \cdots \Rightarrow s_k \Rightarrow s_{k+1}$$

(A) By induction hypothesis, we have that there exists a monotonic increasing function $g$ and an execution

$$\emptyset \xrightarrow{r_1}{}_{R_s} F_1 \xrightarrow{r_2}{}_{R_s} \cdots \xrightarrow{r_{k'}}{}_{R_s} F_{k'}$$

such that the conditions hold and $g(k) = k'$.

(B) By Proposition A.15, there exists a strictly monotonically increasing function $f$ such that $\gamma_j = \lambda_{f(j)}$ and $f(0) = 0$.

(C) By eliminating $\exists$ on $g$ and $f$, we use the function $g_0$ and $f_0$, respectively.

(D) By (B), $\gamma_k = \lambda_{f_0(k)}$

(E) By (D) and Definition A.12, in $traces^{\mathcal{K}_s}(\gamma_0)$ the segment of traces from $s_k$ to $s_{k+1}$ is

$$[\rho_{f_0(k)+1}, \ldots, \rho_{f_0(k+1)}]$$

and

$$\lambda_{f_0(k)} \xrightarrow{\rho_{f_0(k)+1}}{}_{\mathcal{K}} \lambda_{f_0(k)+1} \xrightarrow{\rho_{f_0(k)+2}}{}_{\mathcal{K}} \cdots \xrightarrow{\rho_{f_0(k+1)}}{}_{\mathcal{K}} \lambda_{f_0(k+1)}$$

We now proceed by case distinction over the type of transitions from $s_k$ to $s_{k+1}$. We will extend the previous executions by a number of steps, say *step*, from $F_{k'}$ to some $F_{k'+step}$, and prove that the conditions hold for $k + 1$, and a function $g$, defined as follows:

$$g(j) := \begin{cases} g_0(j) & \text{if } i \in \{0, ..., k\} \\ g_0(k) + step & \text{if } i = k + 1 \end{cases}$$

**Case:** $T(s_k) = 1$.

(1) By Definition A.17 and Lemma A.18, there is only one possible sub-trace $[\rho_{f_0(k)+1}, \ldots, \rho_{f_0(k)+t+11}]$ for the state transition in (E) as follows:

- $\rho_{f_0(k)+1}$ = Function-Call
- $\rho_{f_0(k)+2}$ = Switch-Context
- $\rho_{f_0(k)+3}$ = Create-Transaction
- $\rho_{f_0(k)+4}$ = Internal-Function-Call
- $\rho_{f_0(k)+5}$ = Save-Cur-Context
- $\rho_{f_0(k)+6}$ = Call
- $\rho_{f_0(k)+7}$ = Init-Fun-Params
- $\rho_{f_0(k)+8}$ = Bind-Params
- $\rho_{f_0(k)+9}$ = Bind-Params
- ...
- $\rho_{f_0(k)+t+9}$ = Bind-Params-End
- $\rho_{f_0(k)+t+10}$ = processFunQuantifiers
- $\rho_{f_0(k)+t+11}$ = Call-Function-Body
- (———End of state transition———)
- $\rho_{f_0(k)+t+12}$ = Function-Body
- ...
- (———End of statements in function———)
- $\rho_{f_0(k)+t+u+12}$ = Update-Cur-Context

- $\rho_{f_0(k)+t+u+13}$ = Return-Context
- $\rho_{f_0(k)+t+u+14}$ = Clear-Recipient-Context
- $\rho_{f_0(k)+t+u+15}$ = Clear-Caller-Context
- $\rho_{f_0(k)+t+u+16}$ = Propagate-Exception-False
- (———End of function———)

Note that by $\alpha_{\rho_e}$, we exclude the possible trace such that $\rho_{f_0(k)+t+u+16}$ = Propagate-Exception-True, which means there are exceptions and the transaction is reverted.

(2) By Proposition A.15, in (E), the trace $[\rho_{f_0(k)+1}, \dots, \rho_{f_0(k)+t+11}]$ in (1) corresponds to the state transition from $s_k$ to $s_{k+1}$. Here, $t$ is assumed to be the number of parameters in the function. $u$ represents the number of steps on executing the statements and nested functions.

(3) By $T(s_k)$, $i_k = \emptyset$, and let $c_k = \texttt{function } f_c(\texttt{d})\{\texttt{stmt}\}$

(4) By (3) and Lemma A.24,

$$C_r(c_k, i_k) = C_p(c_k, i_k) = c_k = \texttt{function } f_c(\texttt{d})\{\texttt{stmt}\}$$

(5) By (4), Lemma A.24 and Definition of $\mathcal{R}$, rule `recv_ext` is generated.

(6) By inductive hypothesis of condition (a),

$$\{\mathsf{Call}_e, \mathsf{GVar}, \mathsf{EVar}\} = F_c(\emptyset, 1) = F_c(i_k, T(s_k)) \subseteq \mathit{names}(F_{g(k)})$$

(7) By (5), (6), construct $F_{g(k)+1}$ by applying rule `recv_ext`, and by Definition of *name*: (It is similar to condition (a).(1)(2) of Base case, and we omit the details.)

$$\{\mathtt{Var}_1\} \subseteq \mathit{names}(F_{g(k)+1})$$

(8) By (4) and Table 3,

$$c_{k+1} = \mathtt{stmt}, i_{k+1} = 1$$

(9) By (8) and Table 3, $T(s_{k+1}) > 1$

(10) By (8), (9) and Table 3, $F_c(i_{k+1}, T(s_{k+1})) = \{\mathtt{Var}_1\}$.

(11) Let *step* = 1, i.e., $g(k+1) = g(k) + 1$

(12) By (7), (10), (11)

$$F_c(i_{k+1}, T(s_{k+1})) = \{\mathtt{Var}_1\} \subseteq \mathit{names}(F_{g(k+1)})$$

*Condition **(a)** proved.*

(13) By (B), $\mathbb{G}(\lambda_{f_0(k)}) = \mathbb{G}(\gamma_k)$

(14) By $T(s_k)$, $T(s_k) = 1$

(15) By (14) and inductive hypothesis, $\mathbb{G}(\gamma_k) \leftrightarrow_v \mathit{gvars}(\mathsf{GVar})$

(16) By (5), $\mathit{gvars}(\mathsf{GVar}) = \mathit{gvars}(\mathtt{Var}_1)$

(17) By assumption of $f$, $\lambda_{f_0(k)+t+11} = \gamma_{k+1}$

(18) By (2), inductive hypothesis (h), and Definition A.7,A.20

$$\mathbb{G}(\lambda_{f_0(k)+t+11}) = \cdots = \mathbb{G}(\lambda_{f_0(k)+2}) = \mathbb{G}_g(\lambda_{f_0(k)}) = \mathbb{G}(\lambda_{f_0(k)})$$

(19) By (15), (16), (18), $\mathbb{G}(\lambda_{f_0(k)+t+11}) \leftrightarrow_v \mathit{gvars}(\mathtt{Var}_1)$

(20) By (17), (19), $\mathbb{G}(\gamma_{k+1}) \leftrightarrow_v \mathit{gvars}(\mathtt{Var}_1)$

(21) By (9), (20), *condition **(b)**, **(c)** hold.*

(22) Similar to (15), $\mathbb{E}(\gamma_k) \leftrightarrow_v \mathit{evars}(\mathsf{EVar})$

(23) Similar to (16), $\mathit{evars}(\mathsf{EVar}) = \mathit{evars}(\mathtt{Var}_1)$

(24) By Definition A.7,A.20,

$$\mathbb{E}(\lambda_{f_0(k)+t+11}) = \cdots = \mathbb{E}(\lambda_{f_0(k)})$$

(25) Similar to (21), *condition **(d)**,**(e)** hold*

(26) By Definition A.7, and inductive hypothesis (i)

$$\mathbb{L}(\lambda_{f_0(k)+7}) = \cdots = \mathbb{L}(\lambda_{f_0(k)+2}) = \mathbb{L}_g(\lambda_{f_0(k)}) = \emptyset$$

(27) By Definition A.22, let valuation $\mathcal{E}(\sigma(\mathit{seq}(\mathtt{d}))) = \mathtt{d}$

(28) By (26), (27) and Definition A.22, let

$$\mathbb{L}(\lambda_{f_0(k)+t+9}) \leftrightarrow_v \sigma(\mathit{seq}(\mathtt{d}))$$

(29) By Definition A.7, $\mathbb{L}(\lambda_{f_0(k)+t+11}) = \cdots = \mathbb{L}(\lambda_{f_0(k)+t+9})$

(30) By (7), generated rule `recv_ext` satisfies:

$$\mathit{vars}(\mathtt{Var}_1) \backslash \mathit{gvars}(\mathtt{Var}_1) = \sigma(\mathit{seq}(\mathtt{d}))$$

(31) Similar to (21), by (28), (29), (30), *condition **(f)** **(g)** hold.*

(32) By Definition A.7, *condition **(h)**, **(i)** hold* trivially.

(33) By Definition A.21 and A.7,

$$\mathbb{L}_A(\lambda_{f_0(k)+t+11}) = \cdots = \mathbb{L}(\lambda_{f_0(k)+2}) = \mathbb{L}_A(\lambda_{f_0(k)}) \cup \{\emptyset\} = \{\emptyset\}$$

(34) By (2), (17), (33), let $\emptyset \leftrightarrow_v \emptyset$, *condition (j)* holds.

(35) By (12), (20) and inductive hypothesis (k), *condition **(k)*** hold.

**Case:** $T(s_k) = 2$.

(1) Similar to (1)(2) in case $T(s_k) = 1$, (abbreviated as $\mathit{Case}_1.1 \sim 2$), the sub-trace for the state transition in (E) is as follows:

- $\rho_{f_0(k)+1}$ = Function-Call
- $\rho_{f_0(k)+2}$ = Switch-Context
- $\rho_{f_0(k)+3}$ = Create-Transaction
- $\rho_{f_0(k)+4}$ = Nested-Function-Call
- $\rho_{f_0(k)+5}$ = Call
- $\rho_{f_0(k)+6}$ = Init-Fun-Params
- $\rho_{f_0(k)+7}$ = Bind-Params
- $\rho_{f_0(k)+8}$ = Bind-Params
- ...
- $\rho_{f_0(k)+t+8}$ = Bind-Params-End
- $\rho_{f_0(k)+t+9}$ = processFunQuantifiers
- $\rho_{f_0(k)+t+10}$ = Call-Function-Body
- (———End of state transition———)
- $\rho_{f_0(k)+t+11}$ = Function-Body
- ...
- (———End of statements in function———)
- $\rho_{f_0(k)+t+u+11}$ = Return-Context
- $\rho_{f_0(k)+t+u+12}$ = Clear-Recipient-Context
- $\rho_{f_0(k)+t+u+13}$ = Clear-Caller-Context
- $\rho_{f_0(k)+t+u+14}$ = Propagate-Exception-False
- (———End of function———)

(2) Similar to $\mathit{Case}_1.2$, $[\rho_{f_0(k)+1}, \dots, \rho_{f_0(k)+t+10}]$ in (1) corresponds to the state transition from $s_k$ to $s_{k+1}$. Here, $t$ represents the number of parameters in the function. $u$ represents the number of steps on executing the statements.

(3) Similar to $\mathit{Case}_1.3$, let $c_k = x(c_x).f_x(p); \mathtt{stmt}$

(4) Let the caller and recipient ID be $id_c$, $id_r$, respectively.

(5) Similar to $\mathit{Case}_1.4$, $C_r(c_k, i_k) = x(c_x).f_x(p); \mathtt{stmt}$

(6) Similar to $\mathit{Case}_1.5$, rule `in_call` is generated.

(7) By Definition of $\mathcal{R}$, the first rule for $f_x(p)$ is generated.

(8) Similar to $\mathit{Case}_1.6$, $\{\mathsf{Var}_i^{id_c}\} \subseteq \mathit{names}(F_{g(k)})$

(9) Similar to $\mathit{Case}_1.7$, $\{\mathsf{Call}_{in}, \mathsf{Var}_{i\circ 1}^{id_c}, \mathsf{GVar}^{id_c}\} \subseteq \mathit{names}(F_{g(k)+1})$

(10) Similar to $\mathit{Case}_1.7$, by (7), $\{\mathsf{Var}_1^{id_r}\} \subseteq \mathit{names}(F_{g(k)+2})$

(11) Let *step*=1, i.e., $g(k+1) = g(k) + 2$

(12) Similar to $\mathit{Case}_1.12$, *Condition **(a)*** proved.

(13) By (B), $\gamma_k = \lambda_{f_0(k)}$

(14) Similar to $\mathit{Case}_1.15$, $\mathbb{G}(\gamma_k, id_c) \leftrightarrow_v \mathit{gvars}(\mathsf{Var}_i^{id_c})$

(15) By inductive hypothesis (k), we have $V$, such that

$$\mathbb{G}(\gamma_k, id_r) \leftrightarrow_v gvars(V)$$

(16) By assumption of $f$, $\lambda_{f_0(k)+t+10} = \gamma_{k+1}$

(17) Similar to $Case_1$.18,

$$\mathbb{G}(\lambda_{f_0(k)+t+10}) = \cdots = \mathbb{G}(\lambda_{f_0(k)+2}) = \mathbb{G}_g(\gamma_k, id_r) = \mathbb{G}(\gamma_k, id_r)$$

(18) For caller $id_c$, by (5), (6), $\mathbb{G}(\gamma_{k+1}, id_c) \leftrightarrow_v gvars(\mathsf{GVar}^{id_c})$

(19) For recipient $id_r$, by (10), (15), (17),

$$\mathbb{G}(\gamma_{k+1}, id_r) = \mathbb{G}(\gamma_{k+1}) = \mathbb{G}(\lambda_{f_0(k)+t+10})$$

$$\mathbb{G}(\gamma_{k+1}, id_r) \leftrightarrow_v \mathsf{Var}_1^{id_r}$$

(20) Similar to $Case_1$.21, by (19), *condition **(b), (c)** hold.*

(21) Similar to $Case_1$.25, *condition **(d)**,**(e)** hold.*

(22) Similar to $Case_1$.31, *condition **(f)** **(g)** hold.*

(23) By Definition A.7, *condition **(h)**, **(i)** hold* trivially.

