Rich Specifications for Ethereum Smart Contract Verification

CHRISTIAN BRÄM, ETH Zurich, Switzerland
MARCO EILERS, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland
ROBIN SIERRA, ETH Zurich, Switzerland
ALEXANDER J. SUMMERS, University of British Columbia, Canada

Smart contracts are programs that execute inside blockchains such as Ethereum to manipulate digital assets. Since bugs in smart contracts may lead to substantial financial losses, there is considerable interest in formally proving their correctness. However, the specification and verification of smart contracts faces challenges that do not arise in other application domains. Smart contracts frequently interact with unverified, potentially adversarial outside code, which substantially weakens the assumptions that formal analyses can (soundly) make. Moreover, the core functionality of smart contracts is to manipulate and transfer resources; describing this functionality concisely requires dedicated specification support. Current reasoning techniques do not fully address these challenges, being restricted in their scope or expressiveness (in particular, in the presence of re-entrant calls), and offering limited means of expressing the resource transfers a contract performs.

In this paper, we present a novel specification methodology tailored to the domain of smart contracts. Our specification constructs and associated reasoning technique are the first to enable: (1) sound and precise reasoning in the presence of unverified code and arbitrary re-entrancy, (2) modular reasoning about collaborating smart contracts, and (3) domain-specific specifications based on resources and resource transfers, which allow expressing a contract's behaviour in intuitive and concise ways and exclude typical errors by default. We have implemented our approach in 2Vyper, an SMT-based automated verification tool for Ethereum smart contracts written in the Vyper language, and demonstrated its effectiveness in succinctly capturing and verifying strong correctness guarantees for real-world contracts.

Additional Key Words and Phrases: Ethereum, smart contracts, specification, software verification, resources

1 INTRODUCTION

Smart contracts are programs that execute inside blockchains such as Ethereum, and allow the execution of resource transactions between different parties without the need for a trusted third party. Smart contracts tend to be comparatively short but non-trivial programs, and since any bugs can and do frequently lead to the loss of potentially-large amounts of money [Güçlütürk 2018], using formal methods to ensure their correctness is both practically viable and highly desirable.

Compared to other programs, smart contracts have some peculiarities that are not sufficiently supported by classical specification and verification techniques. (1) Smart contracts frequently interact with other smart contracts, for instance, to perform transactions. These other contracts are typically developed by unknown parties and cannot be assumed to be verified; they might even exhibit adversarial behaviour to gain a financial advantage. As a result, standard modular reasoning techniques such as separation logic [Reynolds 2002], which reason about calls under the assumption that *all* code is verified, do not apply in this setting. This problem is exacerbated by the presence of *re-entrant* calls, i.e. callbacks from functions called by the contract itself. A sound reasoning technique must account for *all* behaviours a call to an unverified contract could possibly exhibit, which requires novel ways of specifying the calling contract. (2) Many contracts collaborate with other contracts,

Authors' addresses: Christian Bräm, c.braem@gmx.ch, ETH Zurich, Switzerland; Marco Eilers, marco.eilers@inf.ethz.ch, ETH Zurich, Switzerland; Peter Müller, peter.mueller@inf.ethz.ch, ETH Zurich, Switzerland; Robin Sierra, robin.sierra@outlook.com, ETH Zurich, Switzerland; Alexander J. Summers, alex.summers@ubc.ca, University of British Columbia, Canada.

forming distributed applications. To decouple collaborating contracts, their implementations are often hidden behind interfaces. To enable *contract-modular* verification, these interfaces need to be equipped with specifications that provide enough information to allow verification. (3) Typical smart contracts are primarily concerned with modelling and executing resource transactions of different kinds. Examples range from simple token contracts to escrow implementations, ICOs, and complex decentralised finance (DeFi) applications. However, even though notions such as resource ownership and agreed exchanges are central to programmer intentions, resources themselves are often implicit in smart contract implementations. This discrepancy between high-level intentions and low-level implementations easily leads to subtle and potentially-costly mistakes.

Existing (automated) verifiers for smart contracts do not fully address these challenges. Some verifiers [Feist et al. 2019; Lai and Luo 2020; Tikhomirov et al. 2018; Tsankov et al. 2018] prove specific properties, e.g. the absence of overflows or re-entrancy bugs, but cannot be used to prove full functional correctness of a contract. Other verifiers [Hajdu and Jovanovic 2019; Hildenbrandt et al. 2018; Kalra et al. 2018; Permenev et al. 2020] aim to prove arbitrary, user-defined properties using variations of established specification and verification techniques. However, because these techniques are not sufficiently adapted to the setting of smart contracts, they are either not generally applicable or very imprecise in the presence of arbitrary re-entrancy. In general, no existing technique allows reasoning modularly about compositions of multiple smart contracts while preserving interface abstractions. Furthermore, existing techniques offer limited support for specification and reasoning in terms of high-level notions of custom resources such as tokens.

In this paper, we propose a novel specification and verification methodology for the sound, unbounded verification of general (safety) properties of Ethereum smart contracts. We offer specification constructs tailored to the domain of smart contracts, enabling users to prove strong functional correctness properties of arbitrary smart contracts, with specifications that capture their intended resource manipulations explicitly.

We make four main contributions:

(1) Reasoning in the presence of unverified code: To the best of our knowledge, we present the first smart-contract verification technique that is sound and precise in the presence of calls to unverified contracts with arbitrary re-entrancy. Our technique: (a) identifies and proves properties which cannot be invalidated by calls to unverified code, including vital properties such as access control (which lacks direct language support and must be implemented manually in smart contracts), and (b) provides specification constructs that abstract over *all* modifications a call can *potentially* make, exploiting language-level encapsulation guarantees.

(2) Modular reasoning about collaborating smart contracts: We demonstrate the challenges of verifying collaborating contract structures and show that our specification methodology can express all required information at the interface level, forming the basis of the first *modular* verification technique for collaborating smart contracts.

(3) Intuitive specifications of resource manipulation: We introduce a specification mechanism for contracts which manage resources and resource transactions. Specifications are expressed directly at the abstraction level of transferring, exchanging, and loaning resources, which results in concise and intuitive specifications and, effectively, makes smart contract specifications more similar to actual business contracts. Ubiquitous properties of resources such as ownership, access control, and non-duplicability are baked into our system, avoiding potentially repetitive and error-prone boilerplate specifications; violations of these properties are found by default. While other blockchain systems build (different kinds of) resources directly into the language [Blackshear et al. 2019], our approach provides additional default guarantees and is the first to enable reasoning about custom resources in smart contracts systems without dedicated language support.

```
beneficiary: address
highestBid: int256
highestBidder: address
ended: bool
pendingReturns: map(address, int256)
lock: bool

def bid():
  assert not self.lock
  assert msg.value > self.highestBid and not self.ended
  self.pendingReturns[self.highestBidder] += self.highestBid
  self.highestBidder = msg.sender
  self.highestBid = msg.value

def withdraw():
  assert not self.lock
  toSend = self.pendingReturns[msg.sender]
  self.pendingReturns[msg.sender] = 0
  self.lock = True
  send(msg.sender, value=toSend)
  self.lock = False

def end():
  assert not self.lock
  assert not self.ended and msg.sender == self.beneficiary
  self.ended = True
  self.lock = True
  send(self.beneficiary, value=self.highestBid)
  self.lock = False
  self.highestBid = 0
```

Fig. 1. Simplified auction contract written in Vyper.

(4) Implementation and evaluation: We implemented our approach in 2Vyper, an automated, SMT-based verification tool for the Vyper language [Ethereum 2021c] for Ethereum smart contracts. To the best of our knowledge, 2Vyper is the first verification tool specifically aimed at Vyper contracts. It supports the entire current Vyper language and allows specifying contracts and interfaces in the form of readable, source-level code annotations. Our evaluation shows that 2Vyper enables automated verification of strong correctness properties of (collaborating) real-world contracts with reasonable performance and annotation overhead. In particular, we demonstrate that 2Vyper can verify contracts that use re-entrancy patterns not supported by other verification tools, and that it enables modular verification of collaborating smart contracts used in practice.

Outline. The paper is structured as follows: We introduce Ethereum smart contracts in Sec. 2. In Sec. 3, we informally introduce the specification constructs we use to reason about contracts containing re-entrant calls; subsequently, we show how they can be used to reason about collaborating contracts in Sec. 4. We introduce our resource-based specification approach in Sec. 5, and present our verification technique in the form of a Hoare logic in Sec. 6. We describe our implementation in 2Vyper and evaluate it in Sec. 7. We discuss related work in Sec. 8 and conclude in Sec. 9.

2 ETHEREUM SMART CONTRACTS

Ethereum smart contracts are programs usually written in a high-level language, most-commonly Solidity [Ethereum 2021b] or the newer Vyper [Ethereum 2021c] language, and then compiled to bytecode for execution in the Ethereum Virtual Machine (EVM) [Wood et al. 2014]. Fig. 1 shows

```
1  minter: address
2  balances: map(address, int256)
3
4  def transfer(from: address, to: address, amount: uint256):
5    assert self.balances[from] >= amount and msg.sender == from
6    newAmount: int256 = self.balances[from] - amount
7    assert newAmount >= 0
8    self.balances[to] += amount
9    self.balances[from] = newAmount
10   to.notify(from, self, amount)
11
12 def mint(to: address, amount: uint256):
13   assert msg.sender == self.minter
14   self.balances[to] += amount
```

Fig. 2. Simplified token contract implemented in Vyper. The minter can create new tokens by calling mint; other users call transfer to give their own tokens to another user. **assert** statements ensure that the transaction reverts if a user tries to spend tokens they do not own.

an example of a Vyper smart contract implementing an auction. Note that in this example and throughout the paper, we will use a simplified presentation of Vyper and Ethereum contracts and omit details that are irrelevant to our approach (e.g. that Ether can be transferred only by calling functions marked as payable, or that Vyper functions revert when encountering under- or overflows[1]). We also ignore the fact that contract execution consumes *gas*, i.e. a fixed cost associated with every executed instruction, which is not relevant for proving safety properties.

