Solver-aided inference of abstraction invariant for the safe evolution of smart contracts

Filipe Arruda
Centro de Informática, Universidade Federal de Pernambuco, Brazil

Pedro Antonino
The Blockhouse Technology Limited, UK

Augusto Sampaio
Centro de Informática, Universidade Federal de Pernambuco, Brazil

A. W. Roscoe
The Blockhouse Technology Limited, UK
University College Oxford Blockchain Research Centre, UK

Abstract

The main contribution of this paper is a constructive and automatic strategy for the safe evolution of smart contracts that involves data refinement. In particular, an abstraction invariant that relates the state variables of the original and the evolved smart contracts is automatically inferred. Our solver-aided technique uses the Alloy model finder to infer candidate invariants from state variable valuations extracted from controlled executions of Solidity contracts. This is coupled with a verifier to check the validity of a candidate invariant and is carried out in a loop until a valid invariant is yielded or a certain bound is reached. We evaluate how our strategy fares when tackling real Solidity implementations of some Ethereum and other standards; we show that it has better coverage of data structures and accuracy than related approaches.

2012 ACM Subject Classification Software and its engineering → Software evolution; Theory of computation → Verification by model checking; Security and privacy → Logic and verification; Theory of computation → Hoare logic; Theory of computation → Abstraction

Keywords and phrases Smart contract, Data refinement, Abstraction invariant inference, Safe evolution

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Smart contracts are programs that manage a portfolio of digital assets that can be extremely valuable; they are deployed and executed on a blockchain [1]. Their code alone dictates how the assets are managed so a bug in the code can result in asset losses worth vasts amounts of money [9, 50]. Formal verification offers a systematic approach to analysing smart contracts against their intended specifications, helping to eliminate in this process these costly flaws.

Well-established refinement techniques can be used to ensure the safe evolution of smart contract implementations. For example, in traditional *algorithmic refinement* techniques, the preconditions of public functions can be weakened and their postconditions strengthened. More generally, implementation evolution might embody a change of data representation (types of the state variables). In this case, the reasoning, known as *data refinement*, requires an *abstraction invariant*, α, that relates the data representations of two contracts, say an original (or abstract) contract C and a refinement (or concrete) contract C'.

A typical limitation of several data refinement strategies is requiring users to provide the abstraction invariant α. This tends to be error-prone and has been a stumbling block for achieving fully automated support for program evolution. Existing approaches to the automatic inference of α are still very limited in scope. For example, the work in [14] automatically infers relations described as extensional sets of pairs, rather than as predicates used in the refinement proofs. A more recent approach [11] learn relations, described as predicates, from simulations of Solidity smart contracts. It takes as input two Solidity contracts (denoted *abstract* and *concrete*) and infers an abstraction invariant that relates their data representations. However, only simple invariants relating variables of primitive types and variable renaming are handled; collections like arrays and maps are considered only in the restricted cases of identity/renaming relations.

The main contributions of this paper are: (1) We devised a fully automatic strategy that infers an abstraction invariant α in the form of a predicate. Our strategy takes an abstract and a concrete Solidity contract implementations as input and performs executions to find assignments (valuations) for the state variables of both contracts. In our approach, these assignments are then provided as input to the Alloy Analyzer (together with an Alloy model that encodes a grammar for expressing abstraction invariants) to generate a candidate $\hat{\alpha}$. (2) The validity of $\hat{\alpha}$ is established based on given specifications (S and S') for the abstract and concrete contracts. This checks whether S is data refined by S'. These specifications are used only for the validation of $\hat{\alpha}$, and not in the inference process. The verification is conducted using the tool solc-verify [32] using an encoding template to allow the refinement verification. To the best of our knowledge, our strategy is the first to lift abstract invariants to (and, hence, prove data refinement between) smart contract specifications. (3) The proposed strategy was able to infer abstraction invariants that have allowed the automatic verification of commit histories of implementation (and specification) evolution of Ethereum standards, such as ERC20 and ERC721, and other standards. Some abstraction invariants relate only state variables of primitive types, but others infer invariants involving variables whose types are maps, arrays, and structs. In the case of array and map variables, the inference is able to generate universally quantified predicates.

In the next section, we introduce the Solidity language, using a running example, the design-by-contract paradigm and the *solc-verify* tool. Section 3 presents our abstraction-invariant inference strategy whose evaluation is discussed in Section 4. Section 5 presents related work and is followed by our conclusions.

2 Background

Solidity. To introduce Solidity and to illustrate the proposed strategy, we use a running example based on a real refactoring of the DigixDAO ERC20 Token. A fragment of the abstract contract implementation in Solidity (version `0.5.0`) is presented in Listing 1. A contract in Solidity allows the declaration of types, attributes and functions. In our example, there is one attribute that is a mapping denoted by `users` from addresses (represented by a 160-bit number) to a user-defined struct `User`, in which a single field balance, whose type is a 256-bit unsigned integer, is defined. An address is similar in nature to a bank account; digital assets (like tokens) associated with an address are held in (i.e. managed via) it. Our example has a single public function `transfer` that transfers a token amount (parameter `_value`) from the caller (whose address is stored in the implicit argument `msg.sender`) to a destination address (parameter `_to`). This function yields a boolean value that states whether the execution was successful.

The applicability of a function can be captured using the `require(condition)` statement. If the `condition` holds, the execution proceeds normally; otherwise, the function execution aborts and the state before the start of the function execution is preserved. The first conjunct in the `require` clause of the implementation of the `transfer` function requires that the sender must have enough balance for the transfer; the second conjunct requires that crediting the `_value` amount in the destination address must not generate an overflow, and that this value is greater than zero. The next two assignments capture the effect of the transfer: the balance of the sender is decreased by `_value` and that of the destination is increased by the same amount. The `emit` keyword is used to communicate an event in Solidity; in our example, it is used to log the transfer transaction.

Design by contract and solc-verify. In the design-by-contract paradigm, the specification of a component includes invariants, and, for each of its public functions, pre- and postconditions. The component constructor (the initialisation) must ensure that the invariant holds, and, for each function, the preconditions are assumed and the function implementation must guarantee that the postconditions and invariants hold upon termination (partial correctness). Our strategy considers partial correctness, as supported by the major verification tools for Solidity smart contracts, including solc-verify, the tool used here.

```
contract ERC20Token {
    struct User { uint256 balance; }
    mapping (address => User) public users;

    function transfer(address _to, uint256 _value) public returns (bool
        success) {
        require(users[msg.sender].balance >= _value && users[_to].balance
             + _value > users[_to].balance);
        users[msg.sender].balance -= _value;
        users[_to].balance += _value;
        ...
    } ...
}
```

Listing 1 Abstract ERC20Token implementation

Listing 2 shows a fragment of the specification for `transfer` in the syntax of solc-verify. It includes two postconditions that are implicitly conjoined. The first postcondition determines that the sender's balance should be decreased by the transferred amount, provided that the sender is not the recipient; otherwise, the balance must be preserved. For an attribute x, `__verifier_old_uint(x)` holds the value of `x` at the start of the function execution. The notation for conjunction (`&&`), disjunction (`||`) and negation (`!`) are standard. The second postcondition states that the recipient's balance must increase by the transferred amount, unless the sender is also the recipient, in which case the balance must be preserved.

```
(users[msg.sender].balance == __verifier_old_uint(users[msg.sender].
    balance) - _value && msg.sender != _to) || (users[msg.sender].
    balance ==  __verifier_old_uint(users[msg.sender].balance) && msg
    .sender == _to)
(users[_to].balance == __verifier_old_uint(users[_to].balance) +
    _value  && msg.sender  != _to) || (users[_to].balance ==
    __verifier_old_uint(users[_to].balance) && msg.sender == _to)
```