(24) Similar to $Case_1$.33,

$$\mathbb{L}_A(\lambda_{f_0(k)+t+10}) = \cdots = \mathbb{L}(\lambda_{f_0(k)+2}) = \mathbb{L}_A(\lambda_{f_0(k)}) \cup \{\mathbb{L}(\gamma_k)\}$$

(25) Similar to $Case_1$.20, by (8),

$$\mathbb{L}(\gamma_k) \leftrightarrow_v vars(\mathsf{Var}_i^{id_c}) \backslash gvars(\mathsf{Var}_i^{id_c})$$

(26) By (5), (6), $\mathsf{Var}_{i\circ 1}^{id_c} \in names(F_{g(k+1)})$ and

$$vars(\mathsf{Var}_i^{id_c}) \backslash gvars(\mathsf{Var}_i^{id_c}) = vars(\mathsf{Var}_{i\circ 1}^{id_c}) \backslash gvars(\mathsf{Var}_{i\circ 1}^{id_c})$$

(27) By (16), (24), (25), (26), *condition **(j)*** holds.

(28) Similar to $Case_1$.35, by (10), (18), (19), *condition **(k)*** holds.

**Case:** $T(s_k) = 3$.

(1) Similar to (1), (2) in case $T(s_k) = 1$, the possible sub-traces for the state transition in (E) are as follows:

Sub-trace 1:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement-Main-Contract
- $\rho_{f_0(k)+3}$ = Write
- $\rho_{f_0(k)+4}$ = WriteAddress-GlobalVariables
- $\rho_{f_0(k)+5}$ = Gas-Cal
- (———End of state transition———)
- $\rho_{f_0(k)+6}$ = Function-Body

Sub-trace 2:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement-Main-Contract
- $\rho_{f_0(k)+3}$ = Write
- $\rho_{f_0(k)+4}$ = WriteAddress-LocalVariables
- $\rho_{f_0(k)+5}$ = Gas-Cal
- (———End of state transition———)
- $\rho_{f_0(k)+6}$ = Function-Body

Sub-trace 3:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement
- $\rho_{f_0(k)+3}$ = Write
- $\rho_{f_0(k)+4}$ = WriteAddress-GlobalVariables
- $\rho_{f_0(k)+5}$ = Gas-Cal
- (———End of state transition———)
- $\rho_{f_0(k)+6}$ = Function-Body

Sub-trace 4:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement
- $\rho_{f_0(k)+3}$ = Write
- $\rho_{f_0(k)+4}$ = WriteAddress-LocalVariables
- $\rho_{f_0(k)+5}$ = Gas-Cal
- (———End of state transition———)
- $\rho_{f_0(k)+6}$ = Function-Body

The difference between Exe-Statement-Main-Contract and Exe-Statement is that the first one occurs in a function of an external call and the second one is the function of an internal call, but the behaviors are the same. The difference between WriteAddress-GlobalVariables and WriteAddress-LocalVariables is that the first one writes the value to the cell *ctContext* which stores global variables and second one writes the value to the cell *Memory* which stores local variables. Therefore, the proofs for the above sub-traces are similar, and we choose *sub-trace 1* for proving as follows.

(2) Similar to $Case_1$.2, $[\rho_{f_0(k)+1}, \dots, \rho_{f_0(k)+5}]$ in (1) corresponds to the state transition from $s_k$ to $s_{k+1}$.

(3) Similar to $Case_1$.3, let $c_k = v_1 \leftarrow v_2; \mathtt{stmt}$ (For readability, the value of left-hand of the name is assumed to be the ID of the variable in $\mathcal{K}_s$)

(4) Similar to $Case_1$.4, $C_r(c_k, i_k) = v_1 \leftarrow v_2; \mathtt{stmt}$

(5) Similar to $Case_1$.5, rule `var_assign` is generated.

(6) Similar to $Case_1$.6, $\{\mathsf{Var}_i\} \subseteq names(F_{g(k)})$

(7) Similar to $Case_1$.10, $F_c(i_{k+1}, T(s_{k+1})) = \{\mathtt{Var}_{\mathtt{i}\circ\mathtt{1}}\}$

(8) Let *step*=1, i.e., $g(k+1) = g(k) + 1$

(9) Similar to $Case_1$.12, *Condition **(a)*** proved.

(10) By (B), $\gamma_k = \lambda_{f_0(k)}$

(11) Similar to $Case_1$.15, $\mathbb{G}(\gamma_k) \leftrightarrow_v gvars(\mathsf{Var}_i)$

(12) By assumption of $f$, $\lambda_{f_0(k)+5} = \gamma_{k+1}$

(13) Similar to $Case_1$.18,

$$\mathbb{G}_{\lambda_{f_0(k)+5}} = \mathbb{G}_{\lambda_{f_0(k)+4}}$$

(14) Compute value $x$ which satisfies

$$\lambda_{f_0(k)} \Rrightarrow \langle \iota(\lambda_{f_0(k)})\ \_\rangle_{contractStack}\ \langle\langle \iota(\lambda_{f_0(k)})\rangle_{ctId}$$
$$\langle v_1 \mapsto d\rangle_{ctContext}\ \ \langle\_\ d \mapsto x\ \_\rangle_{ctStorage}\rangle_{contractInstance}$$

(15) By (14), Definition A.7,

$$\mathbb{G}_{\lambda_{f_0(k)+5}} = \mathbb{G}_{\lambda_{f_0(k)}} \cup \{(v_1, v_2)\} \backslash \{(v_1, x)\}$$

(16) By (4), (11), (15),

$$(v_1, x) \leftrightarrow_v \sigma_v(v_1)$$

(17) By (4), (7)

$$gvars(\{\mathsf{Var}_{i\circ 1}\}) = gvars(\{\mathsf{Var}_i\}) \cup \{\sigma_v(v_1))\} \backslash \{\sigma_v(v_1))\}$$

(18) Since $v_2 = \mathcal{E}(\sigma_v(v_2))$, build mapping

$$(v_1, v_2) \leftrightarrow_v \sigma_v(v_2)$$

(19) By (10), (11), (15), (16), (17), (18), and Remark 1,

$$\mathbb{G}(\gamma_{k+1}) \leftrightarrow_v gvars(\mathsf{Var}_{i\circ 1})$$

(20) Similar to $Case_1$.10, *condition **(b)**,**(c)** hold.*

(21) By Definition A.7, *condition **(d)**, **(e)**, **(f)**, **(g)**, **(h)**, **(i)**, **(j)**, **(k)** hold* trivially.

**Case:** $T(s_k) = 4$.

(1) Similar to (1), (2) in case $T(s_k) = 1$, the possible sub-traces for the state transition in (E) are as follows:

Sub-trace 1:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement-Main-Contract
- $\rho_{f_0(k)+3}$ = Var-Declaration
- $\rho_{f_0(k)+4}$ = Gas-Cal
- (———End of state transition———)
- $\rho_{f_0(k)+5}$ = Function-Body

Sub-trace 2:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement
- $\rho_{f_0(k)+3}$ = Var-Declaration
- $\rho_{f_0(k)+4}$ = Gas-Cal
- (———End of state transition———)
- $\rho_{f_0(k)+5}$ = Function-Body

Similar to *Case*.2, we choose *sub-trace 1* for proving as follows.

(2) Similar to $Case_1$.2, $[\rho_{f_0(k)+1}, \dots, \rho_{f_0(k)+4}]$ in (1) corresponds to the state transition from $s_k$ to $s_{k+1}$.

(3) Similar to $Case_1$.3, let $c_k = \tau\ v_1 \leftarrow v_2; \texttt{stmt}$

(4) Similar to $Case_1$.4, $C_r(c_k, i_k) = \tau\ v_1 \leftarrow v_2; \texttt{stmt}$

(5) Similar to $Case_1$.5, rule `var_declare` is generated.

(6) Similar to $Case_1$.6, $\{\mathsf{Var}_i\} \subseteq names(F_{g(k)})$

(7) Similar to $Case_1$.10, $F_c(i_{k+1}, T(s_{k+1})) = \{\texttt{Var}_{\texttt{i}\circ\texttt{1}}\}$

(8) Let *step*=1, i.e., $g(k+1) = g(k) + 1$

(9) Similar to $Case_1$.12, *Condition **(a)*** proved.

(10) By (B), $\gamma_k = \lambda_{f_0(k)}$

(11) Similar to $Case_1$.15,

$$\mathbb{L}(\gamma_k) \leftrightarrow_v vars(\texttt{Var}_\texttt{i}) \backslash gvars(\texttt{Var}_\texttt{i})$$

(12) By assumption of $f$, $\lambda_{f_0(k)+4} = \gamma_{k+1}$

(13) By Definition A.7

$$\mathbb{L}_{\lambda_{f_0(k)+4}} = \mathbb{L}_{\lambda_{f_0(k)}} \cup \{(v_1, v_2)\}$$

(14) By (5), generated rule `var_declare` satisfies:

$$vars(\texttt{Var}_{\texttt{i}\circ\texttt{1}}) \backslash gvars(\texttt{Var}_{\texttt{i}\circ\texttt{1}}) =$$
$$vars(\texttt{Var}_\texttt{i}) \backslash gvars(\texttt{Var}_\texttt{i}) \cup [\![\sigma_v(v_2)]\!]$$

(15) By (13),(14), since Since $v_2 = \mathcal{E}(\sigma_v(v_2))$, build mapping

$$(v_1, v_2) \leftrightarrow_v \sigma_v(v_2)$$

(16) By (10), (11), (12), (13), (14),(15), and Remark 1,

$$\mathbb{L}(\gamma_{k+1}) \leftrightarrow_v vars(\texttt{Var}_{\texttt{i+1}}) \backslash gvars(\texttt{Var}_{\texttt{i+1}})$$

(17) Similar to $Case_1$.10, *condition **(f)**, **(g)** hold.*

(18) By Definition A.7, *condition **(b)**, **(c)**, **(d)**, **(e)**, **(h)**, **(i)**, **(j)**, **(k)** hold* trivially.

**Case:** $T(s_k) = 5$.

(1) Similar to (1), (2) in case $T(s_k) = 1$, the possible sub-traces for the state transition in (E) are as follows:

Sub-trace 1:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement-Main-Contract
- $\rho_{f_0(k)+3}$ = R5
- (———End of state transition———)
- $\rho_{f_0(k)+4}$ = Exe-Statement-Main-Contract
- ...
- $\rho_{f_0(k)+i+4}$ = Exe-Statement-Main-Contract
- (———End of branch———)
- $\rho_{f_0(k)+i+4}$ = Function-Body

Sub-trace 2:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement
- $\rho_{f_0(k)+3}$ = R5
- (———End of state transition———)
- $\rho_{f_0(k)+4}$ = Exe-Statement
- ...
- $\rho_{f_0(k)+i+4}$ = Exe-Statement
- (———End of branch———)
- $\rho_{f_0(k)+i+4}$ = Function-Body

Similar to *Case*.2, we choose *sub-trace 1* for proving as follows.

(2) Similar to $Case_1$.2, $[\rho_{f_0(k)+1}, \dots, \rho_{f_0(k)+3}]$ in (1) corresponds to the state transition from $s_k$ to $s_{k+1}$.

(3) Similar to $Case_1$.3, let

$$c_k = \texttt{if}\ e_b\ \texttt{then}\ \texttt{stmt}_1\ \texttt{else}\ \texttt{stmt}_2; \texttt{stmt}_3$$

(4) Similar to $Case_1$.4,

$$C_r(c_k, i_k) = \texttt{if}\ e_b\ \texttt{then}\ \texttt{stmt}_1\ \texttt{else}\ \texttt{stmt}_2; \texttt{stmt}_3$$

(5) Similar to $Case_1$.5, since $e_b = \texttt{true}$, we get $\mathcal{E}(\theta_e(e_b)) = \mathcal{E}(\sigma_v(e_b)) = \texttt{true}$ and rule `if_true` can be applied.

(6) Similar to $Case_1$.6, $\{\mathsf{Var}_i\} \subseteq names(F_{g(k)})$

(7) Similar to $Case_1$.10, $F_c(i_{k+1}, T(s_{k+1})) = \{\texttt{Var}_{\texttt{i}\circ\texttt{1}}\}$

(8) Let *step*=1, i.e., $g(k+1) = g(k) + 1$

(9) Similar to $Case_1$.12, *Condition **(a)*** proved.

(10) By Definition A.7, *condition **(b)**, **(c)**, **(d)**, **(e)**, **(f)**, **(g)**, **(h)**, **(i)**, **(j)**, **(k)** hold* trivially.

**Case:** $T(s_k) = 6$. The proof of the case is similar to $Case_5$, and we omit the proof of this case.

**Case:** $T(s_k) = 7$.

(1) Similar to (1), (2) in case $T(s_k) = 1$, the possible sub-traces for the state transition in (E) are as follows:

Sub-trace 1:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement-Main-Contract
- $\rho_{f_0(k)+3}$ = Require
- (———End of state transition———)
- $\rho_{f_0(k)+4}$ = Function-Body

Sub-trace 2:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement
- $\rho_{f_0(k)+3}$ = Require
- (———End of state transition———)
- $\rho_{f_0(k)+4}$ = Function-Body

Similar to *Case*.2, we choose *sub-trace 1* for proving as follows.

(2) Similar to $Case_1$.2, $[\rho_{f_0(k)+1}, \dots, \rho_{f_0(k)+3}]$ in (1) corresponds to the state transition from $s_k$ to $s_{k+1}$.