A contract can declare *fields* that form its persistent state. In our example, the contract stores the beneficiary of the auction, the current highest bid and bidder, and the amounts of *wei*, a sub-unit of Ethereum's built-in currency *Ether*, it owes to previous bidders who have been outbid. In addition to the explicitly declared fields, every contract has a built-in balance field that tracks the amount of Ether currently held by the contract. Unlike ordinary fields, which can be written to directly by the contract (but, crucially, *not* by other contracts), the balance cannot be written to directly. Note that the Ether currency with its sub-unit wei is the only resource with native language support; all other resources must be implemented by the programmer.

Contracts define a set of functions and a special constructor function called __init__ that is executed when the contract is set up. Smart contracts are executed as *transactions*: a caller outside the blockchain can request to invoke a contract's function, and miners can then decide to execute this function as part of the next block. If this happens, the function is executed, and may in turn call functions of the same or other contracts as part of the same transaction[2]. (Note that throughout this paper, we inline internal calls to private functions for simplicity.) External calls typically occur via *interfaces* that list (a subset of) the available functions of a contract. Importantly, there is no observable concurrency while all transitively-called functions are executed.

The intended workflow of the auction contract is that clients call the bid function and transfer along a larger amount of Ether than the current highest bid. If another client bids a higher value later, the contract updates pendingReturns to remember that no-longer-highest bidders can get their Ether back. Such a bidder can call withdraw to have the Ether transferred back to them. Contracts can transfer Ether to other contracts via send commands (e.g. in function end), where the parameter value specifies the transferred amount.

[1]Our tool allows one to verify that a function does not revert due to under- or overflows.

[2]Throughout this paper, when we say that to our contract interacts with other contracts, we mean other contract *instances*, i.e. contracts deployed at other addresses that may contain the same or (usually) different code than our contract.

Ethereum transactions can *revert*, meaning that they abort and all state changes they made are reset, for several reasons. In particular, contracts themselves commonly use **assert** commands to revert the transaction if the asserted condition is false. This is intentionally-possible behaviour and used to enforce that e.g. arguments supplied by the caller are valid. As an example, a call to the end function will revert if the auction is already over and bid will revert if the new bid is not higher than the current highest bid.

In addition to the contract's fields and explicitly declared arguments, a contract can always access the implicit arguments msg and block, which contain information about the current call and the block the current transaction is a part of. For example, msg has the particularly important field msg.sender, which contains the address of the caller of the current function. Function end uses this variable to ensure that only the beneficiary of the auction can end the auction, whereas the bid function uses msg.value to obtain the amount of Ether sent with the call.

Custom resources. While the auction contract works directly with the built-in Ether currency, many real contracts implement or work with *tokens* [Vogelsteller and Buterin 2015], i.e., custom currencies tracked via ad-hoc implementations in smart contract fields. Fig. 2 shows a very simple version of a token contract. Its state consists of a map that represents the balances that each other contract holds for this token. Contracts can call transfer to transfer tokens from one contract to another, which simply corresponds to updating the map. This contract enforces important properties common to resources in general: Each client holding a balance should *be able to transfer only tokens that it owns*. This implicit notion of resource *ownership* (tracked via numeric values in a map, here) is a native notion in our specification methodology, explained in Sec. 5. This contract's implementation enforces this intention by reverting if it is asked to transfer tokens away from anyone except the caller. Similarly, the right to mint *new* tokens is restricted to a special privileged contract (represented by its *address*), self.minter.

The token contract also illustrates the infamous *re-entrancy* vulnerabilities: the subtle potential for a called contract to perform malicious callbacks and achieve undesirable outcomes. Say, for example, that lines 9 and 10 in the token contract were swapped, i.e., the contract first calls the receiver contract to notify it that it has received tokens, *before* reducing their balance. If the notified contract called the sender of the transaction, it could in turn call back into the token contract and transfer the tokens it just transferred away *a second time*; in particular, the **assert** on line 5 would not prevent the transfer because the balance was not yet updated at the time the callback happens. This allows clients of the token contracts to create tokens out of thin air. Variations of this pattern are behind most re-entrancy vulnerabilities, e.g., the infamous DAO exploit [Güçlütürk 2018]; as we will show in Sec. 5, our explicit resource reasoning will uncover such coding errors by default.

3 VERIFICATION IN THE PRESENCE OF UNVERIFIED CODE AND RE-ENTRANCY

Smart contracts frequently interact with other contracts; for example, the auction contract above has to send Ether to (i.e. call) any contract that has previously placed a bid, has been outbid, and calls the withdraw function. While calls to functions of the same contract (*internal* calls) can simply be verified by inlining the callee function, calls to other contracts (*external* calls) are challenging for two reasons. First, as we explained earlier, the implementations of other contracts are in general not verified, and so we cannot reason about external calls using e.g. standard pre- and postconditions. Second, the implementations of other contracts are in general not known: we cannot make any assumptions about the callbacks they perform directly or via other contracts[3]. In the simplest case, an external call modifies the state of its contract and immediately returns. However, calls can trigger

[3]It is common to limit the amount of gas sent with a call s.t. it is not sufficient to perform callbacks, but relying on this is considered bad practice, since the gas cost of Ethereum Virtual Machine instructions can change.

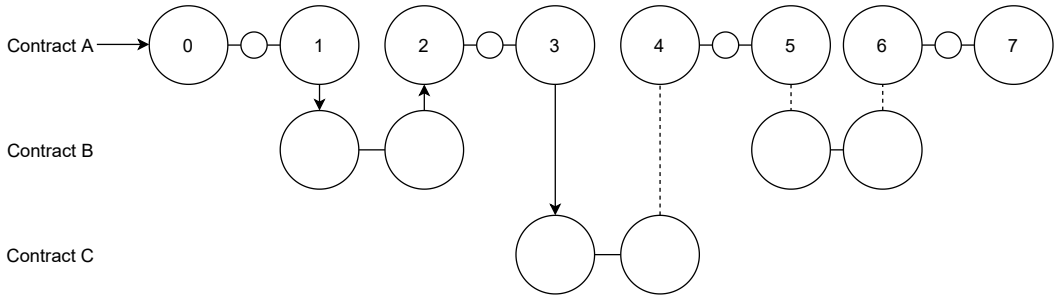

Fig. 3. Example of a smart contract transaction involving re-entrancy. Calls and returns are marked by arrows and dashed lines, respectively.

more complicated interactions such as those shown in Fig. 3, where contract A calls contract B, which performs a re-entrant call to A; subsequently, A performs an external call to a third contract C before all calls return. In this section, we present a specification and verification technique that is sound in the presence of unverified code and arbitrary re-entrancy. In the following, we assume that all calls are external (internal calls are inlined) and that the implementation of the callee is neither known nor verified (known or verified callees could provide stronger assumptions about the effects of the calls, but we focus on the common, most difficult case here).

3.1 The Challenge

The problem of re-entrancy by itself is not specific to smart contracts; it can also occur, for example, between objects in object-oriented programming languages. When reasoning about programs in such languages, this problem is usually solved by constraining what a called function may assume about the state and which parts of the heap it may manipulate [Barnett et al. 2004; Kassios 2006; Müller 2002; Reynolds 2002]. However, these verification techniques require that *all* executed code is verified (and therefore known to adhere to the rules of the verification technique). In a smart contract setting, however, external code is often unverified, potentially malicious, and cannot be soundly assumed to follow *any* particular rules beyond those of the execution environment. For the same reason, classical *preconditions* on public functions are of limited use in this setting, since one cannot rely on external callers actually respecting them. In order to reason soundly, we have to conservatively assume that any external call may lead to arbitrary callbacks into the original contract and, in particular, mutate the original contract's state in any possible way.

Some existing reasoning techniques for smart contracts either assume [Permenev et al. 2020] or aim to prove [Albert et al. 2020; Grossman et al. 2018] that re-entrancy cannot lead to behaviours that cannot also occur without re-entrancy (i.e. that contracts are *effectively callback-free* (ECF) [Grossman et al. 2018]). However, these techniques do not apply to the increasing number of contracts that use re-entrant calls as an essential part of their intended workflow (e.g. [Minacori 2020]). In contrast, our methodology applies to *all* contracts, even if they are not ECF. A central resulting challenge that our work addresses is to be able to express and prove critical properties of a contract despite the potential for arbitrary re-entrancy, as we explain next.

3.2 Specification and Verification Technique

We propose to use a two-pronged approach to smart contract specification and verification: (1) We introduce a novel specification construct that lets us specify constraints on how a contract manipulates its own state. These constraints can be verified without considering external calls. (2) We

introduce two additional specification constructs that allow us to reason about external calls, even when the callee functions are unverified and potentially trigger re-entrant calls. Both ingredients exploit a key feature of smart contracts that differs from most standard object-oriented languages: *All* contract state is private, i.e. cannot be directly modified by functions in other smart contracts. In particular, all updates to a contract's local state are performed by the contract's own code.

Reasoning about call-free code. Since the state of a contract can be modified only by its own functions, one can express many important properties as constraints on local code, that is, the call-free code segments between (external) calls. For instance, if *all* such code segments of a contract only ever increase the value of a counter, its value will never get smaller, even when external functions (with potential re-entrancy) are called. We refer to call-free sequences of statements as *local segments*; in a sense, they represent the atomic operations a contract can perform. In Fig. 3, the local segments of contract A are the ones between the state pairs (0, 1), (2, 3), (4, 5) and (6, 7).

One class of properties that can be enforced by imposing constraints only on local segments is access control, i.e. restricting the right to perform certain operations or modify certain data to specific callers. Access control is particularly important for smart contracts, since, unlike in standard object-oriented programs, they have to store *all* the data of their clients in their own storage (e.g. the balances in the token contract), making it vital to enforce that clients can only modify parts of that storage that conceptually belong to them. Access control restrictions are therefore a necessary part of the public specification of a contract. For example, for the auction contract from Fig. 1 we may want to prove that only the beneficiary can end the auction.

To express constraints on local segments, we introduce *segment constraints*—two-state assertions on the local state of a contract that must hold between the start and end states of *each* local segment in the contract. That is, segment constraints are specified per contract, not per individual segment. In our auction example, we can express the access restriction for the ended field using the segment constraint msg.sender $\neq$ **old**(self . beneficiary) $\Rightarrow$ self .ended = **old**(self .ended). Since, by definition, there are no external calls between the start and end states of a local segment, we can verify segment constraints without considering external, unverified code.

