Scar: Verification-Based Development of Smart Contracts

Jonas Schiffl[1]
KASTEL - Institute of Information Security and Dependability, Karlsruhe Institute of Technology, Germany

Bernhard Beckert
KASTEL - Institute of Information Security and Dependability, Karlsruhe Institute of Technology, Germany

Abstract

Compared to other kinds of computer programs, smart contracts have some unique characteristics, *e.g.*, immutability, and the public availability of source code. This means that any vulnerability has a large probability of being exploited. Since smart contracts cannot be patched, it is very important that smart contracts are correct and secure upon deployment. While much research has been invested in this goal, smart contract correctness and security remains a challenging problem.

In this paper, we present the Scar approach for model-driven development of correct and secure smart contract applications. Before implementing an application, smart contract developers first describe it in terms of an intuitive, platform-agnostic metamodel. Within this model, they can also specify high-level security and behavioral correctness properties, and check whether the model contains any inconsistencies. Finally, a combination of code generation, static analyses, and formal verification of automatically generated formal annotations leads to an implementation that is correct and secure w.r.t. the initial model.

2012 ACM Subject Classification Software and its engineering → Formal methods

Keywords and phrases Smart Contracts, Formal Verification, Security, Safety and Liveness

Digital Object Identifier 10.4230/OASIcs.FMBC.2025.14

Category Tool Paper

Funding This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs.

1 Introduction

Static analysis and formal verification of smart contracts have been very active fields of research. Numerous approaches for detecting bugs and common weaknesses as well as specifying and verifying user-defined correctness properties have been proposed; a recent overview lists 194 works [9]. However, smart contract security is still a problem. There are entire websites dedicated to listing all the instances of smart contract applications being hacked (for example, [16] and [4]). Many of the exploited applications were even audited, prior to the attacks, by companies like CertiK [1], who perform audits consisting of both manual code review and automated static analysis. This indicates that the methods used in the audits are not sufficient to guarantee security.

Although static analysis tools may help to avoid common vulnerabilities, current research shows that they are not sufficient to ensure smart contract application security. For example, Zhang et. al. come to the conclusion that the majority of bugs in smart contracts is "hard to find" and "not machine-auditable" [17]. Only 8% of the vulnerabilities which led to these

[1] Corresponding author

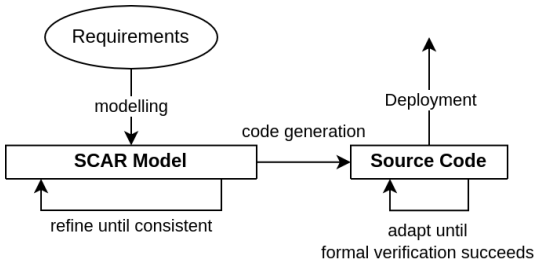

Figure 1 The Scar process of smart contract application development.

attacks could have been detected with *any* state-of-the-art static analysis tool [2]. Moreover, most current vulnerability detection tools "can only detect vulnerabilities in a single and old version of smart contracts" [6]. Formal verification tools go a different route by allowing their users to define the desired properties, and attempting a proof. Unlike with vulnerability detection tools, there is not always a yes or no answer; a proof attempt can always time out. If a proof is found, the program is deemed to fulfill its specification. Examples for formal verification approaches include solc-verify [5], Celestial [3], or VerX [10].

One difficulty with formal verification on the source code level is a lack of abstraction. Some properties, *e.g.*, resource ownership, are easy to grasp for developers, but hard to specify in terms of invariants or pre- and postconditions. To allow more intuitive specification of such properties, model-driven development approaches have been proposed, such as FSolidM [7] or SmartPulse [15]. These approaches target specific platforms and specific properties (*e.g.*, SmartPulse targets temporal properties of Solidity smart contracts).

In this paper, we present Scar, a highly general approach for modeling smart contract applications. The Scar approach consists of three main parts:

- A **metamodel of smart contract applications**, in which these applications can be defined in terms of their structure, *i.e.*, the contracts, state, and functions they consist of. A type system is also part of the metamodel.
- The **ScarML specification language**, which serves to describe the behavior of individual functions in terms of pre- and postconditions, and properties of individual contracts in terms of invariants.
- A **model-driven process** based on formal verification, which leads from the creation of a Scar model to a source code implementation that can be deployed on a specific platform. Formal methods ensure that the implementation is consistent with the model.

This paper is structured as follows: In Section 2, we describe the structure and the semantics of the Scar metamodel and the ScarML specification language.Section 5 gives an overview of the applications of Scar, of its Scala implementation, and about our evaluation. Section 6 conludes.