(3) Similar to $Case_1$.3, let

$$c_k = \texttt{require}\ e_b; \texttt{stmt}$$

(4) Similar to $Case_1.4$, $C_r(c_k, i_k) = \texttt{require}\ e_b;\texttt{stmt}$

(5) Similar to $Case_1.5$, since $e_b = \texttt{true}$, we get $\mathcal{E}(\theta_e(e_b)) = \mathcal{E}(\sigma_v(e_b)) = \texttt{true}$ and rule `require_true` can be applied.

(6) Similar to $Case_1.6$, $\{\mathsf{Var}_i\} \subseteq names(F_{g(k)})$

(7) Similar to $Case_1.10$, $F_c(i_{k+1}, T(s_{k+1})) = \{\texttt{Var}_{\texttt{i}\circ\texttt{1}}\}$

(8) Let *step*=1, i.e., $g(k+1) = g(k) + 1$

(9) Similar to $Case_1.12$, *Condition **(a)*** proved.

(10) By Definition A.7, *condition **(b)**, **(c)**, **(d)**, **(e)**, **(f)**, **(g)**, **(h)**, **(i)**, **(j)**, **(k)** hold* trivially.

**Case:** $T(s_k) = 8$.

(1) Similar to (1), (2) in case $T(s_k) = 1$, the possible sub-traces for the state transition in (E) are as follows:

Sub-trace 1:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement-Main-Contract
- $\rho_{f_0(k)+3}$ = Require
- $\rho_{f_0(k)+4}$ = Exception-Propagation
- $\rho_{f_0(k)+5}$ = Update-Exception-State
- ...
- $\rho_{f_0(k)+t+5}$ = Update-Cur-Context
- $\rho_{f_0(k)+t+6}$ = Return-Context
- $\rho_{f_0(k)+t+7}$ = Clear-Recipient-Context
- $\rho_{f_0(k)+t+8}$ = Clear-Caller-Context
- $\rho_{f_0(k)+t+9}$ = Propagate-Exception-True
- ...

Sub-trace 2:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement
- $\rho_{f_0(k)+3}$ = Require
- $\rho_{f_0(k)+4}$ = Exception-Propagation
- $\rho_{f_0(k)+5}$ = Update-Exception-State
- ...
- $\rho_{f_0(k)+t+6}$ = Return-Context
- $\rho_{f_0(k)+t+7}$ = Clear-Recipient-Context
- $\rho_{f_0(k)+t+8}$ = Clear-Caller-Context
- $\rho_{f_0(k)+t+9}$ = Propagate-Exception-True
- ...

Here, the application of rule Update-Exception-State updates a key value in cell *contractStack* which finally results in the application of Propagate-Exception-True.

Since both traces violates $\alpha_{\rho_e}$, the cases do not satisfy the precondition of the lemma, i.e., *conditions of the case is proved.*

**Case:** $T(s_k) = 9$.

(1) Similar to (1), (2) in case $T(s_k) = 1$, the possible trace for the state transition in (E) is as follows:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement-Main-Contract
- $\rho_{f_0(k)+3}$ = Return-Value
- $\rho_{f_0(k)+4}$ = Update-Cur-Context
- $\rho_{f_0(k)+5}$ = Return-Context
- $\rho_{f_0(k)+6}$ = Clear-Recipient-Context
- $\rho_{f_0(k)+7}$ = Clear-Caller-Context
- $\rho_{f_0(k)+8}$ = Propagate-Exception-False
- (———End of state transition and function———)

Note that we omit traces that does not satisfy $\alpha_{\rho_e}$. Hence, we prove the conditions hold for trace.

(2) Similar to $Case_1.2$, $[\rho_{f_0(k)+1}, \ldots, \rho_{f_0(k)+8}]$ in (1) corresponds to the state transition from $s_k$ to the end of the function.

(3) Similar to $Case_1.3$, let

$$c_k = \texttt{return i}$$

(4) Similar to $Case_1.4$,

$$C_r(c_k, i_k) = c_k = \texttt{return i}$$

(5) Similar to $Case_1.5$, rule `ret_ext` is generated.

(6) Similar to $Case_1.6$, $\{\mathsf{Var}_i\} \subseteq names(F_{g(k)})$

(7) Similar to $Case_1.10$, $F_c(i_{k+1}, T(s_{k+1})) = \{\mathsf{GVar}, \mathsf{EVar}\}$

(8) Let *step*=1, i.e., $g(k+1) = g(k) + 1$

(9) Similar to $Case_1.15$,

$$\mathbb{G}(\gamma_k) \leftrightarrow_v gvars(\texttt{Var}_\texttt{i})$$

$$\mathbb{E}(\gamma_k) \leftrightarrow_v evars(\texttt{Var}_\texttt{i})$$

(10) Similar to $Case_1.16$,

$$gvars(\texttt{Var}_\texttt{i}) = gvars(\texttt{GVar})$$

$$evars(\texttt{Var}_\texttt{i}) = evars(\texttt{EVar})$$

(11) Similar to $Case_1.17$, $\lambda_{f_0(k)+8} = \gamma_{k+1}$

(12) Similar to $Case_1.18$,

$$\mathbb{G}(\lambda_{f_0(k)+8}) = \cdots = \mathbb{G}(\lambda_{f_0(k)+6}) = \mathbb{G}_g(\lambda_{f_0(k)}) = \mathbb{G}(\lambda_{f_0(k)})$$

$$\mathbb{L}(\lambda_{f_0(k)+8}) = \cdots = \mathbb{L}(\lambda_{f_0(k)+6}) = \mathbb{G}_l(\lambda_{f_0(k)}) = \emptyset$$

$$\mathbb{E}(\lambda_{f_0(k)+8}) = \cdots = \mathbb{E}(\lambda_{f_0(k)})$$

(13) By (9), (10), (11), (12), condition ***(b)**, **(c)**, **(d)**, **(e)*** hold.

(14) By (11), (12), condition ***(f)**, **(g)*** hold.

(15) By Definition A.7, ***(h)**, **(i)*** hold trivially.

(16) Similar to $Case_1.33$,

$$\mathbb{L}_A(\lambda_{f_0(k)+8}) = \cdots = \mathbb{L}(\lambda_{f_0(k)+3}) = \mathbb{L}_A(\lambda_{f_0(k)}) \backslash \{\mathbb{L}_{some}\}$$

where $\mathbb{L}_{some}$ is some value.

(17) By (16) and inductive hypothesis (j), *condition **(j)*** holds.

(18) Similar to $Case_1.35$, by (13), *condition **(k)*** holds.

**Case:** $T(s_k) = 10$.

(1) Similar to (1), (2) in case $T(s_k) = 1$, the possible sub-traces for the state transition in (E) are as follows:

Sub-trace 1:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement
- $\rho_{f_0(k)+3}$ = Return-Value
- $\rho_{f_0(k)+4}$ = Return-Context
- $\rho_{f_0(k)+5}$ = Clear-Recipient-Context
- $\rho_{f_0(k)+6}$ = Clear-Caller-Context
- $\rho_{f_0(k)+7}$ = Propagate-Exception-False
- (———End of state transition and function———)

Note that we omit traces that does not satisfy $\alpha_{\rho_e}$. Hence, we prove the conditions hold for trace.

(2) Let the ID of recipient and caller be $id_r$, $id_c$, respectively.

(3) Similar to $Case_1.2$, $[\rho_{f_0(k)+1}, \ldots, \rho_{f_0(k)+7}]$ in (1) corresponds to the state transition from $s_k$ to the end of the function.

(4) Similar to $Case_1.3$, let

$$c_k = \texttt{return i}$$

(5) Similar to *Case*$_1$.4,

$$C_r(c_k, i_k) = c_k = \texttt{return i}$$

(6) Similar to *Case*$_1$.5, rule `ret_in` is generated.

(7) By (6), both the recipient function and the caller function have been applied by $R_s$, where the caller's rule `in_call` and `recv_ret` has been generated.

(8) By (7), let the rule `in_call` is generated at the corresponding state

$$(\gamma_y, i_y, x(c_x).f_x(p); \texttt{stmt})$$

(9) Similar to *Case*$_1$.6, by (8), $\{\mathsf{Var}_{i_y}^{id_c}\} \subseteq \mathit{names}(F_{g(k)})$

(10) Similar to *Case*$_1$.10, $F_c(i_{k+1}, T(s_{k+1})) = \{\mathtt{Var}_{\mathtt{i_y \circ 1 \circ 1}}^{\mathtt{id_c}}\}$

(11) Let *step*=2, i.e., $g(k+1) = g(k) + 2$, and rule `in_call` and `recv_ret` are applied in these steps.

(12) Similar to *Case*$_1$.12, *condition **(a)*** holds.

(13) Similar to *Case*$_1$.17, $\lambda_{f_0(k)+7} = \gamma_{k+1}$

(14) By inductive hypothesis (k), we have $V \in^{\#} F_{g(k)}$, such that

$$\mathbb{G}(\gamma_k, id_c) \leftrightarrow_v \mathit{gvars}(V)$$

(15) Similar to *Case*$_1$.15,

$$\mathbb{G}(\gamma_k, id_r) \leftrightarrow_v \mathit{gvars}(\mathsf{Var}_i^{id_r})$$

(16) Similar to *Case*$_1$.18,

$$\mathbb{G}(\lambda_{f_0(k)+7}) = \cdots = \mathbb{G}(\lambda_{f_0(k)+6}) = \mathbb{G}_g(\lambda_{f_0(k)}, id_c) = \mathbb{G}(\lambda_{f_0(k)}, id_c)$$
$$\mathbb{L}(\gamma_{k+1}, id_r) = \mathbb{L}(\lambda_{f_0(k)+7}, id_r) = \cdots = \mathbb{L}(\lambda_{f_0(k)+5}, id_r) = \emptyset$$
$$\mathbb{L}(\lambda_{f_0(k)+7}, id_c) = \cdots = \mathbb{L}(\lambda_{f_0(k)+6}, id_c) = \mathbb{L}(\gamma_k, id_c) \in \mathbb{L}_A(\gamma_k)$$
$$\mathbb{E}(\lambda_{f_0(k)+7}) = \cdots = \mathbb{E}(\lambda_{f_0(k)})$$
$$\mathbb{G}(\gamma_{k+1}, id_c) = \mathbb{G}(\lambda_{f_0(k)+7})$$
$$\mathbb{G}(\gamma_{k+1}, id_r) = \mathbb{G}(\gamma_k, id_r)$$

(17) By (16) and inductive hypothesis (j), there exists $V_1$, such that $\mathbb{L}_A(\gamma_k) \leftrightarrow_v \mathit{vars}(V_1) \backslash \mathit{gvars}(V_1))$ and $V_1 \in^{\#} F_{g(k)}$

(18) By (11), (17), $V_1 \in^{\#} F_{g(k+1)}$

(19) By (13), (16), (17), (18), conditions ***(f)**, **(g)** hold.*

(20) By (13), (14), (16), condition ***(b)**, **(c)** hold.*

(21) By (13), (16), condition ***(d)**, **(e)** hold.*

(22) By Definition A.7, ***(h)**, **(i)*** hold trivially.

(23) Similar to *Case*$_9$.17, *condition* (***j***) *holds* trivially.

(24) By (11), (14), $V \in^{\#} F_{g(k+1)}$.

(25) By (11), $\mathit{gvars}(\mathsf{Var}_i^{id_r}) = \mathit{gvars}(\mathsf{GVar}^{id_r})$ and $\mathsf{GVar}^{id_r} \in^{\#} F_{g(k+1)}$

(26) Similar to *Case*$_1$.35, by (14), (16), (24), (25), *condition **(k)*** holds.

**Case:** $T(s_k) = 11$.

(1) Similar to (1), (2) in case $T(s_k) = 1$, the possible sub-traces for the state transition in (E) are as follows:

Sub-trace 1:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement-Main-Contract
- $\rho_{f_0(k)+3}$ = Transfer-Fund-Begin
- $\rho_{f_0(k)+4}$ = Gas-Cal
- $\rho_{f_0(k)+5}$ = Transfer-Fund
- (———End of state transition———)
- $\rho_{f_0(k)+6}$ = Function-Body

Sub-trace 2:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement
- $\rho_{f_0(k)+3}$ = Transfer-Fund-Begin
- $\rho_{f_0(k)+4}$ = Gas-Cal
- $\rho_{f_0(k)+5}$ = Transfer-Fund
- (———End of state transition———)
- $\rho_{f_0(k)+6}$ = Function-Body

Similar to *Case*.2, we choose *sub-trace 1* for proving as follows.

(2) Similar to *Case*$_1$.2, $[\rho_{f_0(k)+1}, \ldots, \rho_{f_0(k)+5}]$ in (1) corresponds to the state transition from $s_k$ to $s_{k+1}$.

(3) Similar to *Case*$_1$.3, let $c_k = c_x.\texttt{transfer}(v_1); \texttt{stmt}$ (For readability, assume in $\mathcal{K}_s$, $c_x, c$ is the balance of the receiver and the account of $\gamma$, respectively)

(4) Similar to *Case*$_1$.4, $C_r(c_k, i_k) = c_x.\texttt{transfer}(v_1); \texttt{stmt}$

(5) Similar to *Case*$_1$.5, rule `transfer_succ` is generated.

(6) Similar to *Case*$_1$.6, $\{\mathsf{Var}_i\} \subseteq \mathit{names}(F_{g(k)})$

(7) Similar to *Case*$_1$.10, $F_c(i_{k+1}, T(s_{k+1})) = \{\mathtt{Var_{i \circ 1}}\}$

(8) Let *step*=1, i.e., $g(k+1) = g(k) + 1$

(9) Similar to *Case*$_1$.12, *condition **(a)*** holds.