Reasoning about (external) calls. An external call may modify the state of the calling contract C only via one or several re-entrant calls. These re-entrant calls perform the modifications of C's state by executing an arbitrary number of C's local statements (in Fig. 3, the segments (2, 3) and (4, 5)). Consequently, the effect of an external call can be soundly approximated by the reflexive, transitive closure of C's segment constraints. In the following, we introduce two complementary forms of such transitive constraints, which are useful for expressing different kinds of common properties. Both are *auxiliary* specifications in the sense that they do not directly express vital correctness properties (unlike e.g. segment constraints prescribing access control properties), but instead allow us to preserve properties across external calls.

Transitive segment constraints. The reflexive and transitive closure of all segment constraints of a contract describes the effect of an arbitrary number of local segments. Thus, it is known to hold between the pre-state and the post-state of any external call (i.e. between states 1 and 6 in Fig. 3) and can be used to reason about such calls. Since it is generally not possible to compute the reflexive, transitive closure of our segment constraints automatically, we allow programmers to specify the *transitive segment constraints* of a contract. These are checked to be reflexive and transitive, and are verified to hold across each local segment of the contract. Like segment constraints, transitive segment constraints are two-state assertions on the local state of a contract, similar to history constraints [Liskov and Wing 1993]. Since *any* sequence of local segments is guaranteed to satisfy the transitive segment constraints of a contract, they may soundly be assumed to hold between the

pre- and poststate of each external call. Note that transitive segment constraints do *not* subsume ordinary segment constraints, since typical segment constraints used for access control (e.g. the restriction on who can end an auction, shown above) are not transitive.

Transitive segment constraints are useful to express constancy or monotonicity properties, such as the fact that the auction's beneficiary never changes, or that once the auction is ended, it will not be restarted. The latter can be expressed using the assertion $\mathbf{old}(\mathsf{self.ended}) \Rightarrow \mathsf{self.ended}$; this transitive segment constraint e.g. allows us to prove the postcondition $\mathsf{self.ended}$ for function end (even without the lock, which we will discuss below).

Transitive segment constraints subsume single-state *contract invariants*, which are often useful to specify consistency conditions on contract states, which must hold whenever the contract relinquishes control to other contracts (and, thus, its state becomes observable to the environment). The verification of transitive segment constraints implies that single-state contract invariants hold at the end of each local segment, which includes the state before any call as well as the post-state of each function. Consequently, each function may soundly assume such contract invariants to hold in its pre-state, as well as after the return of each external call, analogously to class or object invariants in object-oriented programs [Drossopoulou et al. 2008; Leavens et al. 2008]. For the auction, an important invariant is that its funds suffice to pay all its obligations, which can be written as the transitive segment constraint $\mathsf{self.balance} \geq \mathit{sum}(\mathsf{self.pendingReturns}) + \mathsf{self.highestBid}$.

Function constraints. It is common that each individual function of a contract *as a whole* satisfies a two-state property, even if some of its local segments do not. Such situations occur for instance if *some* sequences of local segments violate the property, but no function in the contract ever executes such a sequence. The "re-entrancy lock" in the auction contract is an example: The field $\mathsf{self.lock}$ is set to true by withdraw and end before their calls to send, and reset to false afterwards. Since each function of the contract reverts if the locks is set, this pattern ensures that each function of the auction contract leaves the contract state completely unchanged if the lock is set in its pre-state.

However, this property cannot be verified as a transitive segment constraint: Some local segments reset the lock, such that any subsequent state change violates the property. That is, the property does not hold for arbitrary sequences of local segments, but it does hold between the pre-state and the post-state of each contract function. Note that any external call can modify the contract state only by executing these contract functions (via re-entrant calls) from start to finish.

To exploit this fact, we introduce *function constraints*: two-state assertions on the local state of a contract that must hold between the pre- and post-state of every function in the contract. In Fig. 3, this means they have to hold between states 2 and 5 as well as states 0 and 7. Like transitive segment constraints, function constraints are specified per contract; they must be satisfied by *all* of its functions (reflecting that we do not know statically which re-entrant calls are triggered by an external call). Since external calls may trigger the execution of an arbitrary number of contract functions, we require function constraints to be reflexive and transitive. For the lock example, we can express the desired property as the function constraint $\mathbf{old}(\mathsf{self.lock}) \Rightarrow \mathsf{self} = \mathbf{old}(\mathsf{self})$.

Note that function constraints do *not* subsume transitive segment constraints. For instance, in the special case of single-state assertions, transitive segment constraints (that is, the contract invariants discussed above) are known to hold before each call and may, thus, be assumed in the pre-state of each function, whereas function constraints may not. Neither do transitive segment constraints subsume function constraints, as we illustrated with the lock example above.

With these two specification constructs, we can modularly verify important properties in the presence of calls to unverified contracts with arbitrary re-entrancy. In this challenging setting, it is nevertheless necessary to design contracts carefully such that they satisfy strong transitive

```
balances: map(address, int256)

def transfer(from: address, to: address, amount: uint256):
  pass
```

Fig. 4. Minimal interface of the token contract in Fig. 2.

segment constraints and function constraints. In Sec. 5, we will complement these constraints with effect specifications on the resources held by a contract to obtain even stronger guarantees.

4 INTER-CONTRACT INVARIANTS

The specification primitives introduced in the previous section all constrain the state of individual contracts. However, smart contract applications are frequently implemented via sets of collaborating contracts. Such collaborations may require maintaining invariants that *relate* the states of the individual contracts. As an example, consider a modified version of the auction contract from Fig. 1 that deals in tokens of our token contract (Fig. 2) instead of Ether. In such a version, calls to send would be replaced by calls to the token contract's transfer function. As before, the modified auction needs to maintain the invariant that it owns sufficient funds (i.e. tokens) to pay all its obligations, which is now an invariant that involves the states of both the auction contract *and* the token contract: self.highestBid + *sum*(self.pendingReturns) ≥ self.token.balances[self][4].

While such *inter-contract invariants* could in principle be regarded as relations between two or more equal contracts, we break this symmetry and designate, for each such invariant, one contract as the *primary* contract, and all others as *secondary* ones. Intuitively, the primary contract is the one whose correctness relies on the invariant to hold and which is responsible for maintaining it. In the example, the modified auction contract is clearly the primary contract, since the auction's funds must be sufficient for the auction to function correctly, whereas the token contract can have many clients with different functionality and is not responsible for their correctness. This asymmetry is reflected in the above invariant, where self is the auction contract.

Ensuring non-trivial inter-contract invariants requires that all contracts related by the invariant are verified; the state of unverified contracts may change arbitrarily, which precludes the verification of invariants that depend on it[5]. However, we do *not* require to have access to the implementations of the secondary contracts. These may be hidden behind interfaces (a built-in language mechanism in both Vyper and Solidity for abstracting over multiple implementations, see e.g. ERC20 [Vogelsteller and Buterin 2015]). Interfaces declare that a contract offers *at least* some set of functions and fields[6], but do not give any information about their implementation, or preclude the existence of additional functions in the contract. Fig. 4 shows a minimal interface of the token contract from Fig. 2.

Modular verification of an inter-contract invariant consists of proving three main parts. Part 1: The implementation of the primary contract *establishes* the invariant when the contract is created. Part 2: Each function of the primary contract *preserves* the invariant. These two parts are analogous to the verification of local contract invariants, expressed as transitive segment constraints and discussed in the previous section. Part 3: All functions of the secondary contracts *preserve* the invariant. This last part is the main challenge since the implementation of the secondary contracts

[4]We focus on single-state invariants here for simplicity only: our technical solution also supports two-state constraints.
[5]Note, however, that contracts may still call functions of unverified contracts.
[6]Interfaces actually contain constant functions that guarantee not to modify any state instead of fields, but we model them as fields here to simplify the presentation.

is not available during modular verification and because functions of the secondary contracts may be called by (unverified) third parties without the knowledge of the primary contract.

To illustrate this challenge, consider a scenario in which the token contract has a function steal that lets an arbitrary contract steal another contract's funds: If this function existed, a third party could call it to steal the tokens of the auction contract; if the auction contract also had any pending returns, this would break our inter-contract invariant. However, whether or not the token contract has such a function (or any other function that allows one contract to decrease another contract's funds) *cannot* be concluded from its interface alone.

We address this challenge by strengthening interfaces with such guarantees. More precisely, we allow annotating interfaces with novel *privacy constraints* (in addition to standard function postconditions and transitive segment constraints), which express which part of the contract state conceptually belongs to the caller of a function and, thus, cannot be freely manipulated by other callers. Privacy constraints are specific segment constraints of the form $\forall a.\mathsf{msg.sender} \neq a \Rightarrow P$, where P is reflexive and transitive. The segment constraints specified in an interface must be satisfied by *all* functions of a contract implementing the interface, even those not mentioned in the interface. Therefore, privacy constraints convey strong information on how the contract state may evolve, which we will use to prove that inter-contract invariants are preserved.

The privacy constraint $\forall a.\mathsf{msg.sender} \neq a \Rightarrow \mathsf{self.balances[a]} \geq \mathbf{old}(\mathsf{self.balances[a]})$ on the token interface from Fig. 4 expresses that a caller may increase the balance of any contract, but decrease only its own. This is exactly the information needed to prove that calls on the token contract cannot violate the inter-contract invariant stated at the beginning of this section.

Based on this idea, we verify inter-contract invariants as follows. Transitive segment constraints are now (unlike all other specification constructs we have introduced) allowed to contain inter-contract invariants, i.e. they may now refer to the state of other contracts that are reachable from the primary contract. As before, we check that they are reflexive and transitive, prove that they are established by the primary contract's constructor, and verify that each local segment of the primary contract satisfies them. Moreover, they may be assumed to hold in the pre-state of each function of the primary contract. These checks subsume part 1 above (establishing the inter-contract invariant).

It remains to show how we also ensure that the invariants are preserved by each function of the primary contract (part 2) and of the secondary contract (part 3). In the presence of re-entrancy, "preserved" does not only refer to the pre- and post-state of each function, but also to the states before a call (such that call-backs will not find the contract in an inconsistent state); this requirement is analogous to the verification of object-oriented programs with re-entrancy [Drossopoulou et al. 2008]. Our verification technique ensures an even stronger property: after the termination of the constructor, an inter-contract invariant holds before and after each local segment of any contract.