2 The Scar Approach

In this section, we present Scar, a modeling and analysis approach for smart contract applications.

An overview of the intended workflow with the Scar platform is given in Figure 1. After collecting the requirements for an application, developers create a basic Scar model. This basic model defines an application in terms of its smart contracts, their state variables and functions, and the behavior of these functions. The developer then proceeds in two directions:

First, they specify application-level correctness and security properties on the model. They conduct the analyses provided by Scar (or by the integration of Scar with other tools) to verify that these properties are consistent with the basic model. Secondly, they use the code generation capabilities provided by Scar to automatically create a source code skeleton in the desired target programming language, annotated with specification in the language of a formal verification tool suitable for that language. They proceed to implement the functions such that the verification tool is able to prove that the implementation is correct w.r.t. the generated specification. This process is inspired by the Design by Contract paradigm [8].

Figure 2 gives an overview of the **basic metamodel**. The topmost element of the metamodel is the *application*. It consists of *contract* elements. Contracts, in turn consist of *stateVariable* and *function* elements. State variables have a name and a type. Functions have a name, a list of named and typed parameters, an optional return type, and a function contract written in the ScarML functional specification language (see Section 3 below).

In order to describe the initialization of an application, an initial condition can be specified for each contract in the same specification language. To closely reflect the constructors in smart contract programming languages, the initial condition can be parameterized. The full grammar for expressing a Scar model in text is given in Figure 9.

The Scar type system has four primitive types: Booleans (*Bool*), signed and unsigned integers (*Int* and *UInt*, respectively), and strings (*String*). Accounts (*Account*) can be either external accounts, representing a real-world entity, or contracts. Both have a non-negative balance, and every account has a unique ID. Each contract in an application defines its own type, which consists of a name, a list of typed and named state variables, and a list of the functions of that contract. Furthermore, there are two composite types: Arrays and Mappings, which map keys of a primitive type to values (whose type is not restricted). In addition, there are two user-defined types: Enums have a String list of constant names. Structs are records with a list of named and typed fields. The type system is visualized in Figure 5.

For specifying the behavior of an application, we introduce **ScarML**, the Scar modeling language. ScarML is a specification language for contract invariants (for specifying properties of smart contract instances) and function contracts (for specifying the behavior of single functions). Function contracts consist of preconditions, postconditions, and frame conditions. Invariants, preconditions, and postconditions are defined as Boolean *specification expressions*. Frame conditions are specified as *frame expressions*.

The full grammar of ScarML specification expressions is defined in Figure 7. Expressions are typed. Terms of Boolean type can be negated and combined with the standard Boolean connectors. Existentially and universally quantified expressions are also allowed. There are two equality operators, reflecting value equality and referential equality.

There is a special operator for evaluating a term in the pre-state of a transaction (`\old`). Furthermore, there are some constructs that represent the context of a transaction, *i.e.*, the caller, the amount transferred with the call, the system time of the call, and the number of the block in which the transaction is executed. Another domain-specific term is the `\hash` operator. Furthermore, the specification can contain terms representing a (pure) function call. All Boolean, integer, and string literals are also terms. Terms of Boolean type can be used as top-level expressions in invariants, preconditions and postconditions.

3 Semantics

We define the **semantics of ScarML** in terms of an evaluation function over the state. Invariants and pre- and postconditions are evaluated in a *step* τ_i of the execution, which consists of the call context ctx_i and a state s_i (cf. the description of the trace semantics below). The **partial evaluation function** $[\![\cdot]\!]$ maps terms of the ScarML specification language to the set of all possible values. It is parameterized with the step τ_i in which it occurs. As an example, the evaluation of a variable is that variable's state in the context where it is evaluated: $[\![v]\!] := s_i(v)$. The evaluation of mathematical and logical operators is usually just the operation, conducted on the evaluation of the subterms (*e.g.*, $[\![\phi_1 + \phi_2]\!] := [\![\phi_1]\!] + [\![\phi_2]\!]$). The full definition of $[\![\cdot]\!]$ is given in Figure 8.

We say that a condition c is **fulfilled** (or satisfied) in a step τ_i if $[\![c]\!]_{\tau_i} = \top$. A condition c is **satisfiable** if there exists a possible step (*i.e.*, a combination of a state s and a call environment) τ such that c is fulfilled in τ.

As for the **semantics of a Scar model**, we say that a Scar model m is **plausible** if

- all contract invariants are satisfiable, and
- all function pre- and postconditions are satisfiable, and
- the initial conditions of each contract type in m are satisfiable, and
- for each contract type in m, the initial conditions of m imply each contract invariant, *i.e.*, every possible state which satisfies the initial conditions must also satisfy every contract invariant.