(10) By (B), $\gamma_k = \lambda_{f_0(k)}$

(11) Similar to *Case*$_1$.15, $\mathbb{E}(\gamma_k) \leftrightarrow_v \mathit{evars}(\mathsf{Var}_i)$

(12) By assumption of $f$, $\lambda_{f_0(k)+5} = \gamma_{k+1}$

(13) Similar to *Case*$_1$.18,

$$\mathbb{E}_{\gamma_k} = \mathbb{E}_{\lambda_{f_0(k)}} = \cdots = \mathbb{E}_{\lambda_{f_0(k)+4}}$$

(14) Let $(id_c, c) \in \mathbb{E}(\gamma_k)$ and $(id_x, c_x) \in \mathbb{E}(\gamma_k)$

(15) By (11), let $(id_c, c) \leftrightarrow_v \sigma_v(c)$ and $(id_x, c_x) \leftrightarrow_v \sigma_v(c_x)$

(16) By (12),(13),(14)

$$\mathbb{E}_{\gamma_{k+1}} = \mathbb{E}_{\lambda_{f_0(k)+5}} =$$
$$= \mathbb{E}_{\gamma_k} \cup \{(id_c, c - v_1), (id_x, c_x + v_1)\} \backslash \{(id_c, c), (id_x, c_x)\}$$

(17) By (15) and Definition A.22,

$$\mathcal{E}(\sigma_v(c)) = c \wedge \mathcal{E}(\sigma_v(c_x)) = c_x$$

(18) By (5), $\mathit{evars}(\mathsf{Var}_{i \circ 1}) =$

$$\mathit{evars}(\mathsf{Var}_i) \cup \{\sigma_v(c) \ominus \sigma_v(v_1), \sigma_v(c_x) \oplus \sigma_v(v_1)\} \backslash \{\sigma_v(c), \sigma_v(c_x)\}$$

(19) By (17) and Definition A.22, let $\mathcal{E}(\sigma_v(v_1)) = v_1$, we get

$$\mathcal{E}(\sigma_v(c) \ominus \sigma_v(v_1)) = c - v_1$$
$$\mathcal{E}(\sigma_v(c_x) \oplus \sigma_v(v_1)) = c_x + v_1$$

(20) By (19), let $(id_c, c - v_1) \leftrightarrow_v (\sigma_v(c) \oplus \sigma_v(v_1))$, and let $(id_x, c_x \ominus \sigma_v(v_1)) \leftrightarrow_v \sigma_v(c_x) + v_1$

(21) By (11), (16), (18), (20), and Remark 1, *condition **(d)**, **(e)*** hold

(22) By Definition A.7, *condition **(b)**, **(c)**, **(f)**, **(g)**, **(h)**, **(i)**, **(j)**, **(k)** hold* trivially.

**Case:** $T(s_k) = 12$.

(1) Similar to (1), (2) in case $T(s_k) = 1$, the possible sub-traces for the state transition in (E) are as follows:

Sub-trace 1:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement-Main-Contract
- $\rho_{f_0(k)+3}$ = Transfer-Fund-Begin
- $\rho_{f_0(k)+4}$ = Gas-Cal-Fail

- $\rho_{f_0(k)+5}$ = Exception-Propagation
- $\rho_{f_0(k)+6}$ = Update-Exception-State
- ...
- $\rho_{f_0(k)+t+7}$ = Update-Cur-Context
- $\rho_{f_0(k)+t+8}$ = Return-Context
- $\rho_{f_0(k)+t+9}$ = Clear-Recipient-Context
- $\rho_{f_0(k)+t+10}$ = Clear-Caller-Context
- $\rho_{f_0(k)+t+11}$ = Propagate-Exception-True
- ...

Sub-trace 2:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement
- $\rho_{f_0(k)+3}$ = Transfer-Fund-Begin
- $\rho_{f_0(k)+4}$ = Gas-Cal-Fail
- $\rho_{f_0(k)+5}$ = Exception-Propagation
- $\rho_{f_0(k)+6}$ = Update-Exception-State
- ...
- $\rho_{f_0(k)+t+7}$ = Return-Context
- $\rho_{f_0(k)+t+8}$ = Clear-Recipient-Context
- $\rho_{f_0(k)+t+9}$ = Clear-Caller-Context
- $\rho_{f_0(k)+t+10}$ = Propagate-Exception-True
- ...

Similar to *Case*.8, all conditions holds.

**Case:** $T(s_k) = 13$.

(1) Similar to (1), (2) in case $T(s_k) = 1$, the possible sub-traces for the state transition in (E) are as follows:

Sub-trace 1:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement-Main-Contract
- $\rho_{f_0(k)+3}$ = Send-Fund-Begin
- $\rho_{f_0(k)+4}$ = Gas-Cal
- $\rho_{f_0(k)+5}$ = Send-Fund-Successful
- (———End of state transition———)
- $\rho_{f_0(k)+6}$ = Function-Body

Sub-trace 2:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement
- $\rho_{f_0(k)+3}$ = Send-Fund-Begin
- $\rho_{f_0(k)+4}$ = Gas-Cal
- $\rho_{f_0(k)+5}$ = Send-Fund-Successful
- (———End of state transition———)
- $\rho_{f_0(k)+6}$ = Function-Body

The proof for the case is similar to the one for $Case_{11}$, and we omit the proof.

**Case:** $T(s_k) = 14$.

(1) Similar to (1), (2) in case $T(s_k) = 1$, the possible sub-traces for the state transition in (E) are as follows:

Sub-trace 1:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement-Main-Contract
- $\rho_{f_0(k)+3}$ = Send-Fund-Begin
- $\rho_{f_0(k)+4}$ = Gas-Cal
- $\rho_{f_0(k)+5}$ = Send-Fund-Failed
- (———End of state transition———)
- $\rho_{f_0(k)+6}$ = Function-Body

Sub-trace 2:

- $\rho_{f_0(k)+1}$ = Function-Body
- $\rho_{f_0(k)+2}$ = Exe-Statement
- $\rho_{f_0(k)+3}$ = Send-Fund-Begin
- $\rho_{f_0(k)+4}$ = Gas-Cal
- $\rho_{f_0(k)+5}$ = Send-Fund-Failed
- (———End of state transition———)
- $\rho_{f_0(k)+6}$ = Function-Body

(2) Similar to $Case_1$.12, *Condition **(a)*** proved.

(3) By Definition A.7, Condition ***(b)**, **(c)**, **(d)**, **(e)**, **(f)**, **(g)**, **(h)**, **(i)**, **(j)**, **(k)** hold* trivially.

□

It can be easily inferred from Theorem A.25 about soundness of verifying invariant properties and equivalence properties by condition (b), (d), (k). Note that completeness cannot be achieved by FASVERIF. The reason is the over-estimation on valuation $\mathcal{E}$ such that there may not exist a trace in $K_s$ that corresponds to a trace in $R_s$. To achieve completeness, complete constraints on values of balance, global variables, and local variables are required.

RULE REQUIRE

$$\left\langle \frac{\texttt{require(true);}}{\cdot}\cdots\right\rangle_{k} \quad \left\langle \frac{\texttt{require(false);}}{\texttt{exception()}}\cdots\right\rangle_{k}$$

RULE OUT-OF-GAS

$$\left\langle \frac{\texttt{S:Statement}}{\texttt{exception()}}\cdots\right\rangle_{k} \quad \left\langle \ \texttt{\#msgInfo(\_,\_,\_,GasLimit)}\ \right\rangle_{\mathit{Msg}}$$

$$\left\langle \ \texttt{GasC}\ \right\rangle_{\mathit{gasConsumption}}$$

`requires GasC >Int GasLimit`

RULE REVERT

$$\left\langle \frac{\texttt{revert(.ExpressionList);}}{\texttt{exception()}}\cdots\right\rangle_{k}$$

RULE ASSERT

$$\left\langle \frac{\texttt{assert(true);}}{\cdot}\cdots\right\rangle_{k} \quad \left\langle \frac{\texttt{assert(false);}}{\texttt{exception()}}\cdots\right\rangle_{k}$$

RULE EXCEPTION-PROPAGATION

$$\left\langle \frac{\texttt{exception()}}{\texttt{updateExceptionState()}}\cdots\right\rangle_{k}$$

$$\left\langle \ \texttt{ListItem(R)ListItem(C)}\ \cdots\right\rangle_{\mathit{contractStack}}$$

`requires C >=Int 0`

RULE TRANSACTION-REVERSION

$$\left\langle \frac{\texttt{exception()}}{\texttt{updateExceptionState()} \curvearrowright \texttt{revertState()}}\cdots\right\rangle_{k}$$

$$\left\langle \ \texttt{ListItem(R)ListItem(-1)}\ \right\rangle_{\mathit{contractStack}}$$

RULE UPDATE-EXCEPTION-STATE

$$\left\langle \frac{\texttt{updateExceptionState()}}{\cdot}\cdots\right\rangle_{k}$$

$$\left\langle \frac{\texttt{ListItem(\#state(\_,\_,\_,\_,\_))}}{\texttt{ListItem(\#state(\_,\_,\_,\_,true))}}\cdots\right\rangle_{\mathit{functionStack}}$$

RULE REVERT-STATE

$$\left\langle \frac{\texttt{revertState()}}{\begin{array}{c}\texttt{revertInContracts(PreCNum,0)}\curvearrowright \\ \texttt{deleteNewContracts(PreCNum,CNum)}\end{array}}\cdots\right\rangle_{k}$$

$$\left\langle \ \texttt{ListItem(\#state(\_,\_,\_,PreCNum,\_))}\ \cdots\right\rangle_{\mathit{functionStack}}$$

$$\left\langle \ \texttt{CNum}\ \right\rangle_{\mathit{cntContracts}}$$

RULE TRANSFER-FUND-BEGIN

$$\left\langle \frac{\texttt{\#memberAccess(R:Id,F:Id)} \curvearrowright \texttt{MsgValue:Int}}{\begin{array}{c}\texttt{gasCal(\#read,String2Id("Global"))} \curvearrowright \\ \texttt{\#transferFund(}\mathrm{N}_{\mathrm{R}}\texttt{,N,MsgValue)}\end{array}}\cdots\right\rangle_{k}$$

$$\left\langle \ \texttt{ListItem(N:Int)}\ \cdots\right\rangle_{\mathit{contractStack}}$$

$$\left\langle \begin{array}{c}\left\langle \ \texttt{N}\ \right\rangle_{\mathit{ctId}} \\ \left\langle\cdots\ \texttt{R |-> Addr}\ \cdots\right\rangle_{\mathit{ctContext}} \\ \left\langle\cdots\ \texttt{R |-> String2Id("Local")}\ \cdots\right\rangle_{\mathit{ctLocation}} \\ \left\langle\cdots\ \texttt{R |-> address}\ \cdots\right\rangle_{\mathit{ctType}} \\ \left\langle\cdots\ \texttt{Addr |-> }\mathrm{N}_{\mathrm{R}}\ \cdots\right\rangle_{\mathit{ctStorage}}\end{array}\ \cdots\right\rangle_{\mathit{contractInstance}}$$

`requires Id2String(F) ==String "transfer"`

RULE TRANSFER-FUND

$$\left\langle \frac{\texttt{\#transferFund(}\mathrm{N}_{\mathrm{R}}\texttt{,N,MsgValue)}}{\cdot}\cdots\right\rangle_{k}$$

$$\left\langle \begin{array}{c} \left\langle \begin{array}{c} \left\langle\ \texttt{N}\ \right\rangle_{\mathit{ctId}} \\ \left\langle \frac{\mathrm{B}_{\mathrm{N}}}{\mathrm{B}_{\mathrm{N}}\ \texttt{-Int MsgValue}} \right\rangle_{\mathit{balance}} \end{array} \right\rangle_{\mathit{contractInstance}} \\ \left\langle \begin{array}{c} \left\langle\ \mathrm{N}_{\mathrm{R}}\ \right\rangle_{\mathit{ctId}} \\ \left\langle \frac{\mathrm{B}_{\mathrm{R}}}{\mathrm{B}_{\mathrm{R}}\ \texttt{+Int MsgValue}} \right\rangle_{\mathit{balance}} \end{array} \right\rangle_{\mathit{contractInstance}} \end{array} \right\rangle_{\mathit{contractInstances}}$$

RULE REVERT-IN-CONTRACTS

$$\left\langle \frac{\texttt{revertInContracts(PreCNum:Int,0)}}{\cdot}\cdots\right\rangle_{k}$$

$$\left\langle \begin{array}{c}
\left\langle \begin{array}{c} \left\langle\ \mathrm{N}_0\ \right\rangle_{\mathit{ctId}} \\ \left\langle\frac{\mathrm{ORhoC}_0}{\mathrm{RhoC}_0}\right\rangle_{\mathit{ctContext}} \\ \left\langle\ \mathrm{RhoC}_0\ \right\rangle_{\mathit{tempContext}} \\ \left\langle\frac{\mathrm{OS}_0}{\mathrm{S}_0}\right\rangle_{\mathit{ctStorage}} \\ \left\langle\ \mathrm{S}_0\ \right\rangle_{\mathit{tempStorage}} \\ \left\langle\frac{\mathrm{OM}_0}{\mathrm{M}_0}\right\rangle_{\mathit{Memory}} \\ \left\langle\ \mathrm{M}_0\ \right\rangle_{\mathit{tempMemory}} \\ \left\langle\frac{\mathrm{OB}_0}{\mathrm{B}_0}\right\rangle_{\mathit{Balance}} \\ \left\langle\ \mathrm{B}_0\ \right\rangle_{\mathit{tempBalance}} \end{array}\right\rangle_{\mathit{contractInstance}} \\
\left\langle \begin{array}{c} \left\langle\ \mathrm{N}_1\ \right\rangle_{\mathit{ctId}} \\ \left\langle\frac{\mathrm{ORhoC}_1}{\mathrm{RhoC}_1}\right\rangle_{\mathit{ctContext}} \\ \left\langle\ \mathrm{RhoC}_1\ \right\rangle_{\mathit{tempContext}} \\ \left\langle\frac{\mathrm{OS}_1}{\mathrm{S}_1}\right\rangle_{\mathit{ctStorage}} \\ \left\langle\ \mathrm{S}_1\ \right\rangle_{\mathit{tempStorage}} \\ \left\langle\frac{\mathrm{OM}_1}{\mathrm{M}_1}\right\rangle_{\mathit{Memory}} \\ \left\langle\ \mathrm{M}_1\ \right\rangle_{\mathit{tempMemory}} \\ \left\langle\frac{\mathrm{OB}_1}{\mathrm{B}_1}\right\rangle_{\mathit{Balance}} \\ \left\langle\ \mathrm{B}_1\ \right\rangle_{\mathit{tempBalance}} \end{array}\right\rangle_{\mathit{contractInstance}} \\
\cdots \\
\left\langle \begin{array}{c} \left\langle\ \mathrm{N}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}\ \right\rangle_{\mathit{ctId}} \\ \left\langle\frac{\mathrm{ORhoC}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}}{\mathrm{RhoC}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}}\right\rangle_{\mathit{ctContext}} \\ \left\langle\ \mathrm{RhoC}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}\ \right\rangle_{\mathit{tempContext}} \\ \left\langle\frac{\mathrm{OS}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}}{\mathrm{S}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}}\right\rangle_{\mathit{ctStorage}} \\ \left\langle\ \mathrm{S}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}\ \right\rangle_{\mathit{tempStorage}} \\ \left\langle\frac{\mathrm{OM}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}}{\mathrm{M}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}}\right\rangle_{\mathit{Memory}} \\ \left\langle\ \mathrm{M}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}\ \right\rangle_{\mathit{tempMemory}} \\ \left\langle\frac{\mathrm{OB}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}}{\mathrm{B}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}}\right\rangle_{\mathit{Balance}} \\ \left\langle\ \mathrm{B}_{\mathrm{PreCNum}\ -\mathrm{Int}\ 1}\ \right\rangle_{\mathit{tempBalance}} \end{array}\right\rangle_{\mathit{contractInstance}}
\end{array}\right\rangle_{\mathit{contractInstances}}$$