Functions of the primary contract preserve the invariant (part 2). A function of the primary contract could potentially violate the invariant (a) by mutating its own state, (b) by calling functions of a secondary contract, or (c) by calling other, external functions. Case (a) is prevented by verifying that each local segment of the primary contract preserves the invariant. For case (b), we will show below that functions of secondary contracts preserve the invariant. Finally, case (c) follows from a simple inductive argument: since all functions of the primary and of the secondary contract preserve the invariant, and since the state of a smart contract is private, an external call can neither modify the state of the primary and secondary contracts directly, nor violate the invariant via re-entrant calls; therefore, external calls also preserve the invariant.

Functions of secondary contracts preserve the invariant (part 3). A function of a secondary contract could potentially violate the invariant (a) by mutating its own state, (b) by calling functions of the primary or another secondary contract, or (c) by calling other, external functions. Cases (b) and (c)

follow from the inductive argument laid out in the previous paragraph. We reason about case (a) using the privacy constraints introduced above. Recall that these are segment constraints which are verified for each function of the secondary contract.

We verify that each segment of each function of a secondary contract satisfies the specified privacy constraints. So in order to verify that these segments do not violate the inter-contract invariant, it suffices to prove that the invariant is *stable* under state changes allowed by the privacy constraints of all secondary contracts. For calls that do not originate from the primary contract, we may assume the reflexive, transitive two-state assertion P of each privacy constraint. For calls from the primary contract, the privacy constraint of the callee contract is trivial and does not provide any information, but we may still assume the assertion P of all *other* privacy constraints. We perform the stability proof for each individual function call; besides the privacy constraints, this proof may also exploit other information known about the pre- and post-state of the call, for instance, from function constraints.

We will formally define the notion of assertion stability in Sec. 6 and illustrate it here with an example: Consider the transfer function of the token contract, which is a secondary contract. When called from an external contract (in particular, not the primary auction contract), we verify that the invariant self.highestBid + *sum*(self.pendingReturns) ≥ self.token.balances[self] is stable under the privacy constraint self.token.balances[self] ≥ **old**(self.token.balances[self]), which is sufficient to guarantee that the function preserves the invariant.

On the other hand, if transfer is called from the primary contract (for instance, from the adapted withdraw function), the token contract's privacy constraint does not provide any useful information since it allows arbitrary changes to the caller's balance. However, not knowing the implementation of transfer, we cannot make any strong assumptions about its behaviour and, therefore, cannot prove that it preserves our invariant at the end of each of its local segments. Note that we may know that the invariant holds when the function returns from its postcondition, but that is not sufficient in case it transitively calls back the primary contract before its own return, which would then be in an inconsistent state. Consequently, we need to weaken our invariant by making it conditional on the lock field not being set. This weaker invariant can be proved to be stable since lock is set before the call and, by a suitable function constraint, lock remains set after the call. Therefore, the invariant holds trivially in both states and, thus, is preserved no matter how the function mutates the secondary contract. This use of locks is common in smart contracts and has special language support in Vyper; similar conditional invariants also occur in verification techniques for object-oriented programs with re-entrancy [Leino and Müller 2004].

In summary, we have generalised our previously-introduced specification constructs to verify invariants of collaborating contracts. This verification is fully modular, based on the implementation of the primary contract and specified interfaces for all secondary contracts. It is sound even when these contracts interact with unverified code, and in the presence of arbitrary re-entrancy.

5 RESOURCE-BASED SPECIFICATIONS

As mentioned before, the vast majority of smart contracts in some way models resources and resource transfers, like the token and auction contracts we have seen before. Resources have a number of basic properties that are important for the correctness of every contract that works with them: they cannot be duplicated, they have an owner, and they cannot be taken away from that owner without their consent. Explicitly specifying these properties for every smart contract that uses some sort of resource is possible, but laborious and error-prone. Instead, we propose a dedicated specification and verification technique that has basic resource properties built-in and that offers high-level specification constructs to declare resources and to describe resource

transactions. The potential of resource-based reasoning for smart contracts has been recognized before; for instance, Move [Blackshear et al. 2019] has native support for resources in the blockchain and language. However, our technique is the first one that superimposes a resource view on smart contracts that have no such concept built-in.

Compared to specifications that express resource properties via changes of the contract state, our resource specification system has three main advantages (note that Move, due to it having a different resource model, does not have these three advantages built-in, see Sec. 8):

(1) Safety: Basic properties of resources, like the fact that they cannot be duplicated and cannot be taken away from their current owner without their consent, are baked into the system. Our verification approach ensures that these properties hold by default, without developers having to specify them explicitly. As a result, there is no danger that important properties are missing in the specifications, and there is no need to write them down for every contract.
(2) Higher-level reasoning: Developers think about resources as an abstract concept; for instance, they think of a token as a kind of currency, not some contract whose state contains a map. Resource-based specifications let developers describe their contracts' states and interactions on this abstraction level, leading to simpler and more intuitive specifications.
(3) Client documentation: Writing postcondition-based specifications for smart contract functions is often difficult because of potentially re-entrant calls with unbounded effects. Therefore, our system allows specifying and proving novel effect-based function specifications that give a caller an upper bound on the negative consequences it may suffer from calling a function (e.g. losing some amount of Ether) and a lower bound on the positive consequences (e.g. receiving some number of tokens).

In this section, we describe the basic attributes of our resources, the operations that can be performed on them, and how we connect the contract's actual state to the resource state. We show how effect-based function specifications based on resources give callers extra information. Finally, we describe advanced concepts such as *derived* resources, to represent titles to other resources.

5.1 Resource Model

In our system, a resource can represent anything of value, i.e. something that cannot be duplicated and has an *owner*. Resources are owned by addresses on the blockchain, that is, either contracts or users. Ownership implies control of the resource, i.e. only the owner of a resource can transfer or destroy it. Receiving additional resources does not require consent. In this regard, our resources are similar to Ethereum's built-in Ether resource (which is treated as a built-in resource in our system).

Fig. 5 shows the operations that can be performed on resources, and who is allowed to perform them. Resources can be *created* by privileged parties who have the right to do so (usually called minters). They can be *transferred* to other addresses or *destroyed* by their owners (and by noone else). In addition, addresses can *offer* to exchange some resources against others; once an address has made an offer, and a second party makes a matching counter-offer, the *exchange* can be performed at arbitrary points in time and without further agreement by the involved parties (consuming the offers). An address need not own the resources it offers at the point when it makes the offer, but an exchange requires that both addresses actually hold the offered resources. Addresses can *revoke* previous offers they have made. The resulting set of operations is simple but makes it possible to allow modelling the behaviour of a wide range of real smart contracts.

5.2 Resource State

Every smart contract may declare one or more resources. For each resource, all addresses implicitly have a balance (as for the built-in Ether). Similarly, each address has a set of existing offers on

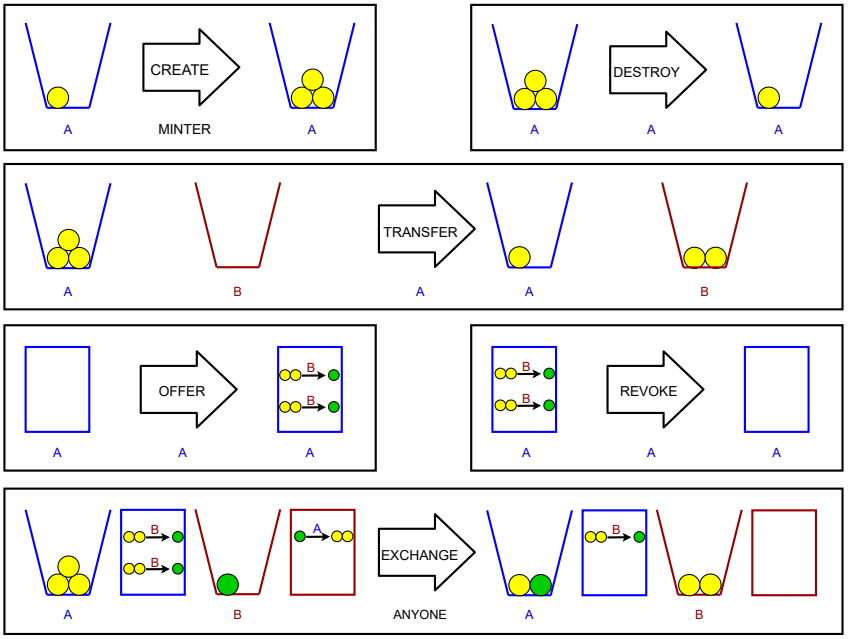

Fig. 5. Operations on resources and offers. The buckets represent resources owned by addresses A and B, respectively; the rectangles contain the offers they have made. The arrows show the name of the resource operation, and the names under the arrows show who is allowed to perform the operation (create-operations may be performed by anyone who has a special creator-resource, representing the right to mint new resources). For example, A (and noone else) can *transfer* two of the three yellow resources it currently owns to B, or it can *offer* B to exchange two of its yellow resources against one of B's green resources at most twice.

the resources it declares. These balances and offer sets are *ghost state*: state that exists only for verification purposes, but is not present at execution time. Our specifications can refer to this state, e.g. a postcondition can refer to the caller's balance for resource R as $\mathsf{balances}_R[\mathsf{msg.sender}]$. Note that specifications about resource state can be arbitrarily combined with the other specification constructs we have introduced; for example, one could write a segment constraint stating that a contract may perform some operation only if it owns some minimum amount of a resource.

The resource ghost state can be changed only by executing *ghost commands*, written in the verified contract, that each perform one of the resource operations mentioned above. As an example, the ghost command $\mathsf{transfer}_R(f, t, a)$ transfers a amount of resource R from f to t. This ghost command requires that f has sufficient amounts of R, and that f is the contract invoking the ghost command, i.e. that f has called the function that contains it (modulo delegation, which we will discuss later). These conditions are checked by the verifier, and they enforce the basic properties of the resource system; the latter check in particular enforces ownership constraints. The resource ghost state has the same encapsulation as ordinary contract state, that is, the ghost commands in a contract can modify only the state of the resources declared in that contract, not of any other resources. A more detailed description of the available ghost commands can be found in App. A.