For a given Scar model, an execution of the application is an infinite trace of steps $\tau_0, \tau_1, \tau_2, ...$ where each step τ_i is either an *environment step* or an *application step*.

The **initial step** τ_0 is defined in the `appInit` part of the application specification. In it, the initial set of contracts is declared, along with a list of parameters according to the initial parameter types that each contract requires. After the initial step, the set of known contracts $\mathcal{C}$ consists of all contracts declared in `appInit`, as well as all contract-typed locations within these (*i.e.*, state variables of contract type, as well as contract-typed elements of arrays, mappings, and structs).

Similarly, the set of known external accounts is initialized with all account-typed state variables and account-typed elements of arrays, mappings, and structs.

For τ_0 to be valid, the initial condition of each contract in $\mathcal{C}$ must be fulfilled.

An **environment step** characterizes a step in which no function of the application is called, and in which the application state does not change, with the possible exception of account balances. Furthermore, block number and system time can increase.

Formally, an environment step tau_i consists of ctx_i, the context of the step (which, in turn, consists of $systime_i$, the system time of the step, and $blocknum_i$, the block number of the step), and s_i, the state of the application after τ_i.

For an environment step τ_i to be **valid**, the following conditions must hold:

- $blocknum_i = blocknum_{i-1} \lor blocknum_i = blocknum_{i-1} + 1$
- $systime_i = systime_{i-1}$ if $blocknum_i = blocknum_{i-1}$
- $systime_i > systime_{i-1}$ if $blocknum_i = blocknum_{i-1} + 1$
- $\forall\, l \in Locs : \begin{cases} s_i >= s_{i-1}, \text{l is balance} \\ s_i = s_{i-1}, \text{else} \end{cases}$

An **application step** is the result of a call to one of the application's functions. Formally, an application step τ_i consists of ctx_i and s_i as above, as well as f_i, the function that was called to reach τ_i. f_i, in turn, consists of $caller_i$, the caller of the function; amt_i, the amount

transferred with the function call; $params_i$, the call parameters; pre_i, the precondition of f_i; $post_i$, the postcondition of f_i; $return_i$, the return value of f_i; and mod_i, the set of locations f_i may modify.

For an application step τ_i to be **valid**, the following conditions must hold:

- $blocknum_i = blocknum_{i-1} \vee blocknum_i = blocknum_{i-1} + 1$
- $systime_i = systime_{i-1}$ if $blocknum_i = blocknum_{i-1}$
- $systime_i > systime_{i-1}$ if $blocknum_i = blocknum_{i-1} + 1$
- pre_i must be fulfilled in τ_{i-1}
- $post_i$ must be fulfilled in τ_i
- $\forall\, l \in Locs \setminus mod_i : \begin{cases} s_i >= s_{i-1}, l \text{ is balance} \\ s_i = s_{i-1}, \text{else} \end{cases}$

With this, we now define the semantics $\mathcal{S}$ of a Scar model as the set of all infinite traces of valid steps: For a plausible Scar model m, the semantics $\mathcal{S}(m)$ is the set of traces $\{\mathrm{T} := \tau_0, \tau_1, \tau_2, ...\}$ where τ_0 is a valid initial step for m, and each $\tau_{i \in \mathcal{N}_{>0}}$ is either a valid environment step or a valid application step. A visualization of the Scar trace semantics is presented in Figure 4.

4 Ensuring Consistency between Model and Code

In order to deploy an application, a model needs to be converted into platform-specific source code. The main point of our model-driven approach is that properties that were proven to hold for the model also hold for the implementation. Therefore, the conversion from model to code needs to be semantics-preserving: The implementation must be a refinement of the model. This means that for every possible execution of the implementation, when deployed, there must be a corresponding trace in the model semantics.

To show that this is the case, the semantics of the platform and of the verification tool have to be taken into account, and it has to be shown that each execution of a deployed application corresponds to an execution trace of its model. This argument has to be made separately for each target platform.

For the Solidity programming language, code is generated as follows from a Scar model: First, a Solidity version header is prepended to the file. Then, a library contract `UTIL` containing the user-defined data types is created. Scar structs and enums are translated to their respective Solidity counterparts. For every contract in the Scar model, a corresponding Solidity `contract` of the same name is created. Since Scar was written with Solidity as the main target language, each Scar type has a directly corresponding Solidity type, making the translation of state variables, function parameters, and return types straightforward. For functions, only headers are generated, which consist of the function name, the translated function parameters, the `public` modifier indicating that the function can be called from everywhere, as well as the return type.

For verification, we use the solc-verify tool [5] for formal verification of Solidity smart contracts. As in Scar, the main specification elements in solc-verify are invariants and function contracts.