RULE DELETE-NEW-CONTRACTS

$$\left\langle \frac{\texttt{deleteNewContracts(PreCNum:Int,CNum:Int)}}{\cdot}\cdots\right\rangle_{k}$$

$$\left\langle \frac{\begin{array}{c}\texttt{INS:Bag}\ \left\langle \left\langle\ \texttt{PreCNum}\ \right\rangle_{\mathit{ctId}} \right\rangle_{\mathit{contractInstance}} \\ \left\langle \left\langle\ \texttt{PreCNum +Int 1}\ \right\rangle_{\mathit{ctId}} \right\rangle_{\mathit{contractInstance}} \cdots \\ \left\langle \left\langle\ \texttt{CNum -Int 1}\ \right\rangle_{\mathit{ctId}} \right\rangle_{\mathit{contractInstance}}\end{array}}{\texttt{INS}}\right\rangle_{\mathit{contractInstances}}$$

RULE SEND-FUND-SUCCESSFUL

$$\left\langle \frac{\texttt{\#sendFund(}\mathrm{N}_{\mathrm{R}}\texttt{,N,MsgValue)}}{\texttt{true}}\cdots\right\rangle_{k}$$

$$\left\langle \begin{array}{c} \left\langle \begin{array}{c} \left\langle\ \texttt{N}\ \right\rangle_{\mathit{ctId}} \\ \left\langle \frac{\mathrm{B}_{\mathrm{N}}}{\mathrm{B}_{\mathrm{N}}\ \texttt{-Int MsgValue}} \right\rangle_{\mathit{balance}} \end{array} \right\rangle_{\mathit{contractInstance}} \\ \left\langle \begin{array}{c} \left\langle\ \mathrm{N}_{\mathrm{R}}\ \right\rangle_{\mathit{ctId}} \\ \left\langle \frac{\mathrm{B}_{\mathrm{R}}}{\mathrm{B}_{\mathrm{R}}\ \texttt{+Int MsgValue}} \right\rangle_{\mathit{balance}} \end{array} \right\rangle_{\mathit{contractInstance}} \end{array} \right\rangle_{\mathit{contractInstances}}$$

RULE SEND-FUND-FAILED

$$\left\langle \frac{\texttt{\#sendFund(}\mathrm{N}_{\mathrm{R}}\texttt{,N,MsgValue)}}{\texttt{false}}\cdots\right\rangle_{k}$$

$$\left\langle \ \texttt{\#msgInfo(\_,\_,\_,GasLimit)}\ \right\rangle_{\mathit{Msg}}$$

$$\left\langle \ \texttt{GasC}\ \right\rangle_{\mathit{gasConsumption}}$$

`requires GasC >Int GasLimit`

**Figure 11: Definition of $\mathcal{K}_s$ (Part 1).**

RULE Send-Fund-Begin

```
⟨  #memberAccess(R:Id,F:Id) ↷ MsgValue:Int
  ---------------------------------------------
   gasCal(#read,String2Id("Global")) ↷
           #sendFund(N_R,N,MsgValue)              ...⟩k
⟨ ListItem(N:Int) ...⟩contractStack
⟨                    ⟨ N ⟩ctId
             ⟨... R |-> Addr ...⟩ctContext
   ⟨... R |-> String2Id("Local") ...⟩ctLocation      ...⟩contractInstance
            ⟨... R |-> address ...⟩ctType
            ⟨... Addr |-> N_R ...⟩ctStorage
requires Id2String(F) ==String "send"
```

RULE Write

```
⟨     X:Id = V:Value
  ----------------------- ...⟩k
  writeAddress(Addr,L,V)
⟨ ListItem(N:Int) ...⟩contractStack
⟨                ⟨ N ⟩ctId
     ⟨... X |-> Addr ...⟩ctContext
       ⟨... X |-> L ...⟩ctLocation         ...⟩contractInstance
   ⟨... X |-> T:EleType ...⟩ctType
```

RULE WriteAddress-GlobalVariables

```
⟨  writeAddress(Addr:Int,String2Id("Global"),
                   V:Value)
  ------------------------------------------------ ...⟩k
  gasCal(#write,String2Id("Global"),OV,V) ↷ V
⟨ ListItem(N:Int) ...⟩contractStack
⟨            ⟨ N ⟩ctId
   ⟨... Addr |-> OV
        ------------ ...⟩ctStorage      ...⟩contractInstance
        Addr |-> V
```

RULE WriteAddress-LocalVariables

```
⟨  writeAddress(Addr:Int,String2Id("Local"),
                  V:Value)
  ----------------------------------------------- ...⟩k
  gasCal(#write,String2Id("Local"),OV,V) ↷ V
⟨ ListItem(N:Int) ...⟩contractStack
⟨            ⟨ N ⟩ctId
   ⟨... Addr |-> OV
        ------------ ...⟩Memory      ...⟩contractInstance
        Addr |-> V
```

RULE Read

```
⟨       X:Id
  --------------------- ...⟩k
  readAddress(Addr,L)
⟨ ListItem(N:Int) ...⟩contractStack
⟨                ⟨ N ⟩ctId
     ⟨... X |-> Addr ...⟩ctContext
       ⟨... X |-> L ...⟩ctLocation         ...⟩contractInstance
   ⟨... X |-> T:EleType ...⟩ctType
```

RULE ReadAddress-GlobalVariables

```
⟨   readAddress(Addr:Int,String2Id("Global"))
  ------------------------------------------------ ...⟩k
  gasCal(#read,String2Id("Global")) ↷ V:Value
⟨ ListItem(N:Int) ...⟩contractStack
⟨           ⟨ N ⟩ctId
   ⟨... Addr |-> V ...⟩ctStorage     ...⟩contractInstance
```

RULE ReadAddress-LocalVariables

```
⟨   readAddress(Addr:Int,String2Id("Local"))
  ----------------------------------------------- ...⟩k
  gasCal(#read,String2Id("Local")) ↷ V:Value
⟨ ListItem(N:Int) ...⟩contractStack
⟨           ⟨ N ⟩ctId
   ⟨... Addr |-> V ...⟩Memory     ...⟩contractInstance
```

RULE AllocateStateVariables

```
⟨  allocateStateVars(N:Int,
   ListItem(Var) Vars:List)
  --------------------------- ...⟩k
    allocate(N, Var) ↷
  allocateStateVars(N, Vars)
```

RULE New-Contract-Instance-Creation

```
⟨        new X:Id (E:ExpressionList)
  ---------------------------------------------- ...⟩k
  updateState(X) ↷ allocateStorage(X) ↷
              initInstance(X,E)
```

RULE UpdateState-Main-Contract

```
⟨   updateState(X:Id)
  --------------------- ...⟩k  ⟨ ⟨ X ⟩cName ...⟩contract
  gasCal(#newInstance)
⟨  N:Int
  ---------- ⟩cntContracts  ⟨  T:Int
  N +Int 1                     ---------- ⟩cntTrans
                               T +Int 1
⟨                  INS:Bag
  ------------------------------------------------ ⟩contractInstances
  INS ⟨   ⟨ N ⟩ctId
         ⟨ X ⟩ctName     ...⟩contractInstance
⟨             Trans:Map
  ---------------------------------- ⟩tranComputation
  Trans (T |-> "new contract")
⟨     L:List
  --------------- ⟩newStack  ⟨ .List ⟩functionStack
  ListItem(X) L
```

RULE UpdateState-Function-Call

```
⟨   updateState(X:Id)
  --------------------- ...⟩k  ⟨ ⟨ X ⟩cName ...⟩contract
  gasCal(#newInstance)
⟨  N:Int
  ---------- ⟩cntContracts
  N +Int 1
⟨                  INS:Bag
  ------------------------------------------------ ⟩contractInstances
  INS ⟨   ⟨ N ⟩ctId
         ⟨ X ⟩ctName     ...⟩contractInstance
⟨     L:List
  --------------- ⟩newStack  ⟨ CallList:List ⟩functionStack
  ListItem(X) L
requires CallList =/=K .List
```

RULE AllocateStorage

```
⟨        allocateStorage(X:Id)
  ---------------------------------- ...⟩k
  allocateStateVars(N -Int 1, Vars)
⟨ ⟨ X ⟩cName ⟨ Vars:List ⟩stateVars ...⟩contract
⟨ N:Int ⟩cntContracts
```

RULE InitInstance-NoConstructor

```
⟨  initInstance(X:Id,E:ExpressionList)
  ------------------------------------- ...⟩k
                N -Int 1
⟨ ListItem(X) L:List
  ------------------ ⟩newStack  ⟨ N:Int ⟩cntContracts
          L
⟨ ⟨ X ⟩cName ⟨ false ⟩Constructor ...⟩Contract
```

RULE InitInstance-WithConstructor

```
⟨      initInstance(X:Id,E:ExpressionList)
  --------------------------------------------- ...⟩k
            functionCall(C;N -Int 1;
          String2Id("constructor");E;
  #msgInfo(C,N -Int 1,0,gasCal(#constructor)))
⟨ ListItem(X) L:List
  ------------------ ⟩newStack  ⟨ N:Int ⟩cntContracts
          L
⟨ ListItem(C:Int) ...⟩contractStack
⟨ ⟨ X ⟩cName ⟨ true ⟩Constructor ...⟩contract
```

RULE Decompose-Solidity-Call

```
⟨  #memberAccess(R:Int,F:Id) ↷ Es:Values ↷
          MsgValue:Int ↷ MsgGas:Int
  ------------------------------------------ ...⟩k
             functionCall(C;R;F;Es;
        #msgInfo(C,R,MsgValue,MsgGas))
⟨ ListItem(C:Int) ...⟩contractStack
```

RULE Function-Call

```
⟨        functionCall(C:Int;R:Int;
           F:Id;Es:Values;M:Msg)
  ------------------------------------------ ...⟩k
         switchContext(C,R,F,M) ↷
  functionCall(F;Es) ↷ returnContext(R)
```

**Figure 12: Definition of $\mathcal{K}_s$ (Part 2).**

RULE AllocateStateVariables-End

$$\left\langle \frac{\texttt{allocateStateVars(N:Int, .List)}}{\cdot} \cdots \right\rangle_{k}$$

RULE Allocate

$$\left\langle \frac{\begin{array}{c}\texttt{allocate(N:Int, \#varInfo(X:Id,}\\ \texttt{T:EleType, L:Id, V:Value))}\end{array}}{\texttt{allocateAddress(N,Addr,L,V)}} \cdots \right\rangle_{k}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId} \\ \left\langle \frac{\texttt{Addr}}{\texttt{Addr +Int 1}} \right\rangle_{slotNum} \\ \left\langle \frac{\texttt{CONTEXT:Map}}{\texttt{CONTEXT (X |-> Addr)}} \right\rangle_{ctContext} \\ \left\langle \frac{\texttt{TYPE:Map}}{\texttt{TYPE (X |-> T)}} \right\rangle_{ctType} \\ \left\langle \frac{\texttt{LOCATION:Map}}{\texttt{LOCATION (X |-> L)}} \right\rangle_{ctLocation} \end{array} \cdots \right\rangle_{contractInstance}$$

RULE AllocateAddress-GlobalVariables

$$\left\langle \frac{\begin{array}{c}\texttt{allocateAddress(N:Int, Addr:Int,}\\ \texttt{String2Id("Global"), V:Value)}\end{array}}{\texttt{gasCal(\#allocate,String2Id("Global"))} \curvearrowright \texttt{V}} \cdots \right\rangle_{k}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId} \\ \left\langle \frac{\texttt{STORAGE:Map}}{\texttt{STORAGE (Addr |-> V)}} \right\rangle_{ctStorage} \end{array} \cdots \right\rangle_{contractInstance}$$

RULE AllocateAddress-LocalVariables

$$\left\langle \frac{\begin{array}{c}\texttt{allocateAddress(N:Int, Addr:Int,}\\ \texttt{String2Id("Local"), V:Value)}\end{array}}{\texttt{gasCal(\#allocate,String2Id("Local"))} \curvearrowright \texttt{V}} \cdots \right\rangle_{k}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId} \\ \left\langle \frac{\texttt{MEMORY:Map}}{\texttt{MEMORY (Addr |-> V)}} \right\rangle_{Memory} \end{array} \cdots \right\rangle_{contractInstance}$$