Listing 2 Postconditions for the (abstract) `transfer` function using solc-verify syntax

3 Strategy

We propose a three-stage strategy to infer abstraction functions (see Figure 1); the input are two contract implementations: an abstract C and a concrete C'; and two specifications: abstract S and concrete S'. Overall, it tries to infer an abstraction invariant (currently

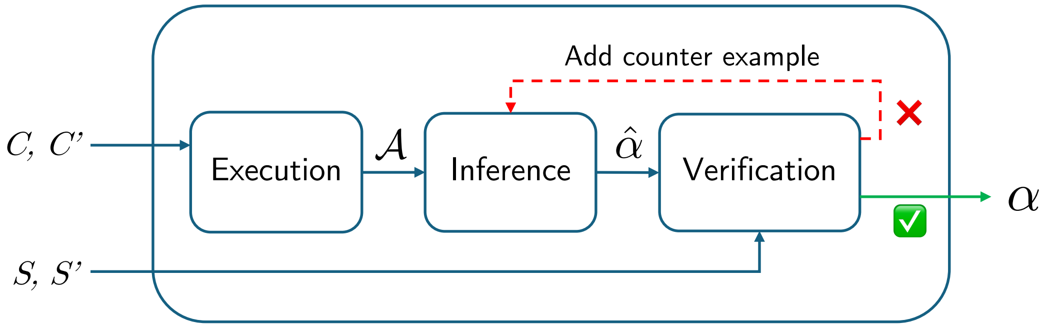

Figure 1 Strategy Overview

restricted to a function) from the states of S' to the states of S by using information extracted from the execution of the corresponding implementations C and C'. We assume that C conforms to S and C' to S' and rely on the fact both S and C share the same data representation, since they have the same member variables; the same holds for S' and C'. The existence of an abstraction function α is also a proof that S' (data) refines S.

The smart contract `ERC20Token` fragment and the corresponding specification presented in the previous section are examples of abstract implementation (C) and specification (S), respectively. An example of a concrete contract implementation fragment (C') is given in Listing 3; it shows an evolution of C that uses a slightly different mapping (`balances`) that links addresses directly to balances represented by integer values. This data representation change is reflected in several places in the code. In the abstract version, a balance associated with a given address, say `addr`, is referenced as `users[addr].balance`. In the concrete version, this is referenced more directly as `balances [addr]`. The change in data representation is also reflected in the specification S' (see Listing 4).

```
contract ERC20Token {
    mapping (address => uint256) public balances;

    function transfer(address _to, uint256 _value) public returns (
        bool success) {
        require(balances[msg.sender] >= _value && balances[_to] +
            _value > balances[_to]);
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        ...
    } ...
}
```

Listing 3 Sample evolution of an ERC20Token contract

```
(balances[msg.sender] == __verifier_old_uint(balances[msg.sender]) -
    _value && msg.sender != _to ) || (balances[msg.sender] ==
    __verifier_old_uint(balances[msg.sender]) && msg.sender == _to)
(balances[_to] == __verifier_old_uint(balances[_to]) + _value  && msg
    .sender  != _to ) || (balances[_to] == __verifier_old_uint(
    balances[_to] ) && msg.sender == _to)
```

Listing 4 Postconditions for the (concrete) `transfer` function using solc-verify syntax

We use these implementation and specification pairs as a running example to illustrate, step by step, our abstraction invariant inference strategy.

3.1 Execution

This step generates a set $\mathcal{A}$ with pairs of states (s', s) where s' is in the state space of C' and s is in the state space of C. These samples of state variable valuations for concrete and abstract states guide our inference step: our inference engine searches for an abstraction function that must satisfy $s = \alpha(s')$, when α exists.

To obtain such pairs, we subject the abstract and concrete contracts to the same executions using a local blockchain network. An execution $\mathcal{E}$ is a finite non-reverting sequence of function calls $\langle f_0, f_1, \ldots f_k\rangle$ such that f_0 is the constructor call. For a chosen execution $\mathcal{E}$, we perform the prescribed sequence of function calls on (an instance of) C and recover its final state s; we use $\mathcal{E}(C)$ to denote the final state after the instance of C performs $\mathcal{E}$, i.e. $s = \mathcal{E}(C)$. Similarly, we do the same to C' to recover the final state s'. We end up with the pair (s', s).

We use a heuristic to select a sequence of function calls: we look for executions in which all member variables of both C and C' are assigned to; otherwise, we would capture only partial behaviours on the state samples. For each contract function, we carry out a static analysis of its body to identify which member variables are being assigned. Then, we can look for executions, possibly involving calling only a subset of the functions of these contracts, which cover (assign to) all the member variables of a contract. This heuristic is implemented with a set cover algorithm [36] where each function f_i is associated with the set S_i of variables it assigns to and the cost of this set is given by the function's number of parameters c_i. The algorithm finds a collection of sets such that their union includes all the member variables of the contract while minimising the sum of the costs for the sets in this collection. A sequence of calls involving the functions in this collection gives an execution with minimal "parameter cost". Once we have determined the sequence of function calls, a simple and useful algorithm is employed to select inputs (arguments) for the candidate functions in each execution. Our algorithm selects a random range of values for each primitive type, such as address and integer, and employ a round-robin selection. While this random selection is sufficient for our experiments, we can also integrate well-established input generation tools into our approach, such as Coverage-Guided Fuzzing [3]. We consider exploring this and other tools as future work. The number of executions that this step generates is a parameter of our strategy; for our experiments, two state pairs were enough to find a valid abstraction function.

In the case of state variables of structured types like map, it is particularly challenging to obtain a valuation, particularly in Solidity, as there is not built-in operation to obtain the indices used in a map. So, to find which indices of the contract have been modified by an execution, we instrument the implementation to emit an event that keeps track of the index every time a maplet is updated; we apply the same procedure to state variables of type array.

For our running example, this step chooses the following two executions. The semicolon in the `transfer` call denotes that `msg.sender` is an implicit parameter.

- $\mathcal{E}_0 = \langle$`constructor(), transfer(0x02, 1; msg.sender = 0x01)`$\rangle$,
- $\mathcal{E}_1 = \langle$`constructor(), transfer(0x01, 2; msg.sender = 0x02)`$\rangle$.

These executions give rise to the set $\mathcal{A}_{re} = \{(\sigma'_1, \sigma_1), (\sigma'_2, \sigma_2)\}$ where σ_i and σ'_i are defined below; note that $\sigma_i = \mathcal{E}_i(C)$ and $\sigma'_i = \mathcal{E}_i(C')$. The symbol $\mapsto$ denotes a maplet, an indice-to-value mapping for a map value, and we use `.name` to denote the element `name` of a struct field.

$$\begin{aligned}
\sigma_1 &= (\texttt{users} = \{\texttt{0x01} \mapsto \{\texttt{.balance} = 0\}, \texttt{0x02} \mapsto \{\texttt{.balance} = 1\}\}) \\
\sigma_1' &= (\texttt{balances} = \{\texttt{0x01} \mapsto 0, \texttt{0x02} \mapsto 1\}), \\
\\
\sigma_2 &= (\texttt{users} = \{\texttt{0x01} \mapsto \{\texttt{.balance} = 2\}, \texttt{0x02} \mapsto \{\texttt{.balance} = 0\}\}) \\
\sigma_2' &= (\texttt{balances} = \{\texttt{0x01} \mapsto 2, \texttt{0x02} \mapsto 0\})
\end{aligned} \tag{1}$$

Our implementation performs some optimisations. If two functions f and f' cover the same set of variables but f has more parameters than f', we choose f' over f when selecting executions; fewer parameters means a simpler instantiation. Moreover, when capturing the final state of an execution, our strategy disregard member variables shared between contracts C and C' — i.e. declared with the same name and type—as they are assumed not to have been affected by the refinement. For these cases, the abstraction function simply identifies the value of the abstract and concrete versions for each of these variables. From the set of experiments we have conducted, this assumption is indeed valid. In the general case, however, the concrete and abstract contract might declare the same variable but use them for distinct purposes. In such cases, by identifying these variables, the search for an abstraction function will fail, and the entire set of state variables (properly qualified with the contact name for disambiguation) is used instead.