5.3 Connecting Resource State and Contract State

In order to be useful for verification, the resource ghost state must be connected to the contract's actual state. In particular, the ghost resource operations must be reflected in the actual contract

implementation. More precisely, operations on the contract state must *refine* the operations on the more abstract ghost state. To ensure this, developers need to provide invariants (i.e. transitive segment constraints) that precisely define the resource state (i.e. balances and existing offers) in terms of the contract state. When verifying a contract, we then enforce that these coupling invariants, like all transitive segment constraints, hold at the end of every local segment. For the token contract, the invariant would be $\mathsf{balances}_{token} = \mathsf{self}.\mathsf{balances}$, meaning that the balances of the resource named "token" are recorded in the contract's balances field.

This check essentially forces changes on the resource state and on the contract state to happen in lockstep: If a change happens on the resource state with no equivalent change on the contract state (or vice versa), the invariant cannot be verified. As a result, the properties our system guarantees for resources (like ownership) carry over to the actual contract state. For instance, our system prevents the verification of a function steal that allows arbitrary callers to steal some client's funds: the modification to the contract state must be mirrored in a corresponding change of the resource state (by the coupling invariant). The ghost command that makes this change checks that the transferred funds belong to the function's caller, which would fail here.

5.4 Client Specifications

The system described so far guarantees that others cannot take away the resources owned by a contract. However, the contract itself may perform operations that lead to a loss of resources, e.g. by transferring or destroying them. Any such operation is initiated by the owner calling a function on the contract that declares the resource (e.g. a function on the token contract that contains a transfer ghost command). Therefore, it is vital that functions provide callers with specifications describing how they may affect the caller's resources.

We address this problem by introducing *effects-clauses* on contract functions, which specify which ghost commands will be executed when the function is called (assuming it does not revert). Each function has a multiset of effects, and effects correspond directly to the ghost commands discussed previously, meaning that there are effects for creating, transferring, and exchanging resources, etc. Effects-clauses are unordered and do not give any information regarding *when* during the call the effects occur. As an example, if a function's effects-clause contains $\mathsf{transfer}_R(\mathsf{msg.sender}, t, 4)$ and $\mathsf{transfer}_R(\mathsf{msg.sender}, t, 6)$, this means that after successful execution, it will have transferred 10 R (in two separate steps of 4 and 6) from the caller to address t at some point during the call.

Unlike most traditional effects-systems, where effects are typically transitive, our effects-clauses do not necessarily accumulate *all* effects a function has on the resource state; this would not be achievable when calling unknown code (which may transitively cause unknown effects). However, they are designed in such a way that any negative effect on the resources of an owner A (transfers, making an offer, destroying the A's resources) is caused by a function that (1) is called *directly* by A and (2) declares the negative effect in its effect specification. Consequently, A is aware of *any* potential negative effect on its resources and can abstain from making calls with undesired negative effects. On the other hand, our effect specifications may, but do not necessarily include all neutral or positive effects for the caller. That is, they provide *worst-case information*.

To reach this goal, we enforce that (a) the effects-clause of a function *must* contain the effects of all ghost commands directly in its body, and (b) it *may* declare any additional effects resulting from its own calls to other external functions. To see why this rule is sufficient, consider a function that has a negative effect on a contract's resources. If this effect is achieved *directly* via a ghost operation, it can only affect the caller's resources, because all ghost operations that have a negative effect impose a proof obligation that the resources are owned by msg.sender. According to check (a), this effect will be included in the function's effect clause, meaning that the caller was aware of the effect and allowed it to happen by making the call. Alternatively, if the negative effect is achieved

indirectly via a sequence of additional function calls then, by a simple inductive argument, the sequence must contain another call from the same caller, and that callee function will declare the effect, meaning that (even though the effect was not specified for the original call) the original caller was aware of the consequences of making this other call.

As a result, our effects-clauses enable each contract to know which negative effects a call may have on its resources, such that it can refrain from making calls with undesired effects. This solution gives strong guarantees in the presence of arbitrary re-entrancy, when it is impossible to give the called function a precise postcondition. App. C.2 illustrates the use of effects-clauses in a realistic setting, namely on (an extended version of) the token contract from before.

5.5 Derived Resources

As a final ingredient, our system contains one additional concept to be able to model the difference between physically having a resource and conceptually being its rightful owner. As an example, consider the auction contract again: Whenever a bidder sends some wei to it, that wei now physically belongs to the auction contract, which could in principle do with it whatever it wants. However, conceptually, as long as the auction is running, the bidder is still the owner of the wei it sent, and it rightfully expects to either get it back later (if someone else makes a higher bid) or to exchange it for the auctioned good when the auction ends. That is, after a bid and before the end of the auction, the physical owner of that wei (the auction contract) is different from the conceptual owner (the bidder). This is a relatively common notion that occurs whenever some contract (temporarily) manages another contract's resources, and it obviously comes with certain expectations (e.g. the auction contract should not be able to give the wei it has to anyone but its rightful owners).

Our system has support for this scenario in the form of *derived resources*, representing conceptual ownership of a resource physically owned by someone else; essentially, a kind of title. In our example, the auction contract could declare a resource wei_in_auction derived from the built-in wei resource. When a bidder sends wei to the auction contract, it transfers its wei to it, but gets the same amount of wei_in_auction in return, signifying that it is owed that amount of wei from the auction contract. If another higher bid comes in and the bidder gets its wei back, its titles get destroyed again. In contrast, the winner of the auction exchanges its titles against the auctioned good, so that their bid is now owed to the beneficiary of the auction. At any given point, the amount of titles address c has in the auction contract is $\mathsf{self.pendingReturns}[c] + (\mathsf{self.highestBidder} = c ? \mathsf{self.highestBid} : 0)$, meaning that this is also the amount of wei_in_auction contract c owns.

The existence of an instance of a derived resource is always bound to an instance of the underlying resource it is derived from. That is, if a contract declares resource D derived from another resource R, then an instance of D comes into existence for every instance of R it receives (via a transfer operation or an exchange), and is automatically allocated to the sender of the R; there is no way to create an instance of D without receiving an instance of R. Similarly, whenever the contract sends some amount of R to someone else, this destroys the same number of D instances that other contract currently owns. This mechanism ensures that the contract always owns enough of the original resource to "pay back" its title loans. The reader may recall that this fact was an invariant of the auction contract that we explicitly mentioned in Sec. 3; now, with derived resources, this invariant is checked automatically and does not have to be specified explicitly.

In order to ensure that contracts do not give away resources that (according to an existing title) belong to someone else, our system enforces that the contract may now transfer R to another contract *only* if that other contract already owns a sufficient amount of D, i.e. the original contract already owes the second contract at least the amount to be transferred. As an example, when the auction contract sends some amount toSend of wei to a previous bidder of the auction in line 20, this is allowed only if the bidder currently owns an equal amount of wei_in_auction, and if the

beneficiary has offered to exchange its wei_in_auction back to ordinary wei. If this is the case, then, the moment the send executes, that amount of the beneficiary's wei_in_auction is automatically destroyed, and the offer to exchange it is consumed.

Apart from the aforementioned restrictions, derived resources behave just like other resources. In particular, they can be traded like other resources (e.g. someone could pay for some good in wei_in_auction, meaning that they give the right to get wei back from the auction contract to someone else). This is relevant for some DeFi contracts that give out tokens that represent ownership of some other goods (i.e. derived resources), but are traded as tokens on their own.

Our proof technique enforces the properties listed above by automatically creating and/or destroying instances of the derived resource whenever a contract calls an external function that declares (in its effects-clause) that it performs a transfer or exchange of the underlying resource to or from the calling contract. Sending or receiving wei is a special case but is treated analogously, i.e. when sending wei with a call, this is handled as if the called function declared that it transfers wei away from the calling contract. To avoid that a contract loses resources that conceptually belong to others without its knowledge (which would mean that it cannot perform the aforementioned checks), our system enforces that the contract declaring D cannot make offers to give away R, since such offers could result in the contract losing R-instances (when the exchange happens) at an arbitrary point in the future. App. C.1 shows the auction contract fully annotated with resource specifications, illustrating the use of a derived resource.

5.6 Further Extensions

Our system contains a few more features that we are not able to describe fully for space reasons. The most important is the notion of *delegation*: It is sometimes useful or necessary for collaborating contracts to be able to act in each others' names when interacting with other contracts. To enable this, we allow a contract A to decide to *trust* another contract B w.r.t. outside contract C, meaning that when B interacts with C (and only then), it can perform actions that normally only A would be able to perform (e.g. transfer A's resources to someone else). As a result, all restrictions on who may execute certain ghost command that we have discussed so far are implemented modulo trust. Trusting another address is executed via a ghost command and has a corresponding effect. Since trusting someone weakens the guarantees one has for one's own resources, users must use this feature with caution; however, as with all other potentially negative effects, functions that establish new trust relations must always state that they do so in their effects-clause.

Our core methodology can easily be extended in several further ways; our implementation e.g. has support for resources with identifiers (resources whose instances can be distinguished from one another) useful for modelling non-fungible tokens (NFTs) [Entriken et al. 2018]. Other extensions are possible, e.g. for some contracts it may be useful to have derived resources that represent ownership not of a single resource of a different type, but of different amounts of other resources. This feature (like many others) does not have to be built into the system; it can be emulated by using the existing resource model in combination with additional invariants, segment constraints etc. that represent the additional rights and constraints that would result from such resources. We believe that the set of features we have described gives users a sufficiently expressive model to be able to verify real contracts, while being simple enough for users to reason about.