RULE Switch-Context

$$\left\langle \frac{\texttt{switchContext(C:Int,R:Int,F:Id,M:Msg)}}{\texttt{createTransaction(L)}} \cdots \right\rangle_{k}$$

$$\left\langle \frac{\texttt{L:List}}{\texttt{ListItem(R) L}} \right\rangle_{contractStack} \langle\ \texttt{CNum}\ \rangle_{cntContracts}$$

$$\left\langle \frac{\texttt{M1}}{\texttt{M}} \right\rangle_{Msg} \left\langle \frac{\texttt{MsgList:List}}{\texttt{ListItem(M1) MsgList}} \right\rangle_{msgStack}$$

$$\left\langle \frac{\texttt{CallList:List}}{\begin{array}{c}\texttt{ListItem(\#state(RhoC,F,}\\ \texttt{\#return(false,0),CNum,false)) CallList}\end{array}} \right\rangle_{functionStack}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{C}\ \rangle_{ctId}\ \langle\ \texttt{RhoG}\ \rangle_{globalContext} \\ \left\langle \frac{\texttt{RhoC}}{\texttt{RhoG}} \right\rangle_{ctContext} \end{array} \cdots \right\rangle_{contractInstance}$$

$$\left\langle \frac{\texttt{G}}{\texttt{0}} \right\rangle_{gasConsumption} \left\langle \frac{\texttt{GasList:List}}{\texttt{ListItem(G) GasList}} \right\rangle_{gasStack}$$

RULE Return-Context

$$\left\langle \frac{\texttt{returnContext(R:Int)}}{\begin{array}{c}\texttt{clearRecipientContext(R,RhoG)}\curvearrowright\\ \texttt{clearCallerContext(C,Rho)}\curvearrowright\\ \texttt{propagateException(C,Exception)}\curvearrowright\texttt{E:Value}\end{array}} \cdots \right\rangle_{k}$$

$$\left\langle \frac{\texttt{ListItem(R) ListItem(C) L:List}}{\texttt{ListItem(C) L}} \right\rangle_{contractStack}$$

$$\left\langle \frac{\texttt{M}}{\texttt{M1}} \right\rangle_{Msg} \left\langle \frac{\texttt{ListItem(M1) MsgList:List}}{\texttt{MsgList}} \right\rangle_{msgStack}$$

$$\left\langle \frac{\begin{array}{c}\texttt{ListItem(\#state(Rho,\_,\#return(\_,E),}\\ \texttt{\_,Exception)) CallList:List}\end{array}}{\texttt{CallList}} \right\rangle_{functionStack}$$

$$\left\langle \langle\ \texttt{R}\ \rangle_{ctId}\ \langle\ \texttt{RhoG}\ \rangle_{globalContext}\ \cdots \right\rangle_{contractInstance}$$

$$\left\langle \frac{\texttt{G:Int}}{\texttt{G +Int G1}} \right\rangle_{gasConsumption}$$

$$\left\langle \frac{\texttt{ListItem(G1) GasList:List}}{\texttt{GasList}} \right\rangle_{gasStack}$$

RULE Internal-Function-Call

$$\left\langle \frac{\texttt{functionCall(F:Id;Es:Values)}}{\begin{array}{c}\texttt{saveCurContext(CNum,0)} \curvearrowright\\ \texttt{call(searchFunction(F,checkCallData(Es,0)),Es)}\\ \curvearrowright \texttt{updateCurContext(CNum,0)}\end{array}} \cdots \right\rangle_{k}$$

$$\langle\ \texttt{CNum}\ \rangle_{cntContracts}\ \langle\ \texttt{.List}\ \rangle_{functionStack}$$

RULE Nested-Function-Call

$$\left\langle \frac{\texttt{functionCall(F:Id,Es:Values)}}{\texttt{call(searchFunction(F,checkCallData(Es,0)),Es)}} \cdots \right\rangle_{k}$$

$$\langle\ \texttt{CallList:List}\ \rangle_{functionStack}$$

`requires CallList =/=K .List`

RULE Clear-Recipient-Context

$$\left\langle \frac{\texttt{clearRecipientContext(R:Id,RhoG:Map)}}{\cdot} \cdots \right\rangle_{k}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{R}\ \rangle_{ctId} \\ \left\langle \frac{\texttt{RhoC:Map}}{\texttt{RhoG}} \right\rangle_{ctContext} \end{array} \right\rangle_{contractInstance}$$

RULE Save-Cur-Context

$$\left\langle \frac{\texttt{saveCurContext(CNum:Int,0)}}{\cdot} \cdots \right\rangle_{k}$$

$$\left\langle \frac{\texttt{ListItem(\#state(\_,\_,\_,\_,\_))}}{\texttt{ListItem(\#state(\_,\_,\_,CNum,\_))}} \right\rangle_{functionStack}$$

$$\left\langle \begin{array}{c}
\left\langle \begin{array}{c} \langle\ \mathrm{N}_0\ \rangle_{ctId} \\ \langle\ \mathrm{RhoC}_0\ \rangle_{ctContext} \\ \left\langle \frac{\mathrm{ORhoC}_0}{\mathrm{RhoC}_0} \right\rangle_{tempContext} \\ \langle\ \mathrm{S}_0\ \rangle_{ctStorage} \\ \left\langle \frac{\mathrm{OS}_0}{\mathrm{S}_0} \right\rangle_{tempStorage} \\ \langle\ \mathrm{M}_0\ \rangle_{Memory} \\ \left\langle \frac{\mathrm{OM}_0}{\mathrm{M}_0} \right\rangle_{tempMemory} \\ \langle\ \mathrm{B}_0\ \rangle_{Balance} \\ \left\langle \frac{\mathrm{OB}_0}{\mathrm{B}_0} \right\rangle_{tempBalance} \end{array} \right\rangle_{contractInstance} \\
\left\langle \begin{array}{c} \langle\ \mathrm{N}_1\ \rangle_{ctId} \\ \langle\ \mathrm{RhoC}_1\ \rangle_{ctContext} \\ \left\langle \frac{\mathrm{ORhoC}_1}{\mathrm{RhoC}_1} \right\rangle_{tempContext} \\ \langle\ \mathrm{S}_1\ \rangle_{ctStorage} \\ \left\langle \frac{\mathrm{OS}_1}{\mathrm{S}_1} \right\rangle_{tempStorage} \\ \langle\ \mathrm{M}_1\ \rangle_{Memory} \\ \left\langle \frac{\mathrm{OM}_1}{\mathrm{M}_1} \right\rangle_{tempMemory} \\ \langle\ \mathrm{B}_1\ \rangle_{Balance} \\ \left\langle \frac{\mathrm{OB}_1}{\mathrm{B}_1} \right\rangle_{tempBalance} \end{array} \right\rangle_{contractInstance} \\
\cdots \\
\left\langle \begin{array}{c} \langle\ \mathrm{N}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}\ \rangle_{ctId} \\ \langle\ \mathrm{RhoC}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}\ \rangle_{ctContext} \\ \left\langle \frac{\mathrm{ORhoC}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}}{\mathrm{RhoC}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}} \right\rangle_{tempContext} \\ \langle\ \mathrm{S}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}\ \rangle_{ctStorage} \\ \left\langle \frac{\mathrm{OS}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}}{\mathrm{S}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}} \right\rangle_{tempStorage} \\ \langle\ \mathrm{M}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}\ \rangle_{Memory} \\ \left\langle \frac{\mathrm{OM}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}}{\mathrm{M}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}} \right\rangle_{tempMemory} \\ \langle\ \mathrm{B}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}\ \rangle_{Balance} \\ \left\langle \frac{\mathrm{OB}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}}{\mathrm{B}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}} \right\rangle_{tempBalance} \end{array} \right\rangle_{contractInstance}
\end{array} \right\rangle_{contractInstances}$$

**Figure 13: Definition of $\mathcal{K}_s$ (Part 3).**

RULE Update-Cur-Context

$$\left\langle \frac{\texttt{updateCurContext(CNum:Int,0)}}{\cdot} \cdots \right\rangle_{k}$$

$$\left.\begin{array}{c}
\left\langle \begin{array}{c}
\langle\ \mathrm{N}_0\ \rangle_{\mathit{ctId}} \\
\langle\ \mathrm{RhoC}_0\ \rangle_{\mathit{ctContext}} \\
\left\langle \frac{\mathrm{ORhoC}_0}{\mathrm{RhoC}_0} \right\rangle_{\mathit{tempContext}} \\
\langle\ \mathrm{S}_0\ \rangle_{\mathit{ctStorage}} \\
\left\langle \frac{\mathrm{OS}_0}{\mathrm{S}_0} \right\rangle_{\mathit{tempStorage}} \\
\langle\ \mathrm{M}_0\ \rangle_{\mathit{Memory}} \\
\left\langle \frac{\mathrm{OM}_0}{\mathrm{M}_0} \right\rangle_{\mathit{tempMemory}} \\
\langle\ \mathrm{B}_0\ \rangle_{\mathit{Balance}} \\
\left\langle \frac{\mathrm{OB}_0}{\mathrm{B}_0} \right\rangle_{\mathit{tempBalance}}
\end{array} \right\rangle_{\mathit{contractInstance}} \\
\left\langle \begin{array}{c}
\langle\ \mathrm{N}_1\ \rangle_{\mathit{ctId}} \\
\langle\ \mathrm{RhoC}_1\ \rangle_{\mathit{ctContext}} \\
\left\langle \frac{\mathrm{ORhoC}_1}{\mathrm{RhoC}_1} \right\rangle_{\mathit{tempContext}} \\
\langle\ \mathrm{S}_1\ \rangle_{\mathit{ctStorage}} \\
\left\langle \frac{\mathrm{OS}_1}{\mathrm{S}_1} \right\rangle_{\mathit{tempStorage}} \\
\langle\ \mathrm{M}_1\ \rangle_{\mathit{Memory}} \\
\left\langle \frac{\mathrm{OM}_1}{\mathrm{M}_1} \right\rangle_{\mathit{tempMemory}} \\
\langle\ \mathrm{B}_1\ \rangle_{\mathit{Balance}} \\
\left\langle \frac{\mathrm{OB}_1}{\mathrm{B}_1} \right\rangle_{\mathit{tempBalance}}
\end{array} \right\rangle_{\mathit{contractInstance}} \\
\ldots \\
\left\langle \begin{array}{c}
\langle\ \mathrm{N}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}\ \rangle_{\mathit{ctId}} \\
\langle\ \mathrm{RhoC}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}\ \rangle_{\mathit{ctContext}} \\
\left\langle \frac{\mathrm{ORhoC}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}}{\mathrm{RhoC}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}} \right\rangle_{\mathit{tempContext}} \\
\langle\ \mathrm{S}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}\ \rangle_{\mathit{ctStorage}} \\
\left\langle \frac{\mathrm{OS}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}}{\mathrm{S}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}} \right\rangle_{\mathit{tempStorage}} \\
\langle\ \mathrm{M}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}\ \rangle_{\mathit{Memory}} \\
\left\langle \frac{\mathrm{OM}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}}{\mathrm{M}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}} \right\rangle_{\mathit{tempMemory}} \\
\langle\ \mathrm{B}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}\ \rangle_{\mathit{Balance}} \\
\left\langle \frac{\mathrm{OB}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}}{\mathrm{B}_{\mathrm{CNum}\ -\mathrm{Int}\ 1}} \right\rangle_{\mathit{tempBalance}}
\end{array} \right\rangle_{\mathit{contractInstance}}
\end{array}\right._{\mathit{contractInstances}}$$

RULE Create-Transaction

$$\left\langle \frac{\texttt{createTransaction(L:List)}}{\cdot} \cdots \right\rangle_{k}$$

$$\langle\ \texttt{\#msgInfo(C:Id,R:Id,MsgValue:Int,MsgGas:Int)}\ \rangle_{\mathit{Msg}}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{C}\ \rangle_{\mathit{ctId}} \\ \left\langle \frac{\texttt{B:Int}}{\texttt{B -Int MsgGas}} \right\rangle_{\mathit{Balance}} \end{array} \right\rangle_{\mathit{contractInstance}}$$

$$\left\langle \begin{array}{c} \left\langle \frac{\texttt{CNT:Int}}{\texttt{CNT +Int 1}} \right\rangle_{\mathit{cntTrans}} \\ \left\langle \frac{\texttt{TRAN:Map}}{\texttt{TRAN(CNT |-> "functioncall")}} \right\rangle_{\mathit{tranComputation}} \end{array} \right\rangle_{\mathit{transactions}}$$

RULE Propagate-Exception-True

$$\left\langle \frac{\texttt{propagateException(C:Int,Exception:Bool)}}{\texttt{exception()} \curvearrowright \texttt{propagateException(}\mathrm{C}_1\texttt{,Exception)}} \cdots \right\rangle_{k}$$

$$\left\langle \frac{\texttt{ListItem(C) ListItem(}\mathrm{C}_1\texttt{:Int) ...}}{\texttt{ListItem(}\mathrm{C}_1\texttt{)...}} \right\rangle_{\mathit{contractStack}}$$

`requires Exception ==Bool true`

RULE Propagate-Exception-False

$$\left\langle \frac{\texttt{propagateException(C:Int,Exception:Bool)}}{\cdot} \cdots \right\rangle_{k}$$

`requires Exception ==Bool false`

RULE Clear-Caller-Context

$$\left\langle \frac{\texttt{clearCallerContext(C:Id,Rho:Map)}}{\cdot} \cdots \right\rangle_{k}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{C}\ \rangle_{\mathit{ctId}} \\ \left\langle \frac{\texttt{RhoC:Map}}{\texttt{Rho}} \right\rangle_{\mathit{ctContext}} \end{array} \right\rangle_{\mathit{contractInstance}}$$