If an execution fails due to the unfulfilled precondition of a function, we translate its pre- and postcondition into predicates in Alloy to find an initial state that enables the given sequence of function calls. Then, this initial state found by our strategy is inserted into the contract's constructor. For instance, in our running example, to first transfer a given amount `_value`, the sender must have at least this amount in its balance; this is added to the constructor.

3.2 Inference

For the second stage, we use Alloy [38] to infer an abstraction function. Alloy is a declarative modeling language inspired by other formalisms, such as Z [51], VDM [39], and OCL [47]. It provides ways to describe succinct models, in terms of sets and relations; the features of Alloy that we use are explained on demand.

We first characterise the inference process more generally, and then we explain how it is captured using Alloy. The aim is to infer an abstraction function that maps concrete states to abstract ones (backwards simulation) such that it satisfies the associations $\mathcal{A}$ obtained in the execution step. An abstraction function is defined as a set of equalities. For each abstract state variable y, the inference engine searches for an expression e whose free variables are a subset of the concrete state variables to generate an equality of the form $y = e$. If y is a map or array variable, we have instead a quantified equality $\forall\, i \bullet y[i] = e$ where, in this case, e might also include occurrences of the index variable i, in addition to the concrete variables. To simplify the inference process, we restrict e to be an arithmetic expression so that a pre-processing is carried out to map other Solidity types (like addresses and booleans) into Alloy integers. More precisely, the expression e is defined by the following grammar, where c is an integer constant, x is a primitive concrete variable, $m[i]$ is a map or array access for a concrete or array variable m and quantified index variable i.

$$e ::= c \mid x \mid m[i] \mid e + e \mid e - e \mid e * e \mid e/e \tag{2}$$

We use the associations $\mathcal{A}$, obtained in the Execution step, to generate expressions in a way that the candidate abstraction function $\hat{\alpha}$ satisfies the pairs of states in $\mathcal{A}$. For instance, for our running example and the specific associations $\mathcal{A}_{re}$ created in Equation 1, the first abstraction function is a valid candidate $\hat{\alpha}$ whereas the second is not because it does not satisfy the pairs in $\mathcal{A}_{re}$.

1. $\forall\, i :$ address • users[i].balance $=$ balances[i]
2. $\forall\, i :$ address • users[i].balance $= 5$

When the abstraction function is translated into the the solc-verify notation (for the final, Verification, step of the strategy) the original values of the Solidity state variables are generated back from the Alloy integer encoding. Moreover, in our Alloy encoding, each component of a struct variable is treated as a separate variable of a primitive type; we do not support nested structs for the sake of simplicity but we could easily accommodate them. The same holds for variables of a map type targeting a struct; a separate map variable is generated for each struct field and we do not allow nested maps. In the following, we show some snippets of our encoding in Alloy; the complete encoding is given in the Appendix of the extended version [8].

```
abstract sig Assignment {
    variable: one Variable,
    value: ExecutionId -> (Int + Mapping)
}
one sig Assignment_MappingType_balances_Target extends Assignment {} {
  variable = Var_balances_Target;
  value =
    Exe0 -> Mapping_MappingType_balances_Target_Exe0
    + Exe1 -> Mapping_MappingType_balances_Target_Exe1
}
one sig Mapping_MappingType_balances_Target_Exe0 extends Mapping {} {
    element = 1 -> 0 + 2 -> 1 }
one sig Mapping_MappingType_balances_Target_Exe1 extends Mapping {} {
    element = 1 -> 2 + 2 -> 0 }
```

Listing 5 Alloy assignment signature and example assignment

```
one sig AbsFun {
 equations: set Equation
}
abstract sig Equation {
 abstractVar: one Variable,
 concreteExpr: one Expression
} {
    abstractVar.version = Abstract
    concreteExpr.variables.version = Concrete
}
sig EqualsMapping extends Equation {} {}
fact {
  ∀ eqmap: EqualsMapping  •
    eqmap.concreteExpr in (IndexedExpression +
        MappingAssignmentExpression) and
    eqmap.abstractVar.type in MappingType and
    ∀ executionId: domain[univ.concreteExpr.evaluation] •
        ∀ index: (variable.(univ.abstractVar).value[executionId].
            element).univ  •
            eq[int eqmap.concreteExpr.evaluation[executionId].element[
                index],
                int variable.(eqmap.abstractVar).value[executionId].
                    element[index]]
}
```

Listing 6 Abstraction Function - Alloy Encoding

Listing 5 illustrates how pairs of states in $\mathcal{A}_{re}$ in Equation 1 are encoded. Particularly, it encodes concrete states σ_1' and σ_2' for the variable `balances`. A (state variable) valuation is automatically translated into an Alloy signature that extends the `Assignment` signature. A signature defines a set and has fields that define its properties. `Assignment` has a field `variable`; the quantifier *one* indicates that an `Assignment` has a single variable. The other field, `value`, associates executions (`ExecutionId`) with valuations represented by the set union (`+`) of integer and map values. The values associated with the two executions (`Exe0` and `Exe1`), for variable `balances`, are mappings; they both extend a signature `Mapping` (ommited) in which we define the elements indexed by a key. The first value includes two maplets `1 -> 0` and `2 -> 1`, representing a balance of `0` tokens at address `1` and `1` token at address `2`; analogously, the second one includes two maplets: `1 -> 2 + 2 -> 0`.

The signature in `AbsFun` Listing 6 models an abstraction function, which is defined as a set of equations. Each `Equation` is an equality between a variable and an expression and must obey a constraint stated as an implicit fact: the variable must be a state variable of the abstract contract, and the expression has as free variables (a subset of) those of the concrete contract. The semantics of equality for an equation is defined separately for primitive and structured types like maps. For primitive types (given by the signature `Equals`), this is captured by a constraint which states that the evaluations of the variable and the associated expression must yield the same value. For maps, this is defined by the signature `EqualsMapping` in Listing 6 that allows relating indexed variables and expressions using quantification. This is stated as a `fact` that lifts equality from simple equations to quantified ones, by requiring that the equality holds for each indexed element.

Given our encoding, Alloy finds a model—representing a candidate abstraction function $\hat{\alpha}$—considering a fixed bound on the size of the expressions. Since Alloy is a bounded model finder, the bound of the analysis can be increased until a fitting one is identified or a given hard limit is reached. If the bound limit is reached, we assume that there is no candidate abstraction invariant.

3.3 Verification

The verification phase checks whether the candidate abstraction function $\hat{\alpha}$, generated by the Alloy Analyzer, is a valid abstraction invariant. We use an encoding in solc-verify of the properties expected of an abstraction invariant α from S' to S to verify that.

We formalise a specification as a triple (D, I, A) containing a data representation D (a set of variable declarations), an interface I (a set of function signatures with a designated *constructor* function), and annotations A including a pre- and postcondition for every function f in I (denoted by pre_f and $post_f$) and an invariant (denoted by inv).