The differences between Scar and Ethereum/Solidity are mainly in the area of data types. Scar supports fewer types than Solidity. Therefore, the Solidity implementation has to be limited to the variables that actually occur in the model to ensure consistency.

However, the Scar types contain mathematical integers, which are not present in Solidity; the Scar type system is not a subset of the Solidity type system. Therefore, care has to be taken to avoid situations where this discrepancy leads to a violation of consistency. To give

an example, this could happen if a precondition is not satisfiable over mathematical integers, but satisfiable over Solidity's machine integers. This would lead to a situation where the function could never occur in the semantics of the model, but it could be called successfully in the implementation. Thus, the implementation would allow more behaviors than the model. This needs to be avoided by the translation, or by additional measures or precautions developers have to take.

In case of the Scar to Solidity code and annotation generation we developed, the implementation is consistent if the generated source code is not changed except for implementing the generated functions and adding helper functions (*i.e.*, functions that cannot be called from outside their contract and which do not change the state in any way). Furthermore, the generated function preconditions and contract invariants must not contain arithmetic overflows, and the implementation must be proven correct by SOLC-VERIFY against the generated specification (function contracts and contract invariants).

With this, it is guaranteed that the original model was syntactically correct, and that the application state cannot be changed outside of the behavior specified in the model. Thus, every execution of the implementation is also a trace of the model: Model and implementation are consistent.

5 Application and Evaluation

Scar allows specification and verification of application-level properties of smart contracts. These properties are difficult to specify on the source code level. On the level of a Scar model, however, the relevant specification constructs can be easily introduced in a way that keeps specification concise and intuitive. We have previously developed two approaches for specifying and verifying application-level properties, namely, temporal properties, and access control policies.

In [12], we developed a specification language for temporal properties of smart contract applications. The properties can be specified and verified directly on the model level. If there is an implementation consistent with the model (as described in Section 4), then it also fulfills the temporal properties specified on the model.

In [13], we developed a security approach for smart contract applications, based on the notion of actors and capabilities. Using this approach in Scar, developers can define roles and capabilities on the model level. Capabilities include calling a certain functionality, changing the state of an application, and transferring currency. Our approach makes it possible to specify security policies on the abstract model level, which matches the intuition much closer than the source code level. As with temporal properties, the adherence of a model to a given security specification can be verified purely on the model level, given that the implementation is consistent (in the above sense) with the model.

Scar has been implemented in the Scala programming language. The repository [11] is publicly available. The project's core is the `model` package, in which all model elements are defined as Scala classes. The top-level model element is the smart contract application `SCApp`. The syntax of the `.scar` model is defined in an ANTLR grammar. A simplified overview grammar for the basic metamodel is shown in Figure 9. First, the user-defined types are specified, followed by the initial configuration of the application. Then, the contracts are declared with their state variables, initial conditions, invariants, and functions (including function contracts).

We evaluated the Scar approach on a set of example applications, including a bank, an escrow, an auction, and a casino contract, as well as the more complex Palinodia application [14]. While this does not constitute a full evaluation, some conclusions can still be drawn:

First of all, Scar is general enough: Its data types and structural model elements are sufficient to model all desired applications and their basic functionality.

Second, even though the set of examples is not large, almost all syntactic elements of the Scar specification language appear in at least one of the examples. From this, it can be concluded that the modeling language is not overly complex.

Another metric for assessing the quality of a metamodel is how succinct the models are, compared to some reference. In Figure 12, the length of the Scar models of the examples described in this section is compared to the length of the corresponding Solidity code. This comparison should be taken with a grain of salt; it is easy to change the relative length by leaving out or specifying additional properties in the Scar model.

When comparing the length of the Scar models and the Solidity source code the models are based on, it can be observed that the models are always shorter, but often not by much. For very simple functionality, such as assignments or transfers, the Scar specification and the corresponding Solidity code are practically identical. In such cases, Scar's approach leads to a duplication, and might increase the workload of a developer. On the other hand, functions with such an easy specification can also be implemented without much effort, and in the future, Scar will incorporate functionality for automatic program synthesis, or correct-by-construction code generation.

Scar's benefits become more obvious when the modeled functionality becomes more complex. This includes properties of array and mapping data types, which can be succinctly specified in Scar. Another recurring case was the use of an array as an index data structure to a mapping, which happens both in the simple bank example and in the real-world Palinodia application. Implementing this functionality is rather complex, requiring tens of lines of code. The abstract property describing the intended behavior, however, is very simple to describe in both cases, and this is reflected in the succinctness of the Scar model.