RULE Call

$$\left\langle \frac{\texttt{call(N:Int,Es:Values)}}{\begin{array}{c}\texttt{initFunParams(N,Es)} \curvearrowright \\ \texttt{processFunQuantifiers(N)} \curvearrowright \\ \texttt{callFunBody(N)}\end{array}} \cdots \right\rangle_{k}$$

RULE Init-Fun-Params

$$\left\langle \frac{\texttt{initFunParams(N:Id,Es:Values)}}{\begin{array}{c}\texttt{BindParams(Ps,Es)} \curvearrowright \\ \texttt{BindParams(Rs,Es)}\end{array}} \cdots \right\rangle_{k}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{\mathit{fId}}\ \langle\ \texttt{Ps:List}\ \rangle_{\mathit{inputParameters}} \\ \langle\ \texttt{Rs:List}\ \rangle_{\mathit{returnParameters}} \end{array} \right\rangle_{\mathit{function}}$$

RULE Call-Function-Body

$$\left\langle \frac{\texttt{callFunBody(N)}}{\begin{array}{c}\texttt{funBody(B)} \curvearrowright \texttt{updateReturnParams(N)} \curvearrowright \\ \texttt{updateReturnValue(N)}\end{array}} \cdots \right\rangle_{k}$$

$$\langle\ \langle\ \texttt{N}\ \rangle_{\mathit{fId}}\ \langle\ \texttt{B}\ \rangle_{\mathit{Body}}\ \cdots \rangle_{\mathit{function}}$$

RULE Bind-Params

$$\left\langle \frac{\texttt{BindParams(T:EleType P:Id Ps:List,E:value Es:List)}}{\texttt{BindParams(Ps,Es)}} \cdots \right\rangle_{k}$$

$$\left\langle \begin{array}{c}
\langle\ \texttt{N}\ \rangle_{\mathit{ctId}}\ \left\langle \frac{\texttt{Addr}}{\texttt{Addr +Int 1}} \right\rangle_{\mathit{slotNum}} \\
\left\langle \frac{\texttt{CONTEXT:Map}}{\texttt{CONTEXT(P |-> Addr)}} \right\rangle_{\mathit{ctContext}} \\
\left\langle \frac{\texttt{TYPE:Map}}{\texttt{TYPE(P |-> T)}} \right\rangle_{\mathit{ctType}} \\
\left\langle \frac{\texttt{LOCATION:Map}}{\texttt{LOCATION(P |-> String2Id("Local"))}} \right\rangle_{\mathit{ctLocation}} \\
\left\langle \frac{\texttt{MEMORY:Map}}{\texttt{MEMORY(Addr |-> E)}} \right\rangle_{\mathit{Memory}}
\end{array} \right\rangle_{\mathit{contractInstance}}$$

RULE Bind-Params-End

$$\left\langle \frac{\texttt{BindParams(.List,.List)}}{\cdot} \cdots \right\rangle_{k}$$

RULE Function-Body

$$\left\langle \frac{\texttt{funBody(S:Statement Ss:Statements)}}{\texttt{exeStmt(S)} \curvearrowright \texttt{funBody(Ss)}} \cdots \right\rangle_{k}$$

$$\left\langle \frac{\texttt{funBody(.Statements)}}{\cdot} \cdots \right\rangle_{k}$$

RULE r1

$$\left\langle \frac{\texttt{S:Statement Ss:Statements}}{\texttt{exeStmt(S)} \curvearrowright \texttt{Ss}} \cdots \right\rangle_{k}$$

RULE r5

$$\left\langle \frac{\texttt{if (true) S:Statement else S1:Statement}}{\texttt{exeStmt(S)}} \cdots \right\rangle_{k}$$

RULE r6

$$\left\langle \frac{\texttt{if (false) S:Statement else S1:Statement}}{\texttt{exeStmt(S1)}} \cdots \right\rangle_{k}$$

RULE Exe-Statement

$$\left\langle \frac{\texttt{exeStmt(S:NoBlockStatement)}}{\texttt{S}} \cdots \right\rangle_{k}$$

$$\left\langle \begin{array}{c}\texttt{ListItem(\#state(\_,\_,} \\ \texttt{\#return(false,\_),\_,false))}\end{array} \cdots \right\rangle_{\mathit{functionStack}}$$

**Figure 14: Definition of $\mathcal{K}_s$ (Part 4).**

RULE Exe-Statement-Main-Contract

$$\left\langle \frac{\texttt{exeStmt(S:NoBlockStatement)}}{\texttt{S}} \cdots \right\rangle_{k}$$

$$\langle\ \texttt{.List}\ \rangle_{functionStack}$$

RULE Exe-Statement-End

$$\left\langle \frac{\texttt{exeStmt(S:NoBlockStatement)}}{\cdot} \cdots \right\rangle_{k}$$

$$\left\langle \begin{array}{c}\texttt{ListItem(\#state(\_,\_,}\\ \texttt{\#return(ReturnFlag,\_),}\\ \texttt{\_,ExceptionFlag))}\end{array} \cdots \right\rangle_{functionStack}$$

requires (ReturnFlag ==Bool true)

orBool (ExceptionFlag ==Bool true)

RULE Less-GlobalVariables

$$\left\langle \frac{\texttt{X:Id < Y:Id}}{\begin{array}{c}\texttt{gasCal(\#read,String2Id("Global"))} \curvearrowright\\ \texttt{gasCal(\#read,String2Id("Global"))} \curvearrowright\\ \mathrm{V_X}\ \texttt{<Int}\ \mathrm{V_Y}\end{array}} \cdots \right\rangle_{k}$$

$$\langle\ \texttt{ListItem(N:Int)}\ \ldots\rangle_{contractStack}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId}\\ \langle\ldots\ \texttt{X |-> }\mathrm{Addr_X}\texttt{ , Y |-> }\mathrm{Addr_Y}\ \ldots\rangle_{ctContext}\\ \left\langle\cdots \begin{array}{c}\texttt{X |-> String2Id("Global"),}\\ \texttt{Y |-> String2Id("Global")}\end{array}\cdots\right\rangle_{ctLocation}\\ \langle\ldots\ \texttt{X |-> Int , Y |-> Int}\ \ldots\rangle_{ctType}\end{array}\right\rangle_{contractInstance}$$

RULE Less-LocalVariables

$$\left\langle \frac{\texttt{X:Id < Y:Id}}{\begin{array}{c}\texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright\\ \texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright\\ \mathrm{V_X}\ \texttt{<Int}\ \mathrm{V_Y}\end{array}} \cdots \right\rangle_{k}$$

$$\langle\ \texttt{ListItem(N:Int)}\ \ldots\rangle_{contractStack}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId}\\ \langle\ldots\ \texttt{X |-> }\mathrm{Addr_X}\texttt{ , Y |-> }\mathrm{Addr_Y}\ \ldots\rangle_{ctContext}\\ \left\langle\cdots \begin{array}{c}\texttt{X |-> String2Id("Local"),}\\ \texttt{Y |-> String2Id("Local")}\end{array}\cdots\right\rangle_{ctLocation}\\ \langle\ldots\ \texttt{X |-> Int , Y |-> Int}\ \ldots\rangle_{ctType}\end{array}\right\rangle_{contractInstance}$$

RULE Equal-GlobalVariables

$$\left\langle \frac{\texttt{X:Id == Y:Id}}{\begin{array}{c}\texttt{gasCal(\#read,String2Id("Global"))} \curvearrowright\\ \texttt{gasCal(\#read,String2Id("Global"))} \curvearrowright\\ \mathrm{V_X}\ \texttt{==Int}\ \mathrm{V_Y}\end{array}} \cdots \right\rangle_{k}$$

$$\langle\ \texttt{ListItem(N:Int)}\ \ldots\rangle_{contractStack}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId}\\ \langle\ldots\ \texttt{X |-> }\mathrm{Addr_X}\texttt{ , Y |-> }\mathrm{Addr_Y}\ \ldots\rangle_{ctContext}\\ \left\langle\cdots \begin{array}{c}\texttt{X |-> String2Id("Global"),}\\ \texttt{Y |-> String2Id("Global")}\end{array}\cdots\right\rangle_{ctLocation}\\ \langle\ldots\ \texttt{X |-> Int , Y |-> Int}\ \ldots\rangle_{ctType}\end{array}\right\rangle_{contractInstance}$$

RULE Equal-LocalVariables

$$\left\langle \frac{\texttt{X:Id == Y:Id}}{\begin{array}{c}\texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright\\ \texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright\\ \mathrm{V_X}\ \texttt{==Int}\ \mathrm{V_Y}\end{array}} \cdots \right\rangle_{k}$$

$$\langle\ \texttt{ListItem(N:Int)}\ \ldots\rangle_{contractStack}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId}\\ \langle\ldots\ \texttt{X |-> }\mathrm{Addr_X}\texttt{ , Y |-> }\mathrm{Addr_Y}\ \ldots\rangle_{ctContext}\\ \left\langle\cdots \begin{array}{c}\texttt{X |-> String2Id("Local"),}\\ \texttt{Y |-> String2Id("Local")}\end{array}\cdots\right\rangle_{ctLocation}\\ \langle\ldots\ \texttt{X |-> Int , Y |-> Int}\ \ldots\rangle_{ctType}\end{array}\right\rangle_{contractInstance}$$

RULE More-GlobalVariables

$$\left\langle \frac{\texttt{X:Id > Y:Id}}{\begin{array}{c}\texttt{gasCal(\#read,String2Id("Global"))} \curvearrowright\\ \texttt{gasCal(\#read,String2Id("Global"))} \curvearrowright\\ \mathrm{V_X}\ \texttt{>Int}\ \mathrm{V_Y}\end{array}} \cdots \right\rangle_{k}$$

$$\langle\ \texttt{ListItem(N:Int)}\ \ldots\rangle_{contractStack}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId}\\ \langle\ldots\ \texttt{X |-> }\mathrm{Addr_X}\texttt{ , Y |-> }\mathrm{Addr_Y}\ \ldots\rangle_{ctContext}\\ \left\langle\cdots \begin{array}{c}\texttt{X |-> String2Id("Global"),}\\ \texttt{Y |-> String2Id("Global")}\end{array}\cdots\right\rangle_{ctLocation}\\ \langle\ldots\ \texttt{X |-> Int , Y |-> Int}\ \ldots\rangle_{ctType}\end{array}\right\rangle_{contractInstance}$$

RULE More-LocalVariables

$$\left\langle \frac{\texttt{X:Id > Y:Id}}{\begin{array}{c}\texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright\\ \texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright\\ \mathrm{V_X}\ \texttt{>Int}\ \mathrm{V_Y}\end{array}} \cdots \right\rangle_{k}$$

$$\langle\ \texttt{ListItem(N:Int)}\ \ldots\rangle_{contractStack}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId}\\ \langle\ldots\ \texttt{X |-> }\mathrm{Addr_X}\texttt{ , Y |-> }\mathrm{Addr_Y}\ \ldots\rangle_{ctContext}\\ \left\langle\cdots \begin{array}{c}\texttt{X |-> String2Id("Local"),}\\ \texttt{Y |-> String2Id("Local")}\end{array}\cdots\right\rangle_{ctLocation}\\ \langle\ldots\ \texttt{X |-> Int , Y |-> Int}\ \ldots\rangle_{ctType}\end{array}\right\rangle_{contractInstance}$$

RULE Var-Declaration

$$\left\langle \frac{\texttt{T:EleType X:Id = V:Value}}{\texttt{gasCal(\#allocate,String2Id("Local"))} \curvearrowright \texttt{V}} \cdots \right\rangle_{k}$$

$$\langle\ \texttt{ListItem(N:Int)}\ \ldots\rangle_{contractStack}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId}\\ \left\langle \frac{\texttt{Addr}}{\texttt{Addr +Int 1}} \right\rangle_{slotNum}\\ \left\langle \frac{\texttt{MEMORY:Map}}{\texttt{MEMORY(Addr |-> V)}} \right\rangle_{Memory}\\ \left\langle \frac{\texttt{CONTEXT:Map}}{\texttt{CONTEXT(X |-> Addr)}} \right\rangle_{ctContext}\\ \left\langle \frac{\texttt{TYPE:Map}}{\texttt{TYPE(X |-> T)}} \right\rangle_{ctType}\\ \left\langle \frac{\texttt{LOCATION:Map}}{\texttt{LOCATION(X |-> String2Id("Local"))}} \right\rangle_{ctLocation}\end{array}\right\rangle_{contractInstance}$$

RULE Var-Assignment

$$\left\langle \frac{\texttt{X:Id = V:Value}}{\texttt{gasCal(\#write,String2Id("Global"),OV,V)} \curvearrowright \texttt{V}} \cdots \right\rangle_{k}$$

$$\langle\ \texttt{ListItem(N:Int)}\ \ldots\rangle_{contractStack}$$

$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId}\\ \langle\ldots\ \texttt{X |-> Addr}\ \ldots\rangle_{ctContext}\\ \langle\ldots\ \texttt{X |-> String2Id("Global")}\ \ldots\rangle_{ctLocation}\\ \left\langle\cdots \frac{\texttt{Addr |-> OV}}{\texttt{Addr |-> V}} \cdots\right\rangle_{ctStorage}\end{array}\right\rangle_{contractInstance}$$

RULE Return-Value

$$\left\langle \frac{\texttt{return E:Value}}{\texttt{1}} \cdots \right\rangle_{k}$$

$$\left\langle \frac{\begin{array}{c}\texttt{ListItem(\#state(\_,\_,}\\ \texttt{\#return(\_,\_),\_,\_))}\end{array}}{\begin{array}{c}\texttt{ListItem(\#state(\_,\_,}\\ \texttt{\#return(true,E),\_,\_))}\end{array}} \cdots \right\rangle_{functionStack}$$