A specification induces a transition system $(Q^{\hat{s}}, \Sigma, \Delta)$ in the conventional way: Q is the set of states induced by the valuations of variables in D (the state space of this system is Q extended with the designated special starting state $\hat{s}$, i.e. $Q^{\hat{s}}$), Σ contains all the function calls for interface I, and the transition relation $\Delta \subseteq Q^{\hat{s}} \times \Sigma \times Q$ consists of: initial transitions $(\hat{s}, e, s')$ where e is a constructor call and s' satisfies the constructor postcondition, and transitions (s, e, s') for $e \in Q$ and calls e of any other function f such that if s satisfies the precondition of this function then s' must satisfy its postcondition. A path in this transition system is a sequence of states and calls $\langle s_0, e_0, s_1, \ldots, s_{n-1}, e_{n-1}, s_n \rangle$ such that $(s_i, e_i, s_{i+1}) \in \Delta$ for all transitions in this path. We use $paths(S)$ to denote the paths of the transition system induced by S.

When considering two specifications S and S', we refer to the elements of S' (or of its induced transition system) using their dashed versions. For instance, Q is the induced set of states for S whereas Q' is that of S'.

▶ **Definition 1.** *A function α from Q' to Q is an* abstraction invariant *from S' to S iff:*

- $\forall\, s \in Q' \bullet inv'(s) \Rightarrow inv(\alpha(s))$
- *For each function $f \in I$ with precondition $pre(\cdot)$, postcondition $post(\cdot)$ (and dashed counterparts) with input and output argument spaces Ins and $Outs$:*
  - $\forall\, s \in Q', is \in Ins \bullet pre(\alpha(s), is) \Rightarrow pre'(s, is)$
  - $\forall\, s, s' \in Q', is \in Ins, os \in Outs \bullet$ $post'(s, s', is, os) \Rightarrow post(\alpha(s), \alpha(s'), is, os)$

This abstraction function demonstrates that a concrete specification (data) refines an abstract one. We have that $S \sqsubseteq_{\alpha} S'$ (i.e. S' refines S via abstraction function α) *iff* $paths(S) \upharpoonright pre \supseteq \alpha(paths(S')) \upharpoonright pre$. Intuitively, this definition states that there must be a function that ensures that the concrete specification behave as prescribed by the abstract specification for well-behaved abstract paths, respecting the abstract preconditions. We use $paths(X) \upharpoonright p$ to denote the paths of specification X such that each transition (s, e, s') along the path respect the corresponding precondition, i.e. $p_f(s)$ with f the function call in e; note that in our refinement definition both operands of $\supseteq$ rely on the preconditions of S. Additionally, we use $\alpha(X)$ to denote the lifting of paths in X using the function α, namely, the paths resulting from lifting the states along them using α.

▶ **Theorem 2.** *If α is an abstraction invariant from S' to S then $S \sqsubseteq_{\alpha} S'$.*

Verification tools for the design-by-contract paradigm usually do not provide built-in support to reason about data refinement. This includes solc-verify. So, we needed to engineer an encoding to represent (and check) our abstraction invariant requirements in Definition 1 in terms of pre-/postcondition verification. For this purpose, we define the `AbsFunc` contract, as shown in the template in Listing 7.

```
contract AbsFunc {

    /// @notice precondition $CandidateAbsFuncPredicate$
    /// @notice precondition $ConcreteInvariant$
    /// @notice postcondition $AbstractInvariant$
  function inv() public {}

    /// @notice precondition $CandidateAbsFuncPredicate$
    /// @notice precondition $AbstractPreconditionForFunc$
    /// @notice postcondition $ConcretePreconditionForFunc$
  function Func_pre(type1 arg1, type2 arg2, ...) public returns (
      ret_type1 ret_arg1, ret_type2 ret_arg2, ...) {}

    /// @notice precondition $CandidateAbsFuncPredicate$
    /// @notice precondition $ConcretePostconditionForFunc$
    /// @notice postcondition $AbstractPostconditionForFunc$
  function Func_post(type1 arg1, type2 arg2, ...) public returns (
      ret_type1 ret_arg1, ret_type2 ret_arg2, ...) {}
}
```

Listing 7 Refinement verification

Given S, S' and $\hat{\alpha}$, for each function `Func` in I, it creates the annotated functions `Func_pre` and `Func_post` to capture the pre-/postcondition requirements (respectively) enforced on $\hat{\alpha}$ by `Func`, namely, that it relates states respecting precondition weakening and postcondition strengthening from S to S'. Similarly, the invariant requirement is captured by function `inv`. This encoding relies on the fact solc-verify (and similar tools) assume the precondition in order to prove the postcondition of a function. This is exactly the proof obligation that we need to prove the implications between pre- and postconditions of S and S' as described in Definition 1. Moreover, note that the candidate abstraction function $\hat{\alpha}$ is encoded as a precondition (i.e. it is assumed to hold) in all these annotations.

For our running example, the candidate function generated by Alloy, when translated back to the input notation of solc-verify, is given in Listing 8. This function is verified to be a valid abstraction function with respect to our specifications using our `AbsFun` contract encoding.

```
forall (adress addr) users[addr].balance = balances[addr]
```

Listing 8 Abstraction function

If the abstraction function verification does not hold, we treat the (inferred) candidate function as a counterexample. The strategy goes back to the inference step and our Alloy encoding is updated to block the finding of this function again. This process is repeated until a valid abstraction function is found or a parametrized limit is reached. If such a limit is reached, we assume there is no valid abstraction function between the concrete and abstraction contracts.

4 Evaluation

To evaluate the proposed strategy, we analyse implementations of smart contracts based on Ethereum and other standards. Since there is no consensus regarding the data model for each implementation, the data representation may evolve to correct bugs or to optimise transaction costs. Therefore, some implementation evolutions require abstraction invariants that relate the state variables of the original and those of the new contract to allow a data refinement proof.

In Table 1, we present some of these evolutions. All these contracts, together with the associated specifications that we have created from the documentation and reference implementations, were submitted to the automated analysis performed by our framework. Each row in the table represents either: *a*) third-party implementations [11], in which we inform the community reference contract as **Abstract** and the other repository, which implemented a version of the reference contract, as **Concrete**; or *b*) contract evolution, identified by its commit slug (as a subscript in the **Abstract** and **Concrete** columns) on the git repository.

The **Category** organises the data refinement into representative groups: *a*) Renaming, for contracts whose state variables differ only by their names; *b*) Identity, for contract evolutions that preserve the data representation but allow algorithmic refinement and interface extension (new functions); *c*) Destructuring, for mappings to structs that are decomposed into a mapping for each field of the struct; and *d*) Structuring, the reverse of Destructuring.

Our tool is parametrised by: the `alloy_scope` that defines the maximum range for other entities involved in the inference, such as assignments or relations; the `integer_range` that determines the maximum range of integers considered during execution (any number exceeding this range is normalised to fit within it); and the `rounds_count` that specifies how many times both abstract and concrete contracts are executed. To evaluate performance, we measured the execution times under varying parameter combinations (each ranging from 1 to 6). We

Standard	Abstract	Concrete	Category
ERC20	OpenZeppelin	Molochventures	Renaming
Escrow	OpenZeppelin	Stacktical	Identity
ERC721	OpenZeppelin	Ayidouble	Identity
ERC165	OpenZeppelin	Jbaylina	Renaming
Whitelist	OpenZeppelin	Hanzo	Renaming
ERC20	DigixDao$_{(91fa712)}$	DigixDao$_{(6c717c)}$	Deestructuring
ERC20	DigixDao$_{(6c717c)}$	DigixDao$_{(91fa712)}$	Structuring
ERC721	OpenZeppelin$_{(3a5da75)}$	OpenZeppelin$_{(07603d)}$	Deestructuring
ERC721	OpenZeppelin$_{(07603d)}$	OpenZeppelin$_{(3a5da75)}$	Structuring
ERC20	Uniswap$_{(064aefb)}$	Uniswap$_{(e382d70)}$	Identity
ERC20	SetProtocol$_{(b1cc53)}$	SetProtocol$_{(a4bfef0)}$	Structuring
ERC20	SetProtocol$_{(a4bfef0)}$	SetProtocol$_{(b1cc53)}$	Deestructuring
CrowdSale	OpenZeppelin	ConsenSysMesh	Renaming

Table 1 Abstraction invariant inference for real-world contracts

applied this analysis to all contracts in the table and used Random Forest to identify which parameters significantly influenced the success of the inference. The results highlighted the alloy scope and the integer range as key parameters. Figure 2 is a heat map showing the success rate in the combination of these parameters. We observed that within smaller scopes (between 1 and 2), the success rate is low, but approaches 100% when `alloy_scope >= 3`. However, as the scope increased beyond a certain point, we encountered a state explosion problem, which exhausted system resources, thus reducing the success rate.