As for verification, we modeled all applications in Scar, generated a Solidity code skeleton from the model, and attempted to implement it so that verification with the solc-verify tool succeeded. This was possible in most cases. Failures include functionality which relies on hashing; while our implementation is likely correct, solc-verify is unable to reason about the Keccak hash function used in Solidity. In our examples, the implementation did not require auxiliary specification such as loop invariants.

6 Conclusion and Future Work

In this paper, we presented the Scar approach for model-driven development of correct and secure smart contract applications. Our approach enables developers to model an application and its behavior in an abstract, intuitive manner. At the same time, it provides a process, based on formal verification, which guarantees that the implementation is consistent with the model.

There are several future research directions. One is the application of the Scar approach to other platforms, languages, and verification tools. We are currently developing methods to apply Scar to Rust smart contracts written for the Solana platform.

Other possible research directions include simulation, and generating runtime checks. Scar's state transition semantics make it very suitable for the execution with random parameters, and the relevant parts of the application state can easily be highlighted for visualization. This can help developers to quickly build an intuition whether their model actually captures their intention. In cases where formal verification is not successful, runtime checks can be generated from ScarML specification, to ensure that the application fails gracefully in the presence of programming errors.

References

1 CertiK. CertiK - Securing The Web3 World, 2024. URL: https://www.certik.com.

2 Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Ben Livshits. Smart Contract and DeFi Security: Insights from Tool Evaluations and Practitioner Surveys, April 2023. doi:10.48550/arXiv.2304.02981.

3 Samvid Dharanikota, Suvam Mukherjee, Chandrika Bhardwaj, Aseem Rastogi, and Akash Lal. Celestial: A Smart Contracts Verification Framework. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 133–142, October 2021. doi:10.34727/2021/ISBN.978-3-85448-046-4_22.

4 David Gerard. Attack of the 50 Foot Blockchain, June 2024. URL: https://davidgerard.co.uk/blockchain/.

5 Akos Hajdu and Dejan Jovanovic. Solc-verify: A Modular Verifier for Solidity Smart Contracts. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments*, pages 161–179, Cham, 2020. Springer International Publishing.

6 Daojing He, Rui Wu, Xinji Li, Sammy Chan, and Mohsen Guizani. Detection of Vulnerabilities of Blockchain Smart Contracts. *IEEE Internet of Things Journal*, pages 1–1, 2023.

7 Anastasia Mavridou and Aron Laszka. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security*, pages 523–540, Berlin, Heidelberg, 2018. Springer. doi:10.1007/978-3-662-58387-6_28.

8 B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, October 1992.

9 Sundas Munir and Walid Taha. Pre-deployment Analysis of Smart Contracts – A Survey, January 2023. doi:10.48550/arXiv.2301.06079.

10 Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677, May 2020. doi:10.1109/SP40000.2020.00024.

11 Jonas Schiffl. Jonas Schiffl / Scar · GitLab, June 2024. URL: https://gitlab.kit.edu/jonas.schiffl/Scar.

12 Jonas Schiffl and Bernhard Beckert. A Practical Notion of Liveness in Smart Contract Applications. In *OASIcs FMBC 2024*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/OASICS.FMBC.2024.8.

13 Jonas Schiffl, Alexander Weigl, and Bernhard Beckert. Static Capability-based Security for Smart Contracts. In *2023 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, Athens, July 2023.

14 Oliver Stengele, Andreas Baumeister, Pascal Birnstill, and Hannes Hartenstein. Access Control for Binary Integrity Protection using Ethereum. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*, SACMAT '19, pages 3–12, New York, NY, USA, May 2019. Association for Computing Machinery. doi:10.1145/3322431.3325108.

15 Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 555–571, May 2021. doi:10.1109/SP40001.2021.00085.

16 Molly White. Web3 is Going Just Great, February 2024. URL: https://web3isgoinggreat.com/.

17 Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying Exploitable Bugs in Smart Contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 615–627, May 2023. doi:10.1109/ICSE48619.2023.00061.