6 PROOF TECHNIQUE

We summarise here the formalisation of our technique as a separation logic; for space reasons, full details are relegated to App. B. We do so for a simple smart contract language reflecting the core of Vyper, with the following commands:

$$c \quad ::= \quad \texttt{skip} \mid x := e \mid \texttt{self}.f := e \mid x := e.\mathsf{fun}(e, \mathsf{value} = e) \mid c; c \mid \textbf{assert}\ e$$

$$\frac{e_r : T \quad T.\mathsf{fun}(x) \text{ ensures } Q \text{ performs } S \quad S' \subseteq S}{\vdash \left\{\begin{array}{c} \mathsf{TSC}[\mathbf{old}_{last}/\mathbf{old}] \\ \wedge \mathsf{SC}[\mathbf{old}_{last}/\mathbf{old}] \\ \wedge e_O \end{array}\right\} \begin{array}{c} x := e_r.\mathsf{fun} \\ (e_a, \mathsf{value} = e_v) \end{array} \left\{\begin{array}{c} \mathsf{perf}(S')[e_r/\,\mathsf{self}\,][x/\,\mathsf{result}\,] \\ [\,\mathsf{self}\,/\mathsf{msg}.sender][e_a/x] * \\ \mathbf{old}_{call}(e_O) \wedge \\ \mathsf{TSC}[\mathbf{old}_{call}/\mathbf{old}] \wedge \\ \mathsf{FC}[\mathbf{old}_{call}/\mathbf{old}] \wedge \\ Q[e_r/\,\mathsf{self}\,][x/\,\mathsf{result}\,] \\ [\,\mathsf{self}\,/\mathsf{msg}.sender][e_a/x] \wedge \\ e_N \Rightarrow \mathbf{old}(e_N) \end{array}\right\} \quad (SCall)$$

$$\frac{FV(R) \cap mods(c) = \emptyset \quad \vdash \{P\}\, c\, \{Q\} \quad R \text{ is stateless if } c \text{ contains a call}}{\vdash \{P * R\}\, c\, \{Q * R\}} \quad (Frame)$$

$$\vdash \left\{\begin{array}{c} e_a \geq 0 * \\ (a \neq 0 \Rightarrow \\ \mathsf{trusts}\,(e_f, \mathsf{msg.sender}, \mathsf{true})) \\ * \mathsf{owns}_R(e_f, e_a) \end{array}\right\} \mathsf{transfer}_R(e_f, e_t, e_a) \left\{\begin{array}{c} \mathsf{owns}_R(e_t, e_a) * \\ (e_a \neq 0 \Rightarrow \\ \mathsf{trusts}\,(e_f, \mathsf{msg.sender}, \mathsf{true})) \\ * \mathsf{perf}(\,\mathsf{transfer}_R(f, t, a)) \end{array}\right\} \quad (Transfer)$$

Fig. 6. Selected Hoare Logic rules; full rules included in App. B.

To reflect the design of Vyper only fields of self (the current contract) can be assigned; function calls take a second argument representing the amount of wei to send along with a call. We assume a standard expression language with a reserved result identifier (to refer to function results in postconditions); field lookups include those on the implicit msg and block arguments. To express two-state assertions such as our segment constraints, our formalisation includes *labels* l denoting earlier points in execution, and expressions $\mathbf{old}_l(e)$ denoting the value e had at label l. We use three labels: *pre*, representing the pre-state of the current function, *last*, representing the pre-state of the current local segment, and *call*, representing the pre-state of the last call to another contract.

A state Σ has the form $\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle$, where $\mathcal{H}$ is the heap (a partial map from contract addresses and fields to their values), and σ is the current variable store. $\mathcal{E}$ is a multiset of *effects* produced so far by the current function and $\mathcal{R}$ is a record containing the state of all resources declared in the current contract. In particular, for every such resource R, $\mathcal{R}.\mathsf{balances}_R$ maps addresses to their balances, and $\mathcal{R}.\mathsf{offered}_{R \leftrightarrow R'}$ tracks the offers to exchange R against another resource R' declared in the contract. $\mathcal{R}.\mathsf{trusted}$ is a partial map from pairs of addresses to boolean values that represent whether the first address currently trusts the second; expressions can refer to these maps. Finally, $\mathcal{O}$ maps label names to pairs $(\mathcal{H}, \mathcal{R})$ that represent the heap and resource state at label l.

Expression evaluation in a state, denoted by $[\![e]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})}$, is largely standard; most-interestingly, the evaluation of $[\![\mathbf{old}_l(e)]\!]_{(\sigma,\mathcal{H},\mathcal{R},\mathcal{O})}$ is $[\![e]\!]_{(\sigma,\mathcal{H}',\mathcal{R}',\mathcal{O})}$, where $\mathcal{O}[l] = (\mathcal{H}', \mathcal{R}')$.

We now define our assertion language as follows:

$$\begin{array}{rcl} P, Q & ::= & \mathsf{emp} \mid e \mid P * P \mid P \wedge P \mid P \mathrel{-\!\!*} P \mid P \vee P \mid \\ & & \mathsf{perf}(E) \mid \mathsf{owns}_R(e, e) \mid \mathsf{offers}_{R \leftrightarrow R}(e, e, e, e, e) \mid \mathsf{trusts}\,(e, e, e) \end{array}$$

Assertion truth in a state is defined by a judgement $\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models P$ whose cases are given in Fig. 8 of App. B. In contrast to traditional separation logics [Reynolds 2002], we do not use the *linear/separating* aspects of the $*$ and $\mathrel{-\!\!*}$ connectives to govern access to the (already-encapsulated) *heap*, but rather for the resource state and effects concepts added by our methodology. The separating conjunction $P * Q$ splits the resource state and the effects into two parts; the first described by P

and the second by Q. Descriptions of constituent parts of the resource state come via assertions such as $\mathsf{owns}_R(e_o, e_a)$ that prescribe that $\mathcal{R}$ is empty (no offers, no trust, and all balances are zero) *except* for the balance of e_o, which owns exactly e_a of resource R, and that $\mathcal{E}$ is empty (no effects). The (multiplicative) separating conjunction builds up larger descriptions of these states; e.g. $\mathsf{owns}_R(e_o, e_{a_1}) * \mathsf{owns}_R(e_o, e_{a_2})$ is equivalent to $\mathsf{owns}_R(e_o, e_{a_1} + e_{a_2})$. Similarly, $\mathsf{perf}(E)$ states that *exactly* the effects in multiset E have been performed and no others (and the resource state is empty). The interpretation of other assertions is standard for a classical separation-logic; in particular, an assertion e is true only if there are *no* effects in the current state. As a result, assertions always have to describe the effects-state precisely: If a state containing $\mathcal{E}$ fulfils $P * \mathsf{perf}(E)$, and P does not syntactically contain $\mathsf{perf}()$-assertions, then we must have that $\mathcal{E} = E$. This is important to ensure that functions report all effects they directly cause.

To handle the various kinds of two-state specifications our methodology employs (in each of which $\mathbf{old}(e)$ is used to denote evaluation in the appropriate "old" state), we define a judgement $\Sigma_1, \Sigma_2 \models P$ in which Σ_1 represents the appropriate state to use as the old one (e.g. for local segment constraints we use the state at the start of the local segment):

Definition 6.1. For two states $\Sigma_1 = \langle \mathcal{H}_1, \mathcal{R}_1, \mathcal{E}_1, O_1, \sigma_1 \rangle$ and $\Sigma_2 = \langle \mathcal{H}_2, \mathcal{R}_2, \mathcal{E}_2, O_2, \sigma_2 \rangle$ we define $\Sigma_1, \Sigma_2 \models P$ if and only if $\langle \mathcal{H}_2, \mathcal{R}_2, \mathcal{E}_2, O_2[last \mapsto (\mathcal{H}_1, \mathcal{R}_1)], \sigma_2 \rangle \models P[\mathbf{old}_{last}(\_)/\mathbf{old}(\_)]$

Using this notion, we can define two-state assertion reflexivity and transitivity as follows:

Definition 6.2. An assertion P is *reflexive* if, for all Σ_0, Σ_1, if $\Sigma_0, \Sigma_1 \models P$, then $\Sigma_1, \Sigma_1 \models P$. An assertion P is *transitive* if, for all $\Sigma_0, \Sigma_1, \Sigma_2$, if $\Sigma_0, \Sigma_1 \models P$ and $\Sigma_1, \Sigma_2 \models P$, then $\Sigma_0, \Sigma_2 \models P$.

We can now also define assertion *stability*, i.e. the fact that an assertion is preserved by another:

Definition 6.3. An assertion P is *stable under* Q, written $\mathsf{stable}(P, Q)$, if, for all $\Sigma_0, \Sigma_1, \ldots, \Sigma_n$, if $\Sigma_0, \Sigma_1 \models P$ and $\Sigma_i, \Sigma_{i+1} \models Q$ for all $i \in \{1, \ldots, n-1\}$, then $\Sigma_0, \Sigma_n \models P$.

Finally, we need to define the notion of a stateless assertion:

Definition 6.4. An assertion P is *stateless* if it does not refer to the current state (including resource state) or an old state *except* for the pre-state (i.e. only old-expressions with label *pre* are allowed).

We define our proof technique via a Hoare Logic formulation, whose details are given in Figs. 9 and 10 in App. B. Three important example rules are shown here in Figure 6. Rule (*SCall*) is a simplified version of the most important rule (full version in App. B): that for reasoning about calls. Here, ordinary (SC) and transitive segment constraints (TSC) must hold at the end of every local segment; we therefore require them to be true in the precondition of the Hoare triple, using the state labelled "last" as old state. This notion of "last" state is reset in the postcondition to the new current state (we begin a new local segment), as indicated by $e_N \Rightarrow \mathbf{old}(e_N)$ which can be instantiated for any e_N. A similar connection can be made to facts known before the call via e_O.

The frame rule (*Frame*) is non-standard in that it ensures that *no* information about the last state or the current state can be framed around calls; this represents the fact that the entire contract state can change with every call, due to unknown transitive calls. After a call, one may nonetheless assume the transitive segment constraints and function constraints w.r.t. the call's pre-state. To remember information about said pre-state, we use the same trick as before, and allow assuming any expression e_O after a call about its pre-state that was known to be true before the call.

Each ghost command of Sec. 5 gets a corresponding Hoare Logic rule (e.g. (*Transfer*) shown here), which: (a) checks that the participant for whom the command has a negative effect (e.g. giving away resource) trusts the current msg.sender (typically, this is simply who they are), (b) checks that required resources for the command are available, consuming them, (c) adds appropriate new resources in the postcondition assertion, and (d) records the effect that was performed.

```
#@ performs: revoke[token <-> token](1, 0, to=spender)
#@ performs: offer[token <-> token](1, 0, to=spender, times=amount)
@public
def approveAndCall(spender: address, amount: uint256, data: bytes[1024]):
    #@ revoke[token <-> token](1, 0, to=spender)
    self.allowances[msg.sender][spender] = amount
    #@ offer[token <-> token](1, 0, to=spender, times=amount)
    IERC1363Spender(spender).onApprovalReceived(msg.sender, amount, data)
```

Fig. 7. Simplified code example implementing part of ERC1363 with annotations. The code is intended to be re-entrant (onApprovalReceived performs a callback to the ERC1363 contract) and is therefore not ECF. This allows the contract to do in one transaction what would otherwise (in the general case) require two. The keyword performs marks the effects of the function; when calling the function, the caller first revokes any existing offers and subsequently offers to give exactly amount tokens to the spender; the call to onApprovalReceived will then perform the offered exchange.