Figure 3 is a 3D chart in which the Z-axis represents the execution time. This figure reveals an exponential increase in the execution time as the other parameters grow. Another notable observation is that, for smaller scopes, more iterations are required to identify the correct relationships.

After processing the data, we found the best parameter configuration with respect to execution time and success rate. The combination that demonstrated the best performance was `integer_range=4`, `alloy_scope=2`, `rounds_count=1`. This optimal configuration was then chosen as the default initial argument set to evaluate the tool's efficiency, especially when comparing with other tools' benchmarks. For these parameters, we observed an average execution time of 5.8s. This result highlights the notable efficiency of our tool. However, we cannot make a precise comparison with the strategy reported in [11], since the times were measured in different experiments with distinct hardware configurations. We used a i5-11400F CPU and 32GB RAM. The SAT solver used by the Alloy Analyzer was SAT4J. The evaluation in [11] does not present a hardware configuration.

As previously mentioned, we have implemented some optimisations that have contributed to the efficiency of our tool. Particularly, we single out the assumption that state variables, whose names and types are preserved in an evolution, are related by the identity abstraction function. When the data representation is not affected by an evolution (all the cases with the `Identity` category), there is no need to run our Inference step. This preserves soundness since our framework always verifies a candidate abstraction invariant. If the data refinement verification fails, our strategy iterates to search for a valid abstraction invariant.

We are not aware of any other work able to automatically infer abstraction invariants with the data coverage we have achieved for smart contracts. Particularly, we handle the destructuring/structuring of mappings with structs as target types. This was not considered,

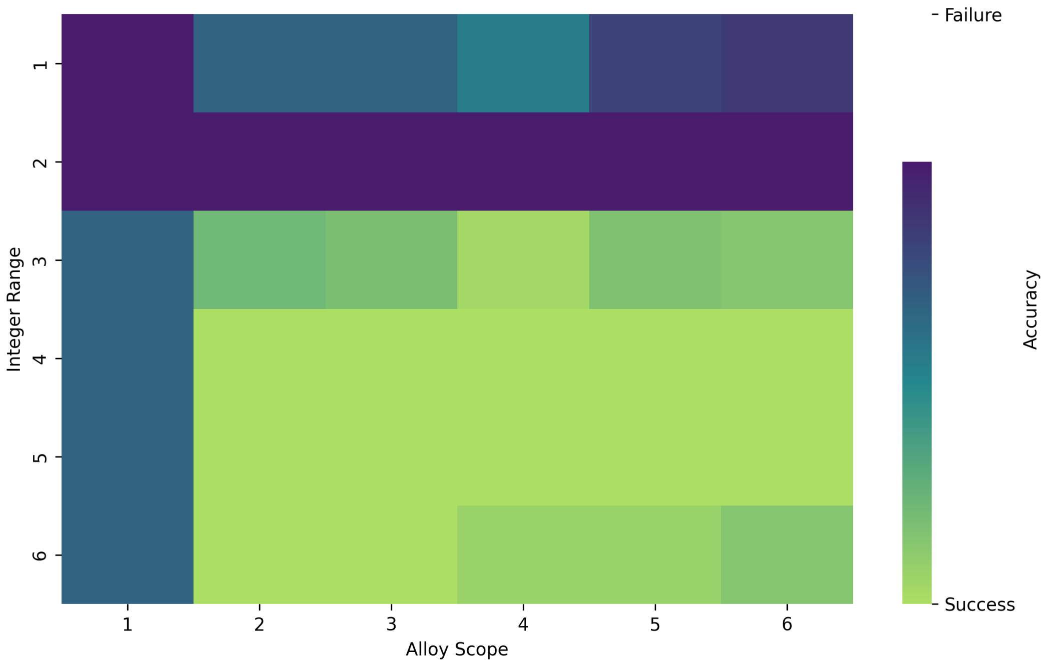

Figure 2 Success rate as a function of both `integer_range` and `alloy_scope`

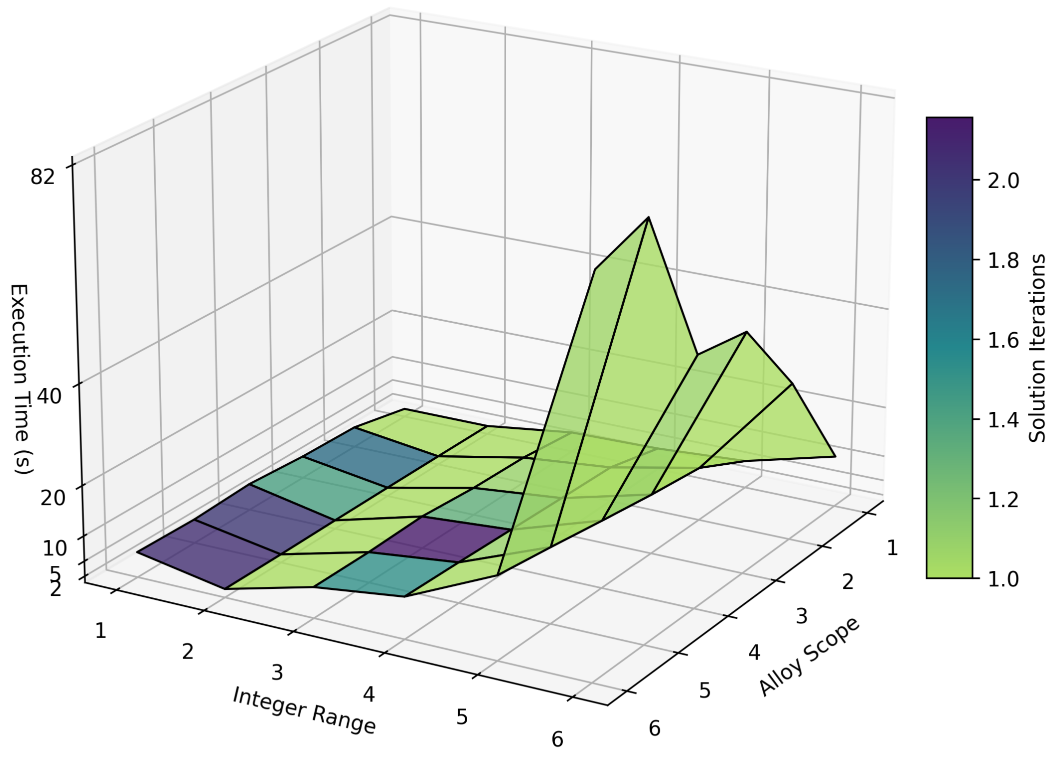

Figure 3 Execution time vs. `alloy_scope` and `integer_range`, colored by iteration count

for instance, in [11], which includes in their evaluation only the renaming and identity categories. Also, for the Whitelist standard (using the OpenZeppelin implementation as the abstract version and the Hanzo implementation[1] as the concrete version) the authors of [11] inform their strategy could not verify the validity of the abstraction function, whereas our tool inferred an abstraction invariant that was successfully verified to be valid.