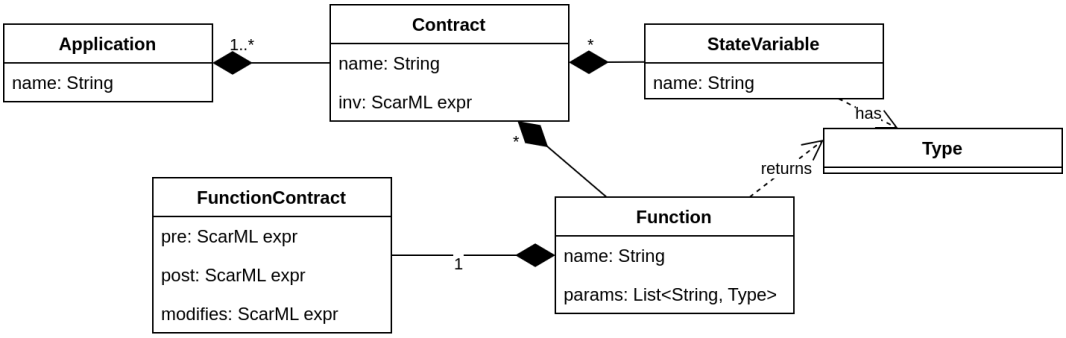

Figure 2 The core metamodel. The Type hierarchy is presented in Figure 5. Function contracts and invariants are expressions in the ScarML functional specification language described in Figure 7.

$$\mathit{frameExpr} ::= \texttt{modifies :}\ \mathit{id} \mid \mathit{id}.e \mid \mathit{id}.* \mid \mathit{id}[e] \mid \mathit{id}\ [e1..e2] \mid \mathit{id}[*]$$

Figure 3 The grammar for ScarML frame condition expressions.

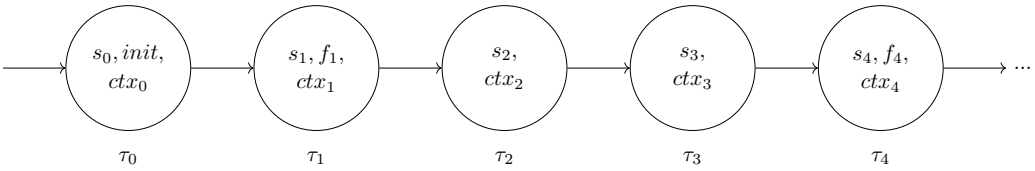

Figure 4 An example execution trace. Steps τ_3 and τ_4 are environment steps.

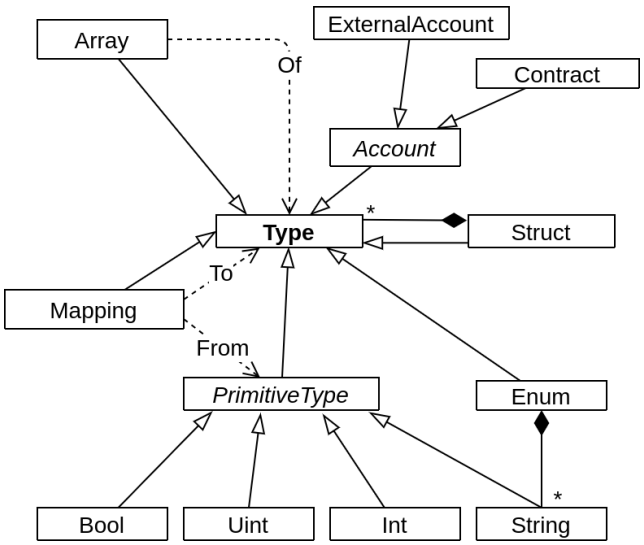

Figure 5 A graphical description of the type system.

$Expr$	$[\![Expr]\!]_{loc}$
`\nothing`	$\emptyset$
id	$\{id\}$
$id.e$	$([\![id]\!]_{loc}, [\![e]\!]_{loc})$
$c.*$	$[\![c]\!]_{loc} \times statevars(c)$
$s.*$	$[\![c]\!]_{loc} \times fields(c)$
id`[`e`]`	$([\![id]\!]_{loc}, [\![e]\!]_{loc})$
m`(*)`	$[\![a]\!]_{loc} \times dom(m)$
m`[*]`	$[\![a]\!]_{loc} \times ValsOf(t_1)$
a`[`$e1$`..`$e2$`]`	$[\![a]\!]_{loc} \times \{n \in \mathbb{N} \mid [\![e1]\!] \leq n \wedge n < [\![e2]\!]\}$
a`(*)`	$[\![a]\!]_{loc} \times dom(a)$
a`[*]`	$[\![a]\!]_{loc} \times \mathbb{N}$

Figure 6 Definition of the function frame evaluation function $[\![\cdot]\!]_{loc}$, where c is a contract, s is a struct, m is a mapping from type t_1 to t_2 and a is an array.