The rule for constructors (not shown here) performs the necessary checks of transitivity and reflexivity of transitive segment and function constraints, and ensures that transitive segment constraints fulfil stability criteria described previously; all details are included in App. B.

7 IMPLEMENTATION AND EVALUATION

We have implemented our work in 2Vyper, an automated verification tool for the Vyper language. 2Vyper is implemented in Python and available open source[7]; it reuses the standard Vyper compiler to type-check input programs. It encodes Vyper programs and specifications into the Viper intermediate verification language [Müller et al. 2016], and uses Viper's infrastructure and ultimately the SMT-solver Z3 [de Moura and Bjørner 2008] to verify the program or otherwise return errors and counterexamples.

While less commonly used than Solidity, Vyper puts a stronger focus on correctness and simplicity, by preventing some errors on the language level (such as over- or underflows, which automatically revert the transaction, unlike in Solidity) and omitting some language features that make code more difficult to reason about (such as inheritance). 2Vyper supports the entire current Vyper language (and several previous versions) and is intended for full-fledged verification of real-world contracts.

2Vyper specifications are written as ♯@ comments in the source code, and use Vyper syntax wherever possible. Fig. 7 shows a simplified excerpt of a function annotated with specifications and containing ghost commands for resource manipulation. In addition to the core correctness properties we have focused on in this paper, 2Vyper also supports reasoning about additional language features (e.g. events) and has additional specification constructs to prove specific kinds of liveness properties (e.g. that auction bidders can eventually get their Ether back; it is not stuck in the auction contract forever) that can be converted to safety properties for verification.

7.1 Evaluation Examples

We have evaluated our approach on a number of real-world smart contracts focusing on existing contracts written in Vyper as well as those involving pertinent features such as inter-contract collaboration or re-entrancy bugs [Arumugam 2019; Blockchains LLC 2016; Ethereum 2021a,d; Minacori 2021; Permenev et al. 2019; Uniswap 2019]. We manually translated some examples without Vyper implementations from Solidity to Vyper.

[7] https://github.com/viperproject/2vyper

Application	Contract	LOC	Ann.	IF LOC	IF Ann.	T
auction	auction	63	30	-	-	12.72
auction_token	auction_token††	96	37	88	33	23.67
DAO	DAO*	17	2	-	-	5.13
ERC20	ERC20	98	31	67	25	11.51
ERC721	ERC721	178	32	-	-	15.95
ERC1363	ERC1363	142	31	88	33	22.59
ICO	gv_option_token	98	26	86	36	10.03
	gv_token	121	24	107	49	15.21
	gv_option_program*	86	12	154	67	29.19
	ico_alloc*	159	30	261	116	101.86
Mana	token*	18	3	50	25	2.78
	crowdsale*	42	14	50	25	5.77
	continuous_sale*	36	8	50	25	3.88
VerX_overview	escrow	60	11	65	33	6.36
	crowdsale	41	9	65	33	6.35
safe_remote_purchase	safe_remote_purchase	71	29	-	-	16.84
serenuscoin	serenuscoin	103	4	-	-	6.40
Uniswap	Uniswap†	398	115	105	45	112.81

Table 1. Evaluated examples. LOC are total lines of code, *including* specification, excluding comments and whitespace. Ann. are lines of specification. IF LOC and IF Ann. have the same meaning for the interfaces that were required to verify the contract and T is verification time in seconds. Contracts marked with a star are simplified versions of the original; applications marked with one or two daggers collaborate with an external ERC20 or ERC1363 contract, respectively (accessed through an interface).

Table 1 shows the examples; while many consist of a single contract, several either consist of multiple collaborating contracts or consist of a single contract interacting with external contracts via interfaces. We include several examples of complex code used in practice, e.g. several ERC tokens, the Uniswap contract (the largest decentralized exchange and fourth-largest cryptocurrency exchange overall), and an application used to implement the Genesis Vision ICO, which raised 2.8 million USD in 2017. Most contracts were verified in their entirety; for the ICO, we made some slight simplifications (in particular, we cut out two option tokens that were used and behaved exactly like a third one and so added nothing of interest); for the Mana and DAO contracts, we focused on specific parts demonstrating inter-contract invariants and a re-entrancy bug, respectively.

7.2 Verified Properties

Here, we describe for some of the examples which properties we verified and which specification constructs we used to express them; descriptions of the remaining examples can be found in App. D.

ERC20, auction and auction_token: We have verified an extended version of the auction contract from Fig. 1 and proved all properties mentioned throughout this paper. We have also verified the widely-used standard Vyper ERC20 implementation, which is a more complex version of the token contract in Fig. 2, by declaring a token resource and annotating all functions with the resource operations they perform. We also used segment constraints to specify when the contract triggers *events*, which are a means for the contract to log which transactions have happened in a way that is visible outside the blockchain, and which can easily be specified using segment constraints.

Finally, we have verified a variant of the auction that deals in custom tokens instead of wei against an ERC1363 interface [Minacori 2020] (see below) annotated with resource-based specifications.

DAO:. We extracted the buggy part of the DAO contract that led to the loss of ca. 50 million USD [Güçlütürk 2018]. Our implementation declares a derived resource for Ether by default (i.e. it assumes that Ether sent to a contract should still conceptually belong to the sender unless otherwise specified). As a result, when the contract tries to send Ether to an address, an error is reported by default, since our resource model requires the user to justify this action by showing that Ether is only sent to its rightful owner. Since this is not always the case, the contract will be rejected.

ICO:. We verified four contracts that implement the Initial Coin Offering (ICO) for Genesis Vision. The ICO progresses in stages, first selling options, then starting the ICO for option holders, and subsequently for the public. Verification required all specification constructs we have presented, e.g. function constraints to describe guarantees made by locks, transitive segment constraints to preserve information across calls (e.g. that the main token, once unfrozen, will never be frozen again), and resource specifications modeling the option token and main token. We used trust to allow that an administrator can freely access other's tokens, which our technique normally rules out. Importantly, we required proving multiple inter-contract invariants to coordinate the four contracts that implement the ICO, e.g. to prove that the main token will be frozen in its first stages.

Some (inter-contract) properties of this example have also been verified in VerX [Permenev et al. 2020]. Notably however, VerX requires the code of all involved contracts at once and does not allow using interface abstractions. In contrast, we use interfaces annotated with specifications to verify each contract modularly. Additionally, while we prove every property proved by VerX, we also proved additional properties (e.g. all standard resource properties such as non-duplicability and ownership, and that the resource operations the contract performs are the expected ones).

Uniswap: Uniswap is a popular application that consists of many different exchanges, which together allow clients to exchange different tokens against each other, using Ether as an intermediary. A single exchange is responsible for a single token and, if it wants to buy other tokens, contacts the respective exchange contracts for those other tokens. We declared the desired resource-effects for each function and proved the exchange contract correct w.r.t. them. Again, we did so modularly, using a standard ERC20 interface for its token contract and another interface for other exchanges.

VerX overview: We verified the crowdsale application (consisting of two contracts) from the VerX paper, which again included an inter-contract invariant that we verified *modularly* using interfaces and privacy constraints. Additionally, since one of the involved contracts implements a state machine, we used transitive segment constraints to define valid transitions between states (e.g. once the contract is in the "refund" state, it remains in this state).

ERC1363: ERC1363 is a new token standard [Minacori 2020] that combines into one transaction what ERC20 does in several, using re-entrancy. Fig. 7 shows an excerpt of a function that intentionally uses re-entrancy in a way that is not ECF and thus cannot be verified using approaches such as VerX: clients call approveAndCall to allow recipient to take some of its wei. The contract calls onApprovalReceived on the recipient, which calls the token contract back to take the tokens.

Conclusion: Our evaluation shows that our specification constructs allow specifying and verifying a wide variety of different properties for real-world contracts. In particular, we can modularly prove inter-contract properties, we can model the resources and resource transactions of different, complex contracts using our resource system (and find typical errors by default), and we can give guarantees for functional correctness and access control even in the presence of unbounded re-entrancy, which allows us to support contracts that employ re-entrancy by design.

7.3 Performance and Annotation Overhead

Table 1 shows the total lines of code of each contract (excluding comments and whitespace, including specification) as well as the lines of annotations we require, and the lines of code and specifications of all interfaces required to verify each contract, as well as the verification time required by 2Vyper. Times were measured by averaging over ten runs, running on a warmed-up JVM.

On our test system (a 12-core Ryzen 3900X with 32GB RAM running Ubuntu 20.04), most contracts can be automatically verified in 5-25 seconds; the two contracts with the longest verification time, both of which are from complex real-world applications, take between 1.5 and 2 minutes. Considering the strong guarantees afforded by our methodology and tool, we believe even the longest of these times is quite acceptable in practice.

The number of lines required for specifications is less than the number of lines of actual code for every contract. This comparatively modest specification overhead is partly due to our domain-specific resource specifications that allow users to express complex properties in a concise way, and partly due to the design of Vyper, which simplifies verification. Overall, considering the potential financial losses resulting from incorrect smart contracts, writing this amount of specification in exchange for strong functional correctness guarantees is clearly worthwhile.

In conclusion, our technique enables concisely specifying complex correctness properties of (collaborating) contracts, while allowing for modular verification that can be automated efficiently.

8 RELATED WORK

A lot of recent work has focused both on finding problems in smart contracts and on proving their absence. Atzei et al. [Atzei et al. 2017] and Luu et al. [Luu et al. 2016] each list different kinds of potential attacks and problems specific to smart contracts. A number of tools have been built to automatically find such problems (e.g. resulting from re-entrancy) using either syntactic patterns [Lai and Luo 2020; Tikhomirov et al. 2018; Tsankov et al. 2018], bounded symbolic execution [Alt and Reitwießner 2018; Luu et al. 2016; Mossberg et al. 2019; Nikolic et al. 2018] or data flow analyses [Feist et al. 2019]. However, most of these tools are not designed to be sound and can therefore miss errors in real contracts [Feist et al. 2019; Tikhomirov et al. 2018; Tsankov et al. 2018]. Additionally, none of these tools allow proving custom functional properties.