Threats to validity.

We discuss here the threats to the validity of the evaluation.

Conclusion validity. Our results seem to suggest that our strategy is more general than the strategy in [11] as it can tackle more types of contracts. The authors of that paper do not provide the execution environment in which their experiment took place so we cannot compare our execution times with theirs but they did provide the examples analysed themselves which we used in our evaluation. Although the strategy in [11] explores relations in contracts with mappings, arrays, and structs, there is no evidence that they support quantification over these data types nor relations involving introduction or removal of structs. It is also important to note that we were able to find and verify relations for all contract evolutions presented in this paper including the ones which that strategy was unable to infer as per [11].

Internal validity. While we chose only standardized contracts to extract well-defined requirements, we wrote ourselves the formal specifications to verify the results. Besides, the solidity contracts needed to be modified both manually (to reorganize the imports) and automatically (to change variable visibility and add auxiliary events). While this is not expected to change their behaviour, it is nevertheless a modification of the original contract.

External validity. We considered only contracts that implement Ethereum and some other standards that might not represent the complexity of other domains.

5 Related work

Smart contract analysis and verification has been a very active topic of research in recent years [7, 9, 35, 41, 45, 56]; see [37, 53, 29, 4] for some surveys of this area. There have been approaches trying to find vulnerabilities on smart contracts in EVM-bytecode form using symbolic execution [41, 42], and static analysis [28, 52, 54, 45]. Solidity smart contracts were also verified using techniques like modular program verification [32, 31], bounded model checking [55, 7], and deductive verification [2]. Furthermore, there are frameworks, like our previous work [6], that create approaches for the safe deployment and upgrade or bug fixing/patching of smart contracts [18, 48, 10, 5].

A method used to automatically show refinement between two systems consists of finding a *simulation relation* between them. There have been approaches that automatically find simulations between abstract models expressed, for instance, using automata and labelled graphs [19, 34, 15, 46]. The approaches in [22, 23] automatically discovers simulations between programs using SMT-solving; the latter is aided by an inductive invariant of the abstract program. It works by iteratively creating abstractions of the target program and trying to find a simulation between the source and target (or an abstract version of it).

In the realm of smart contracts, our work has been inspired by the framework in [11]. It proposes a way to automatically find simulations between smart contracts by performing executions; the authors use learning techniques to create such a simulation from the samplings

[1] https://github.com/hanzoai/solidity/blob/master/contracts/Whitelist.sol

of the final values of the state variables obtained by these executions. Both approaches use a type of abstraction invariant that maps refined states into original ones — this type of simulation is typically called backwards [17] (or upwards [33] or even a co-simulation [25]). It is well-known that using either backwards or forwards simulation alone is not a complete method to show refinement between systems [17, 33]; so, both frameworks are sound but incomplete. Still concerning completeness, our approach is restricted to functional invariant whereas that in [11] handles relations. Even though being limited to functional invariants potentially narrows the applicability of our framework, that was not a limitation for the examples we analysed implementing real Ethereum-standard-based smart contracts. The same can be said about our backwards simulation method. Furthermore, as a distinctive feature of our approach, this restriction to functional simulations has allowed us to automatically infer more elaborate simulations than the approach in [11] and any other approach that we are aware of. Our framework is able to infer invariants that relate variables of array and map types, which involve quantification.

Finally, the approach in [14] automatically discovers different types of relations between Z data types using the Alloy Analyser. It introduces Alloy encodings to find a number of different types of relations (including, forwards and backwards) considering different types of Z semantics. While that work tries to synthesise an extensional relation, ours aims at finding a predicate that characterises an abstraction function. For large system with complex datatypes, finding a non-trivial extensional description of such a relation will be typically impractical, and even so it cannot be directly used in data refinement proof techniques that are parametrised by a relation described in the form of a predicate. We are unaware of any other approach that was able to infer invariant predicates that could be readily used to verify real implementations, including smart contracts.

The problem of automatically generating invariants (and pre-conditions) [21, 24, 16, 40, 20, 49] is closely related to that of discovering simulation/abstraction relations; the latter can be seen as a special case of the former. Some techniques rely on using executions of programs to synthesise *candidate* (i.e. potential) invariants —– these are typically instantiated from a catalogue of invariant templates; the technique in [21] uses a stochastic model to establish the likelihood of these candidates truly being invariants, whereas the technique in [43] uses symbolic execution to validate (but not prove) numerical candidates (including non-linear ones). A number of techniques go further and formally prove or refute generated candidate invariants and speed up this process [24, 12]. Many techniques use constraint-based reasoning to create invariants: be it to create linear invariants [16, 30], to discover inductive invariants [20, 49], or to improve on CEGAR verification methodologies [13]. More recently, learning techniques have been employed in the process of invariant and pre-condition generation [27, 26, 44, 40].

6 Conclusions

We have conceived a fully automatic and sound approach to data refinement of smart contract specifications. As illustrated in Figure 1, the input to the strategy are abstract and concrete specifications, S and S', and their corresponding implementations C and C'. From executions of C and C', we use Alloy to infer a candidate abstraction invariant $\hat{\alpha}$ whose validity is checked using a solc-verify encoding for specifications S and S'; if this candidate function is valid, we demonstrate that S' (data) refines S. We have provided some evidence of the practical relevance of our approach by applying it to several commits of real smart contracts that implement some of the major Ethereum standards. We were able to improve on existing approaches, particularly concerning the generation of the abstraction invariant in the form of a (possibly quantified) predicate. To the best of our knowledge, this is the first work to propose data refinement at the level of smart contract specifications.

The proposed strategy is sound, but not complete. Soundness is ensured by the fact that the validity of a candidate abstraction invariant is formally (and mechanically) checked by solc-verify, based on the data refinement template we conceived. The lack of completeness comes from both the use of a bounded model finder (i.e. Alloy) and the restriction of the current implementation to find abstraction functions, rather than arbitrary relations.

Our medium-term vision is the development of a fully automatic Trusted Deployer framework to ensure that only smart contracts that conform to given interface specifications can be deployed in a blockchain. Particularly, we aim to integrate our results into the Trusted Deployer architecture proposed in [6] so that specification refinement is automatically proved using this technique. This way, smart contract evolution involving data refinement can be transparently proved sound, refraining the user from manually providing abstraction invariants that are known to be complex and error-prone,

As another promising topic for future work, an alternative to the strategy we have proposed here is to infer the abstraction invariant directly from the specifications S and S'. The invariant inferred in this context is also a constructive proof that S' data refines S. Furthermore, this would allow one to calculate a version of the implementation C' from C and from the abstraction invariant. The developer could then further refine C' considering only algorithmic refinement. This is very much aligned with our view that, during the life cycle of a smart contract, it seems inevitable to allow an implementation to evolve, despite the widespread *code is law* paradigm in the blockchain context. It also seems unavoidable to prevent even a specifications from evolving, via interface extension (new functions) or to use a different data representation. The paradigm shift we advocate is *conformance is law*: the concrete contract pair (S', C') must preserve some conformance notion with respect the previously committed pair (S, C) in the context of a trusted deployer, as is our major concern here.