RULE Return

$$\left\langle \frac{\texttt{return}}{\texttt{1}} \cdots \right\rangle_{k}$$

$$\left\langle \frac{\begin{array}{c}\texttt{ListItem(\#state(\_,\_,}\\ \texttt{\#return(\_,\_),\_,\_))}\end{array}}{\begin{array}{c}\texttt{ListItem(\#state(\_,\_,}\\ \texttt{\#return(true,true),\_,\_))}\end{array}} \cdots \right\rangle_{functionStack}$$

**Figure 15: Definition of $\mathcal{K}_s$ (Part 5).**

RULE GAS-CAL

```
⟨ gasCal(X:Id,Y:Id)
  ─────────────────
          ·            ...⟩k  ⟨ #msgInfo(_,_,_,GasLimit) ⟩Msg
⟨   ListItem(G:Int) GasList:List
  ─────────────────────────────────
  ListItem(G +Int G1:Int) GasList  ⟩gasStack
⟨     G
  ──────────
  G +Int G1 ⟩gasConsumption
requires G +Int G1 <=Int GasLimit
```

RULE ADD-GLOBALVARIABLES

```
⟨            X:Id = Y:Id + Z:Id
  ──────────────────────────────────────────────
   gasCal(#read,String2Id("Global")) ↷
   gasCal(#read,String2Id("Global")) ↷              ...⟩k
  gasCal(#write,String2Id("Global"),OV_X,
        V_Y  +Int V_Z) ↷ V_Y  +Int V_Z
⟨ ListItem(N:Int)  ...⟩contractStack
⟨                     ⟨ N ⟩ctId
  ⟨...   X |-> Addr_X , Y |-> Addr_Y ,
               Z |-> Addr_Z                 ...⟩ctContext
  ⟨...   X |-> String2Id("Global") ,
         Y |-> String2Id("Global") ,        ...⟩ctLocation
          Z |-> String2Id("Global")
  ⟨...  X |-> Int , Y |-> Int , Z |-> Int  ...⟩ctType
  ⟨           ... , Addr_X |-> OV_X ,
      Addr_Y |-> V_Y , Addr_Z |-> V_Z , ...
    ─────────────────────────────────────────
       ... , Addr_X |-> V_Y +Int V_Z ,
            Addr_Y |-> V_Y , ...             ⟩ctStorage
                                              ⟩contractInstance
```

RULE ADD-LOCALVARIABLES

```
⟨            X:Id = Y:Id + Z:Id
  ──────────────────────────────────────────────
   gasCal(#read,String2Id("Local")) ↷
   gasCal(#read,String2Id("Local")) ↷               ...⟩k
  gasCal(#write,String2Id("Local"),OV_X,
        V_Y  +Int V_Z) ↷ V_Y  +Int V_Z
⟨ ListItem(N:Int)  ...⟩contractStack
⟨                     ⟨ N ⟩ctId
  ⟨...   X |-> Addr_X , Y |-> Addr_Y ,
               Z |-> Addr_Z                 ...⟩ctContext
  ⟨...   X |-> String2Id("Local") ,
         Y |-> String2Id("Local") ,         ...⟩ctLocation
          Z |-> String2Id("Local")
  ⟨...  X |-> Int , Y |-> Int , Z |-> Int  ...⟩ctType
  ⟨           ... , Addr_X |-> OV_X ,
      Addr_Y |-> V_Y , Addr_Z |-> V_Z , ...
    ─────────────────────────────────────────
       ... , Addr_X |-> V_Y +Int V_Z ,
            Addr_Y |-> V_Y , ...             ⟩Memory
                                              ⟩contractInstance
```

RULE SUB-GLOBALVARIABLES

```
⟨            X:Id = Y:Id - Z:Id
  ──────────────────────────────────────────────
   gasCal(#read,String2Id("Global")) ↷
   gasCal(#read,String2Id("Global")) ↷              ...⟩k
  gasCal(#write,String2Id("Global"),OV_X,
        V_Y  −Int V_Z) ↷ V_Y  −Int V_Z
⟨ ListItem(N:Int)  ...⟩contractStack
⟨                     ⟨ N ⟩ctId
  ⟨...   X |-> Addr_X , Y |-> Addr_Y ,
               Z |-> Addr_Z                 ...⟩ctContext
  ⟨...   X |-> String2Id("Global") ,
         Y |-> String2Id("Global") ,        ...⟩ctLocation
          Z |-> String2Id("Global")
  ⟨...  X |-> Int , Y |-> Int , Z |-> Int  ...⟩ctType
  ⟨           ... , Addr_X |-> OV_X ,
      Addr_Y |-> V_Y , Addr_Z |-> V_Z , ...
    ─────────────────────────────────────────
       ... , Addr_X |-> V_Y −Int V_Z ,
            Addr_Y |-> V_Y , ...             ⟩ctStorage
                                              ⟩contractInstance
```

RULE GAS-CAL-FAIL

```
⟨ gasCal(X:Id,Y:Id)
  ─────────────────
     exception()       ...⟩k  ⟨ #msgInfo(_,_,_,GasLimit) ⟩Msg
⟨      G
  ──────────────
  G +Int G1:Int ⟩gasConsumption
requires G +Int G1 >Int GasLimit
```

RULE SUB-LOCALVARIABLES

```
⟨            X:Id = Y:Id - Z:Id
  ──────────────────────────────────────────────
   gasCal(#read,String2Id("Local")) ↷
   gasCal(#read,String2Id("Local")) ↷               ...⟩k
  gasCal(#write,String2Id("Local"),OV_X,
        V_Y  −Int V_Z) ↷ V_Y  −Int V_Z
⟨ ListItem(N:Int)  ...⟩contractStack
⟨                     ⟨ N ⟩ctId
  ⟨...   X |-> Addr_X , Y |-> Addr_Y ,
               Z |-> Addr_Z                 ...⟩ctContext
  ⟨...   X |-> String2Id("Local") ,
         Y |-> String2Id("Local") ,         ...⟩ctLocation
          Z |-> String2Id("Local")
  ⟨...  X |-> Int , Y |-> Int , Z |-> Int  ...⟩ctType
  ⟨           ... , Addr_X |-> OV_X ,
      Addr_Y |-> V_Y , Addr_Z |-> V_Z , ...
    ─────────────────────────────────────────
       ... , Addr_X |-> V_Y −Int V_Z ,
            Addr_Y |-> V_Y , ...             ⟩Memory
                                              ⟩contractInstance
```

RULE MUL-GLOBALVARIABLES

```
⟨            X:Id = Y:Id * Z:Id
  ──────────────────────────────────────────────
   gasCal(#read,String2Id("Global")) ↷
   gasCal(#read,String2Id("Global")) ↷              ...⟩k
  gasCal(#write,String2Id("Global"),OV_X,
        V_Y  ∗Int V_Z) ↷ V_Y  ∗Int V_Z
⟨ ListItem(N:Int)  ...⟩contractStack
⟨                     ⟨ N ⟩ctId
  ⟨...   X |-> Addr_X , Y |-> Addr_Y ,
               Z |-> Addr_Z                 ...⟩ctContext
  ⟨...   X |-> String2Id("Global") ,
         Y |-> String2Id("Global") ,        ...⟩ctLocation
          Z |-> String2Id("Global")
  ⟨...  X |-> Int , Y |-> Int , Z |-> Int  ...⟩ctType
  ⟨           ... , Addr_X |-> OV_X ,
      Addr_Y |-> V_Y , Addr_Z |-> V_Z , ...
    ─────────────────────────────────────────
       ... , Addr_X |-> V_Y ∗Int V_Z ,
            Addr_Y |-> V_Y , ...             ⟩ctStorage
                                              ⟩contractInstance
```

RULE MUL-LOCALVARIABLES

```
⟨            X:Id = Y:Id * Z:Id
  ──────────────────────────────────────────────
   gasCal(#read,String2Id("Local")) ↷
   gasCal(#read,String2Id("Local")) ↷               ...⟩k
  gasCal(#write,String2Id("Local"),OV_X,
        V_Y  ∗Int V_Z) ↷ V_Y  ∗Int V_Z
⟨ ListItem(N:Int)  ...⟩contractStack
⟨                     ⟨ N ⟩ctId
  ⟨...   X |-> Addr_X , Y |-> Addr_Y ,
               Z |-> Addr_Z                 ...⟩ctContext
  ⟨...   X |-> String2Id("Local") ,
         Y |-> String2Id("Local") ,         ...⟩ctLocation
          Z |-> String2Id("Local")
  ⟨...  X |-> Int , Y |-> Int , Z |-> Int  ...⟩ctType
  ⟨           ... , Addr_X |-> OV_X ,
      Addr_Y |-> V_Y , Addr_Z |-> V_Z , ...
    ─────────────────────────────────────────
       ... , Addr_X |-> V_Y ∗Int V_Z ,
            Addr_Y |-> V_Y , ...             ⟩Memory
                                              ⟩contractInstance
```

**Figure 16: Definition of $\mathcal{K}_s$ (Part 6).**

RULE Div-GlobalVariables

$$\left\langle \frac{\texttt{X:Id = Y:Id / Z:Id}}{\begin{array}{c}\texttt{gasCal(\#read,String2Id("Global"))} \curvearrowright \\ \texttt{gasCal(\#read,String2Id("Global"))} \curvearrowright \\ \texttt{gasCal(\#write,String2Id("Global"),}\mathrm{OV_X}\texttt{,} \\ \mathrm{V_Y}\ \mathrm{/Int}\ \mathrm{V_Z}\texttt{)} \curvearrowright \mathrm{V_Y}\ \mathrm{/Int}\ \mathrm{V_Z}\end{array}} \ \ldots \right\rangle_{k}$$

$$\left\langle \ \texttt{ListItem(N:Int)}\ \ldots \right\rangle_{contractStack}$$

$$\left\langle \begin{array}{c} \langle \ \texttt{N} \ \rangle_{ctId} \\ \left\langle \ldots \begin{array}{c}\texttt{X |->}\ \mathrm{Addr_X}\ \texttt{, Y |->}\ \mathrm{Addr_Y}\ \texttt{,} \\ \texttt{Z |->}\ \mathrm{Addr_Z}\end{array} \ldots \right\rangle_{ctContext} \\ \left\langle \ldots \begin{array}{c}\texttt{X |-> String2Id("Global") ,} \\ \texttt{Y |-> String2Id("Global") ,} \\ \texttt{Z |-> String2Id("Global")}\end{array} \ldots \right\rangle_{ctLocation} \\ \left\langle \ldots \ \texttt{X |-> Int , Y |-> Int , Z |-> Int} \ \ldots \right\rangle_{ctType} \\ \left\langle \frac{\begin{array}{c}\ldots\texttt{ ,}\ \mathrm{Addr_X}\ \texttt{|->}\ \mathrm{OV_X}\ \texttt{,} \\ \mathrm{Addr_Y}\ \texttt{|->}\ \mathrm{V_Y}\ \texttt{,}\ \mathrm{Addr_Z}\ \texttt{|->}\ \mathrm{V_Z}\ \texttt{,} \ldots\end{array}}{\begin{array}{c}\ldots\texttt{ ,}\ \mathrm{Addr_X}\ \texttt{|->}\ \mathrm{V_Y}\ \mathrm{/Int}\ \mathrm{V_Z}\ \texttt{,} \\ \mathrm{Addr_Y}\ \texttt{|->}\ \mathrm{V_Y}\ \texttt{,} \ldots\end{array}} \right\rangle_{ctStorage} \end{array} \right\rangle_{contractInstance}$$

RULE Div-LocalVariables

$$\left\langle \frac{\texttt{X:Id = Y:Id / Z:Id}}{\begin{array}{c}\texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright \\ \texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright \\ \texttt{gasCal(\#write,String2Id("Local"),}\mathrm{OV_X}\texttt{,} \\ \mathrm{V_Y}\ \mathrm{/Int}\ \mathrm{V_Z}\texttt{)} \curvearrowright \mathrm{V_Y}\ \mathrm{/Int}\ \mathrm{V_Z}\end{array}} \ \ldots \right\rangle_{k}$$

$$\left\langle \ \texttt{ListItem(N:Int)}\ \ldots \right\rangle_{contractStack}$$

$$\left\langle \begin{array}{c} \langle \ \texttt{N} \ \rangle_{ctId} \\ \left\langle \ldots \begin{array}{c}\texttt{X |->}\ \mathrm{Addr_X}\ \texttt{, Y |->}\ \mathrm{Addr_Y}\ \texttt{,} \\ \texttt{Z |->}\ \mathrm{Addr_Z}\end{array} \ldots \right\rangle_{ctContext} \\ \left\langle \ldots \begin{array}{c}\texttt{X |-> String2Id("Local") ,} \\ \texttt{Y |-> String2Id("Local") ,} \\ \texttt{Z |-> String2Id("Local")}\end{array} \ldots \right\rangle_{ctLocation} \\ \left\langle \ldots \ \texttt{X |-> Int , Y |-> Int , Z |-> Int} \ \ldots \right\rangle_{ctType} \\ \left\langle \frac{\begin{array}{c}\ldots\texttt{ ,}\ \mathrm{Addr_X}\ \texttt{|->}\ \mathrm{OV_X}\ \texttt{,} \\ \mathrm{Addr_Y}\ \texttt{|->}\ \mathrm{V_Y}\ \texttt{,}\ \mathrm{Addr_Z}\ \texttt{|->}\ \mathrm{V_Z}\ \texttt{,} \ldots\end{array}}{\begin{array}{c}\ldots\texttt{ ,}\ \mathrm{Addr_X}\ \texttt{|->}\ \mathrm{V_Y}\ \mathrm{/Int}\ \mathrm{V_Z}\ \texttt{,} \\ \mathrm{Addr_Y}\ \texttt{|->}\ \mathrm{V_Y}\ \texttt{,} \ldots\end{array}} \right\rangle_{Memory} \end{array} \right\rangle_{contractInstance}$$

**Figure 17: Definition of $\mathcal{K}_s$ (Part 7).**