ϕ ::= $v \mid v$`[`t`]` $\mid v.id$
| `[1-9][0-9]+`
| `-`ϕ_3
| ϕ_3 `[+ | - | * | / | %]` ϕ_4
| `\result` | `\old(`t`)`
| `\caller` | `\amt` | `\systime` | `\blocknum`
| `\send(`ϕ_3`,`ϕ_4`,`ϕ_5`)`
| `\hash(`t`)`
| `true` | `false`
| ϕ_1 `[== | !=]` ϕ_2
| ϕ_1 `[=== | !==]` ϕ_2
| `!`ϕ | ϕ_1 `[&& | || | =>]` ϕ_2
| ϕ_1 `[< | <= | >= | >]` ϕ_2
| `[` $\forall$ | $\exists$ `]` `(`$qVar$`)` `:` ϕ
| `\sum(`ϕ_6`)`
| $StringLiteral$
| $funName$`(`$args$`)`
| `\values(`ϕ`)` | `\keys(`ϕ`)` | `\size(`ϕ`)`
| `\in`
| `\creates` $type$`:`id`(`$params$`)`

Figure 7 The grammar of ScarML. ϕ_1 and ϕ_2 are of type bool, ϕ_3 and ϕ_4 are number-typed, ϕ_5 is of type account, and ϕ_6 is a set of number-typed values.

$$
\begin{array}{rcl}
[\![\texttt{true}]\!] & := & \top \\
[\![\texttt{false}]\!] & := & \bot \\
[\![v]\!] & := & s_i(v) \\
[\![v[t]]\!] & := & s_i((v, [\![t]\!])) \\
[\![\texttt{-}\phi]\!] & := & -[\![\phi]\!] \\
[\![\phi_1 + \phi_2]\!] & := & [\![\phi_1]\!] + [\![\phi_2]\!] \\
[\![\phi_1 - \phi_2]\!] & := & [\![\phi_1]\!] - [\![\phi_2]\!] \\
[\![\phi_1 * \phi_2]\!] & := & [\![\phi_1]\!] * [\![\phi_2]\!] \\
[\![\texttt{\textbackslash result}]\!] & := & return_i \\
[\![\texttt{\textbackslash old}(t)]\!] & := & [\![t]\!]_{s_{i-1}} \\
[\![\texttt{\textbackslash caller}]\!] & := & caller_i \\
[\![\texttt{\textbackslash amt}]\!] & := & amt_i \\
[\![\texttt{\textbackslash systime}]\!] & := & systime_i \\
[\![\texttt{\textbackslash blocknum}]\!] & := & blocknum_i \\
[\![\texttt{\textbackslash send}(\phi_3,\phi_4,\phi_5)]\!] & := & send([\![\phi_3]\!], [\![\phi_4]\!], [\![\phi_5]\!]) \\
[\![\texttt{\textbackslash hash}(t)]\!] & := & hash([\![t]\!]) \\
[\![\phi_1 \texttt{ == } \phi_2]\!] & := & [\![\phi_1]\!] =_{val} [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ != } \phi_2]\!] & := & [\![\phi_1]\!] \neq_{val} [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ === } \phi_2]\!] & := & [\![\phi_1]\!] =_{ref} [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ !== } \phi_2]\!] & := & [\![\phi_1]\!] \neq_{ref} [\![\phi_2]\!]
\end{array}
\qquad
\begin{array}{rcl}
[\![\texttt{!}\phi]\!] & := & \neg[\![\phi]\!] \\
[\![\phi_1 \texttt{ \&\& } \phi_2]\!] & := & [\![\phi_1]\!] \wedge [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ || } \phi_2]\!] & := & [\![\phi_1]\!] \vee [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ => } \phi_2]\!] & := & [\![\phi_1]\!] \Rightarrow [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ < } \phi_2]\!] & := & [\![\phi_1]\!] < [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ <= } \phi_2]\!] & := & [\![\phi_1]\!] \leq [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ >= } \phi_2]\!] & := & [\![\phi_1]\!] \geq [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ > } \phi_2]\!] & := & [\![\phi_1]\!] > [\![\phi_2]\!] \\
[\![\forall\ qVar : \phi]\!] & := & \\
\multicolumn{3}{r}{\forall x \in RangeEval(qVar) : [\![\{qVar/x\}\phi]\!]} \\
[\![\exists\ qVar : \phi]\!] & := & \\
\multicolumn{3}{r}{\exists x \in RangeEval(qVar) : [\![\{qVar/x\}\phi]\!]} \\
[\![\texttt{\textbackslash keys}(m)]\!] & := & domain(m) \\
[\![\texttt{\textbackslash values}(m)]\!] & := & codomain(m) \\
[\![\texttt{\textbackslash sum}(a)]\!] & := & \sum_{i \in domain(a)} [\![a]\!](i) \\
[\![\texttt{\textbackslash size}(m)]\!] & := & |codomain(m)| \\
[\![v(t)]\!] & := & s_i((v, [\![t]\!])) \text{ if } [\![t]\!] \in domain(v) \\
[\![\phi_1/\phi_2]\!] & := & [\![\phi_1]\!]/[\![\phi_2]\!] \text{ if } [\![\phi_2]\!] \neq 0 \\
[\![\phi_1\%\phi_2]\!] & := & [\![\phi_1]\!] mod [\![\phi_2]\!] \text{ if } [\![\phi_2]\!] \neq 0 \\
a \ mod \ n & := & a - n * (a/n)
\end{array}
$$