References

1 Ethereum White Paper. https://github.com/ethereum/wiki/wiki/White-Paper.

2 Wolfgang Ahrendt and Richard Bubel. Functional verification of smart contracts via strong data integrity. In *ISoLA 2020*, pages 9–24, Cham, 2020. Springer International Publishing.

3 Sefa Akca. *Automated Testing for Solidity smart contracts*. PhD thesis, The University of Edinburgh, 2022.

4 Leonardo Alt and Christian Reitwiessner. SMT-based verification of solidity smart contracts. In *ISoLA 2018*, pages 376–388. Springer, 2018.

5 Pedro Antonino, Juliandson Ferreira, Augusto Sampaio, and A. W. Roscoe. Specification is law: Safe creation and upgrade of ethereum smart contracts. In *SEFM 2022*, volume 13550 of *Lecture Notes in Computer Science*, pages 227–243. Springer, 2022.

6 Pedro Antonino, Juliandson Ferreira, Augusto Sampaio, AW Roscoe, and Filipe Arruda. A refinement-based approach to safe smart contract deployment and evolution. *Software and Systems Modeling*, pages 1–37, 2024.

7 Pedro Antonino and A. W. Roscoe. Solidifier: Bounded model checking solidity using lazy contract deployment and precise memory modelling. In *SAC 2021*, SAC '21, page 1788–1797, 2021.

8 Filipe Arruda, Pedro Antonino, Augusto Sampaio, and AW Roscoe. Solver-aided inference of abstraction invariant for the safe evolution of smart contracts. Technical report. URL: https://github.com/fmca/SEFM24/blob/main/SEFM24-extended.pdf.

9 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *POST 2017*, pages 164–186. Springer, 2017.

10 Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. Monitoring smart contracts: Contractlarva and open challenges beyond. In *RV 2018*, volume 11237 of *LNCS*, pages 113–137. Springer, 2018.

11 Sidi Mohamed Beillahi, Gabriela Ciocarlie, Michael Emmi, and Constantin Enea. Behavioral simulation for smart contracts. In *PLDI 2020*, page 470–486, New York, NY, USA, 2020. ACM.

12 Adam Betts, Nathan Chong, Pantazis Deligiannis, Alastair F. Donaldson, and Jeroen Ketema. Implementing and evaluating candidate-based invariant generation. *IEEE Transactions on Software Engineering*, 44(7):631–650, 2018. `doi:10.1109/TSE.2017.2718516`.

13 Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 300–309, New York, NY, USA, 2007. Association for Computing Machinery. `doi:10.1145/1250734.1250769`.

14 Christie Bolton. Using the Alloy Analyzer to verify data refinement in Z. *Electronic Notes in Theoretical Computer Science*, 137(2):23–44, 2005. Proceedings of the REFINE 2005 Workshop (REFINE 2005).

15 Alexandre Boulgakov, Thomas Gibson-Robinson, and A. W. Roscoe. Computing maximal weak and other bisimulations. *Formal Aspects of Computing*, 28(3):381–407, 2016.

16 Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 420–432, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

17 Jim Davies and Jim Woodcock. Using Z. *Specification Refinement and Proof. Series in Computer Science*, 1996.

18 Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, Vikram Saraph, and Eric Koskinen. Proof-carrying smart contracts. In *Financial Cryptography Workshops*, 2018.

19 David L. Dill, Alan J. Hu, and Howard Wong-Toi. Checking for language inclusion using simulation preorders. In *CAV 1992*, pages 255–265. Springer Berlin Heidelberg, 1992.

20 Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &amp; Applications*, OOPSLA '13, page 443–456, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2509136.2509511`.

21 M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001. `doi:10.1109/32.908957`.

22 Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. Automated discovery of simulation between programs. In *LPAR 2015*, pages 606–621. Springer Berlin Heidelberg, 2015.

23 Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. Property directed equivalence via abstract simulation. In *CAV 2016*, pages 433–453, Cham, 2016. Springer International Publishing.

24 Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, pages 500–517, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

25 P. H. B. Gardiner and Carroll Morgan. *A Single Complete Rule for Data Refinement*, pages 111–126. Springer London, London, 1992.

26 Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 69–87, Cham, 2014. Springer International Publishing.

27 Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 499–512, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2837614.2837664`.

28 Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Ethertrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep*, 2018.

29 Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. What are the actual flaws in important smart contracts (and how can we find them)? In *FC 2020*, pages 634–653, Cham, 2020. Springer International Publishing.

30 Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 634–640, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

31 Ákos Hajdu and Dejan Jovanović. SMT-Friendly formalization of the solidity memory model. In *ESOP 2020*, pages 224–250. Springer, 2020.

32 Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. In *VSTTE*, pages 161–179. Springer, 2020.

33 J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined resume. In *ESOP 86*, pages 187–196, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

34 M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS 1995*, pages 453–462, 1995.

35 Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *CSF 2018*, pages 204–217. IEEE, 2018.

36 Dorit S Hochbaum. Approximation algorithms for the set covering and vertex cover problems. *SIAM Journal on computing*, 11(3):555–556, 1982.

37 Bin Hu, Zongyang Zhang, Jianwei Liu, Yizhong Liu, Jiayuan Yin, Rongxing Lu, and Xiaodong Lin. A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. *Patterns*, 2(2):100179, 2021.

38 Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

39 Cliff B Jones. *Systematic software development using VDM*, volume 2. Prentice Hall Englewood Cliffs, 1990.

40 Soonho Kong, Yungbum Jung, Cristina David, Bow-Yaw Wang, and Kwangkeun Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In Kazunori Ueda, editor, *Programming Languages and Systems*, pages 328–343, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

41 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS 2016*, pages 254–269. ACM, 2016.

42 Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *ASE 2019*, pages 1186–1189. IEEE, 2019.

43 ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 605–615, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3106237.3106281`.

44 Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 42–56, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2908080.2908099`.

45 Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *S&P 2020*, pages 18–20, 2020.

46 Francesco Ranzato and Francesco Tapparo. An efficient simulation algorithm based on abstract interpretation. *Information and Computation*, 208(1):1–22, 2010.

47 Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. *ER*, 98:449–464, 1998.

48 Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Evmpatch: Timely and automated patching of ethereum smart contracts. In *USENIX Security 21*, pages 1289–1306. USENIX Association, August 2021.

49 William Schultz, Ian Dardik, and Stavros Tripakis. Plain and simple inductive invariant inference for distributed protocols in tla$^+$. In Alberto Griggio and Neha Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 273–283. IEEE, 2022. URL: `https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_34`, `doi:10.34727/2022/ISBN.978-3-85448-053-2\_34`.

50 David Siegel. Understanding the dao attack. Available at: `https://www.coindesk.com/understanding-dao-hack-journalists` accessed on 25 September 2023.

51 J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

52 Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *WETSEB 2018*, pages 9–16. IEEE, 2018.

53 Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Comput. Surv.*, 54(7), 2021.

54 Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *CCS 2018*, pages 67–82. ACM, 2018.

55 Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In *VSTTE*, pages 87–106, 2020.

56 Scott Wesley, Maria Christakis, Jorge A. Navas, Richard Trefler, Valentin Wüstholz, and Arie Gurfinkel. Verifying solidity smart contracts via communication abstraction in smartace. In *VMCAI 2022*, page 425–449, Berlin, Heidelberg, 2022. Springer-Verlag.

A Alloy encoding

To help with the reviewing process, this appendix details the rules and encodings we have used in our Alloy model. We call rules the elements that are generated dynamically based on some input. For instance, the next two rules take as input assignments to primitive and mapping (or array) variables, arising from our associations gathered in the Execution step, and transform these state valuations into Alloy assignments.

At the top of these rules, there is a description of the input they expect, and the consequent of the rule is a template demonstrating how the inputs are used to generate the associated Alloy constraints.