Figure 8 The ScarML evaluation function $[\![\cdot]\!]$. The function is parameterized with the step τ_i, which contains the environment values $caller_i$, amt_i, $systime_i$, $blocknum_i$, and the prestate s_{i-1} and poststate s_i.

$$
\begin{array}{rcl}
app & ::= & \texttt{application : } userType* \ appInit \ contract+ \\
userType & ::= & structDef \mid enumDef \\
appInit & ::= & \texttt{appInit : } id \texttt{ : } contractType\texttt{(}params\texttt{)} \\
 & & [\texttt{, } id \texttt{ : } contractType\texttt{(}params\texttt{)}]* \\
contract & ::= & init?\ id \ state \ functions \\
init & ::= & initParams?\ initCond+ \\
initParams & ::= & \texttt{initParams : } id \texttt{ : } typeExpr\ [\texttt{, } id \texttt{ : } typeExpr]* \\
initCond & ::= & \texttt{init : } specExpr \\
state & ::= & \texttt{state : } stateVar+ \\
stateVar & ::= & \texttt{var } id \texttt{ : } typeExpr \\
functions & ::= & \texttt{functions : } function+ \\
function & ::= & \texttt{fun } id \texttt{ : } params?\ ret?\ pre*\ post*\ frame* \\
params & ::= & \texttt{params : } id \texttt{ : } typeExpr\ [\texttt{, } id \texttt{ : } typeExpr]* \\
ret & ::= & \texttt{returns : } typeExpr \\
pre & ::= & \texttt{pre : } specExpr \\
post & ::= & \texttt{post : } specExpr \\
typeExpr & ::= & primitiveType \mid arrType \mid mapType \mid id \\
primitiveType & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{uint} \mid \texttt{string} \\
arrType & ::= & typeExpr\texttt{[]} \\
mapType & ::= & \texttt{mapping(}typeExpr\texttt{=>}typeExpr\texttt{)} \\
frame & ::= & \texttt{modifies : } frameExpr
\end{array}
$$

Figure 9 A grammar for the basic metamodel. The *specExpr* symbol is defined in the functional specification language grammar (see Figure 7). The *frameExpr* symbol is defined in Figure 3.

```
application: bank

contract bank:

  invariant: total >= 0
  invariant: total = \sum(\values(m))

  state:
    var total: uint
    var balances: mapping(account=>uint)

  init: total == 0
  init: \size(balances) == 0

  functions:
    fun deposit:
      post: balances[\caller]
            == \old(balances[\caller]) + \amt
      post: \send(\caller, \this, \amt)
      modifies: balances[\caller]
    fun withdraw:
      params: uint amount
      pre: amount <= balances[\caller]
      post: balances[\caller]
            == \old(balances[\caller]) - \amt
      post: \send(\this, \caller, amount)
      modifies: balances[\caller]
```

Figure 10 A simple Scar application model.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >= 0.7;

library UTIL {
  enum Role {ANY}
  function hasRole(address a, Role r) internal pure returns (bool) { }
}

contract bank {
     mapping(address=>uint) balances;
   int total;
   constructor () { }

  /// @notice precondition balances[msg.sender] >= amount
  /// @notice postcondition  balances[msg.sender] == __verifier_uint_old(
      balances[msg.sender]) - amount
  /// @notice postcondition address(this).balance == __verifier_uint_old(
      address(this).balance) - amount
  /// @notice postcondition address(msg.sender).balance ==
      __verifier_uint_old(address(msg.sender).balance) + amount
  function withdraw(int amount)  public   { }

  /// @notice postcondition balances[msg.sender] == __verifier_uint_old(
      balances[msg.sender]) + msg.value
  /// @notice postcondition address(this).balance == __verifier_uint_old(
      address(this).balance) + msg.value
  /// @notice postcondition address(msg.sender).balance ==
      __verifier_uint_old(address(msg.sender).balance) - msg.value
  function deposit()  public   { }

}
```

Figure 11 The generated Solidity code with SOLC-VERIFY annotations.

Example	LoC Solidity	LoC Scar
Bank	11	20
Escrow	24	18
Auction	38	34
Casino	58	53
Palinodia	299	174

Figure 12 Lines of code comparison.