Rule 1. Simple Assignment ⟦var : Variable, inspections : Map(Sim− > Value)⟧ : AlloyModel =

```
one sig name(variable)_Assignment extends Assignment{}{
    variable = name(variable);
    value = values(inspections);
}
values(inspections : Map(Sim− > Value)) : AlloyModel =
  foreach (sim, value) in inspections :
    Sim_sim → ⟦value⟧
```

Rule 2. Mapping Assignment ⟦var : Variable, inspections : Map(Sim → Value)⟧ : AlloyModel =

```
foreach assignment in assignments :
    one signame(var)_Mapping extends Mapping{
    element = foreach key, value in assignment.value : key → ⟦value⟧
}

one sig name(variable)_Assignment extends Assignment{}{
    variable = name(variable);
    value = mappingValues(inspections);
}
mappingValues(inspections : Map(Sim− > Value)) : AlloyModel =
  foreach (sim, value) in inspections :
    Sim_sim → name(var)_Mapping
```

We call encodings the pieces of our model that are static, i.e. they are parts of our Ally model that do not vary and are not generated specifically for a given input. The following encoding describes some basic elements for our encoding such as mapping values (`Mapping`), variable definition (`Variable`), and a variable assignment (`Assignment`).

Encoding 3. Basic Definitions

```
abstract sig Type {}
sig Str, Address, Uint, MappingType extends Type {}
abstract sig Version {}
one sig Abstract, Concrete extends Version {}
abstract sig ExecutionId {}

abstract sig Mapping {
  element: Int -> Int
}

abstract sig Variable {
  name: String,
  type:  Type,
  version: one Version
}


abstract sig Assignment {
    variable: one Variable,
    value: ExecutionId -> (Int + Mapping)
 }
```

The following encoding describes the structure of our expression grammar as per Equation 2.

Encoding 4. Expressions

```
abstract sig Expression {
    variables: set Variable,
    evaluation:  ExecutionId -> one (Int + Mapping)
}
sig SimpleAssignmentExpression extends Expression {
  assignment: one Assignment
}
fact {
  all a: SimpleAssignmentExpression |
    a.variables = a.assignment.variable and
    all executionId : domain[a.evaluation] |
        not (univ.(a.assignment.value) in Mapping) and
        eq[int a.evaluation[executionId], int a.assignment.value[executionId]]
}
abstract sig BinaryExpression extends Expression {
  left: one Expression,
   right: one Expression
}
```

The following encoding describes the arithmetic expressions supported by our encoding.

Encoding 5. Arithmetic Expressions

```
abstract sig ArithmeticExpression  extends BinaryExpression {}
sig SumExpression extends ArithmeticExpression {}
fact {
    all s: SumExpression |
        s.variables = s.left.variables + s.right.variables
        and (all executionId: domain[s.evaluation] |
               s.evaluation[executionId] =
                add[int s.left.evaluation[executionId],
                    int s.right.evaluation[executionId]])
        and no s & (s.^left + s.^right)
}

sig MulExpression extends ArithmeticExpression {}
fact {
    all s: MulExpression |
        s.variables = s.left.variables + s.right.variables
        and (all executionId: domain[s.evaluation] |
               s.evaluation[executionId] =
                mul[int s.left.evaluation[executionId],
                    int s.right.evaluation[executionId]])
        and no s & (s.^left + s.^right)
}

sig DivExpression extends ArithmeticExpression {}
fact {
    all s: DivExpression |
        s.variables = s.left.variables + s.right.variables
        and (all executionId: domain[s.evaluation] |
               s.evaluation[executionId] =
                div[int s.left.evaluation[executionId],
                    int s.right.evaluation[executionId]])
        and no s & (s.^left + s.^right)
}


sig SubExpression extends ArithmeticExpression {}

fact {
    all s: SubExpression |
        s.variables = s.left.variables + s.right.variables
        and (all executionId: domain[s.evaluation] |
               s.evaluation[executionId] = sub[int s.left.evaluation[executionId],
                    int s.right.evaluation[executionId]])
        and no s & (s.^left + s.^right)
}
```

The following encoding describes the indexed expressions we support.

Encoding 6. Indexed Expressions

```
abstract sig IndexedExpression extends BinaryExpression {}
sig SumMapExpression extends IndexedExpression {}

fact {
    all s: SumMapExpression |
        s.variables = s.left.variables + s.right.variables
        and (all executionId: domain[s.evaluation] |
               all index: univ.element.univ |
                    some s.right.evaluation[executionId].element implies {
                        s.evaluation[executionId].element[index] =
                          add[int s.left.evaluation[executionId].element[index],
                            int s.right.evaluation[executionId].element[index]]
                    } else {
                        s.evaluation[executionId].element[index] =
                          add[int s.left.evaluation[executionId].element[index],
                            int s.right.evaluation[executionId]]
                    })
        and no s & (s.^left + s.^right)
}

sig SubMapExpression extends IndexedExpression {}

fact {
    all s: SubMapExpression |
        s.variables = s.left.variables + s.right.variables
        and (all executionId: domain[s.evaluation] |
               all index: univ.element.univ |
                    some s.right.evaluation[executionId].element implies {
                        s.evaluation[executionId].element[index] =
                          sub[int s.left.evaluation[executionId].element[index],
                            int s.right.evaluation[executionId].element[index]]
                    } else {
                        s.evaluation[executionId].element[index] =
                          sub[int s.left.evaluation[executionId].element[index],
                            int s.right.evaluation[executionId]]
                    })
        and no s & (s.^left + s.^right)
}



sig MulMapExpression extends IndexedExpression {}

fact {
    all s: MulMapExpression |
        s.variables = s.left.variables + s.right.variables
        and (all executionId: domain[s.evaluation] |
           all index: univ.element.univ |
                some s.right.evaluation[executionId].element implies {
                    s.evaluation[executionId].element[index] =
                        mul[int s.left.evaluation[executionId].element[index],
                            int s.right.evaluation[executionId].element[index]]
                } else {
                    s.evaluation[executionId].element[index] =
                        mul[int s.left.evaluation[executionId].element[index],
                            int s.right.evaluation[executionId]]
                })
        and no s & (s.^left + s.^right)
}
```

Finally, we have an encoding of our abstraction function as a set of equations, with constraints that properly enforce that the equations that we find, be it indexed or not, must respect the associations (i.e. state pairs) that our rules have encoded.

Encoding 7. Abstraction Invariant

```
one sig AbsFun {
 equations: set Equation
}
abstract sig Equation {
 abstractVar: one Variable,
 concreteExpr: one Expression
}

fact {
    abstractVar.version = Abstract
    concreteExpr.variables.version = Concrete
}

sig Equals extends Equation {} {
  abstractVar.type != MappingType and
  concreteExpr in (ArithmeticExpression + SimpleAssignmentExpression) and
  all executionId: domain[concreteExpr.evaluation + variable.abstractVar.value] |
    abstractVar.type = Uint
        and eq[concreteExpr.evaluation[executionId],variable.abstractVar.value[executionId]]
}

sig EqualsMapping extends Equation {} {}
fact {
  all eqmap: EqualsMapping  |
        eqmap.concreteExpr in (IndexedExpression + MappingAssignmentExpression) and
        eqmap.abstractVar.type in MappingType and
        all executionId: domain[univ.concreteExpr.evaluation] |
            all index: (variable.(univ.abstractVar).value[executionId].element).univ  |
                eq[int eqmap.concreteExpr.evaluation[executionId].element[index],
                    int variable.(eqmap.abstractVar).value[executionId].element[index]]

}

fun domain [r: SimulationId -> (Int + Mapping)] : set SimulationId {
  r.univ
}
```