Recent work has studied the difference between harmless re-entrant executions and re-entrancy vulnerabilities [Cao and Wang 2020]. In particular, Grossman et al. [2018] have introduced the notion of effectively callback free objects, for which (in a smart contract setting) re-entrancy does not introduce any behaviours that are not present in executions without re-entrancy. They provide an algorithm for dynamically checking for ECF-violations and study the decidability of statically proving that a contract is ECF. More recently, Albert et al. [Albert et al. 2020] show a static analysis for deciding ECF based on commutativity and projection.

A number of tools aim to achieve verification of custom functional properties for Ethereum contracts, either on the level of the Solidity language [Hajdu and Jovanovic 2019; Kalra et al. 2018] or on the level of EVM bytecode [Hildenbrandt et al. 2018] - to the best of our knowledge, 2Vyper is the first verifier specifically aimed at the Vyper language. EVM-based verification is not specific to any high-level source language and does not rely on the correctness of the compiler; however, specifications tend to become much more complex on the EVM-level, where high-level abstractions of the source language are lost. Verification tools are either based on SMT solvers [Hajdu and Jovanovic 2019; Permenev et al. 2020], model checking with code generation [Mavridou et al. 2019], matching logic [Hildenbrandt et al. 2018], CHC solving [Kalra et al. 2018], or interactive theorem provers [Hirai 2017], which offer different levels of automation and expressiveness. Existing verification tools that offer dedicated, higher level specification languages (e.g. Hajdu and Jovanovic

[2019]) typically support single-state contract invariants, but offer no special support for reasoning in the presence of arbitrary re-entrancy beyond that, resulting in imprecision. VerX [Permenev et al. 2020] and VeriSolid [Mavridou et al. 2019] can express temporal properties, which subsume ordinary history constraints; however, VerX explicitly only targets contracts that are ECF, and VeriSolid prevents all re-entrancy by generating code that uses locks throughout. Addionally, none of the existing Ethereum verifiers support resource-based specifications.

To the best of our knowledge, the only tools capable of proving user-defined inter-contract properties are VerX and VeriSolid. The former requires the source code of all involved contracts and is therefore not contract-modular, unlike our approach. The latter uses model checking to prove temporal logic properties on a higher-level model of the contracts and their allowed interactions, and generates Solidity code from this model which is correct-by-design [Nelaturu et al. 2020]. In contrast to our approach, VeriSolid does not allow reasoning directly about the code of an existing contract (though contracts can be imported into the model).

Researchers have proposed a number of new smart contract languages that aim to simplify verification and/or make it more difficult to write incorrect code [Blackshear et al. 2019; Coblenz 2017; Sergey et al. 2019]. In particular, the Move language [Blackshear et al. 2019] offers resources on the programming language level. Unlike our resource model, these resources are stateful (in fact, *all* state in Move is stored in resources) and do not have a one-to-one correspondence to physical goods or currency: Receiving n coins, for example, is implemented in Move by adding n to the value stored in one's existing single coin resource, since every address can have at most one resource of every kind. While a linear type system ensures that resources are not duplicated in third party code, the module that defines a resource may modify resource state in arbitrary ways. As a result, incorrect module implementations in Move can potentially violate the properties guaranteed by our resource system (e.g. that resources cannot be taken away from their owners); on the other hand, Move's system allows users to manually implement more complex resource models than ours. Finally, an SMT-based verifier for custom properties of Move programs exists [Zhong et al. 2020] but currently does not offer special support for specifying resource transfers.

Drossopoulou et al. [2020] have recently introduced *holistic* specifications, which (unlike traditional specifications) express *necessary* conditions for an effect to happen, in a setting that allows for arbitrary re-entrancy. Their specifications can express e.g. that if a user's balance in a token contract decreases, then either they must have asked to transfer tokens themselves, or another user with a sufficient allowance must have done so. While this kind of property is similar to ones ensured by our resource system, it is not built-in and must be specified manually. Additionally, holistic specifications do not provide support for reasoning about the post-state of calls with arbitrary re-entrancy, and the required (non-standard) reasoning has not been automated, whereas the proof obligations generated by our approach can be checked and automated using standard techniques.

9 CONCLUSION

In this paper, we have presented a novel approach for specification and verification of Ethereum smart contracts. Our methodology exploits the features of Ethereum, such as strong encapsulation, to provide guarantees even in the presence of arbitrary re-entrancy, and provides domain-specific specification constructs for resources that make specification both more intuitive and less error-prone. Our evaluation shows that out methology can be implemented efficiently and is capable of expressing and proving complex functional specifications for real-world contracts.

REFERENCES

Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020. Taming callbacks for smart contract modularity. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 209:1–209:30. https://doi.org/10.

1145/3428277

Leonardo Alt and Christian Reitwießner. 2018. SMT-Based Verification of Solidity Smart Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 11247)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 376–388. https://doi.org/10.1007/978-3-030-03427-6_28

Sivakumar Arumugam. 2019. Serenuscoin contract. https://github.com/serenuscoin/contracts Accessed on 2021-04-16.

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10204)*, Matteo Maffei and Mark Ryan (Eds.). Springer, 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. 2004. Verification of Object-Oriented Programs with Invariants. *J. Object Technol.* 3, 6 (2004), 27–56. https://doi.org/10.5381/jot.2004.3.6.a2

Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. 2019. Move: A language with programmable resources. https://developers.libra.org/docs/move-paper

Blockchains LLC. 2016. Decentralized Autonomous Organization (DAO) Framework. https://github.com/blockchainsllc/DAO/blob/6967d70e0e11762c1c34830d7ef2b86e62ff868e/DAO.sol Accessed on 2021-04-16.

Qinxiang Cao and Zhongye Wang. 2020. Reentrancy? Yes. Reentrancy Bug? No. In *Dependable Software Engineering. Theories, Tools, and Applications - 6th International Symposium, SETTA 2020, Guangzhou, China, November 24-27, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12153)*, Jun Pang and Lijun Zhang (Eds.). Springer, 17–34. https://doi.org/10.1007/978-3-030-62822-2_2

Michael J. Coblenz. 2017. Obsidian: a safer blockchain programming language. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 97–99. https://doi.org/10.1109/ICSE-C.2017.150

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. 2008. A Unified Framework for Verification Techniques for Object Invariants. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5142)*, Jan Vitek (Ed.). Springer, 412–437. https://doi.org/10.1007/978-3-540-70592-5_18

Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. 2020. Holistic Specifications for Robust Programs. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12076)*, Heike Wehrheim and Jordi Cabot (Eds.). Springer, 420–440. https://doi.org/10.1007/978-3-030-45234-6_21

William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. 2018. EIP-721: ERC-721 Non-Fungible Token Standard. *Ethereum Improvement Proposals* 721 (2018). https://eips.ethereum.org/EIPS/eip-721

Ethereum. 2021a. Solidity by example. https://github.com/ethereum/solidity Accessed on 2021-04-16.

Ethereum. 2021b. Solidity documentation. https://solidity.readthedocs.io/ Accessed on 2020-01-11.

Ethereum. 2021c. Vyper documentation. https://vyper.readthedocs.io/ Accessed on 2020-01-11.

Ethereum. 2021d. Vyper example contracts. https://github.com/vyperlang/vyper/tree/master/examples Accessed on 2021-04-16.

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

Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL* 2, POPL (2018), 48:1–48:28. https://doi.org/10.1145/3158136

Osman Gazi Güçlütürk. 2018. The DAO Hack Explained: Unfortunate Take-off of Smart Contracts. https://medium.com/@ogucluturk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562 Accessed on 2021-03-31.

Ákos Hajdu and Dejan Jovanovic. 2019. solc-verify: A Modular Verifier for Solidity Smart Contracts. *CoRR* abs/1907.04262 (2019). arXiv:1907.04262 http://arxiv.org/abs/1907.04262

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

Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10323)*, Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (Eds.). Springer, 520–535. https://doi.org/10.1007/978-3-319-70278-0_33

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

Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4085)*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer, 268–283. https://doi.org/10.1007/11813040_19

Enmei Lai and Wenjun Luo. 2020. Static Analysis of Integer Overflow of Smart Contracts in Ethereum. In *ICCSP 2020: 4th International Conference on Cryptography, Security and Privacy, Nanjing, China, January 10-12, 2020*. ACM, 110–115. https://doi.org/10.1145/3377644.3377650

Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. 2008. JML reference manual.

K. Rustan M. Leino and Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3086)*, Martin Odersky (Ed.). Springer, 491–516. https://doi.org/10.1007/978-3-540-24851-4_22

Barbara Liskov and Jeannette M. Wing. 1993. Specifications and Their Use in Defining Subtypes. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Eighth Annual Conference, Washington, DC, USA, September 26 - October 1, 1993, Proceedings*, Timlynn Babitsky and Jim Salmons (Eds.). ACM, 16–28. https://doi.org/10.1145/165854.165863

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

Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. 2019. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11598)*, Ian Goldberg and Tyler Moore (Eds.). Springer, 446–465. https://doi.org/10.1007/978-3-030-32101-7_27

Vittorio Minacori. 2020. EIP-1363: ERC-1363 Payable Token. *Ethereum Improvement Proposals* 1363 (2020). https://eips.ethereum.org/EIPS/eip-1363

Vittorio Minacori. 2021. ERC-1363 Payable Token. https://github.com/vittominacori/erc1363-payable-token Accessed on 2021-04-16.

Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. *CoRR* abs/1907.03890 (2019). arXiv:1907.03890 http://arxiv.org/abs/1907.03890

Peter Müller. 2002. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science, Vol. 2262. Springer. https://doi.org/10.1007/3-540-45651-1

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Keerthi Nelaturu, Anastasia Mavridou, Andreas G. Veneris, and Aron Laszka. 2020. Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2020, Toronto, ON, Canada, May 2-6, 2020*. IEEE, 1–9. https://doi.org/10.1109/ICBC48266.2020.9169428

Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 653–663. https://doi.org/10.1145/3274694.3274743

Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2019. VerX smart contract verification benchmarks. https://github.com/eth-sri/verx-benchmarks Accessed on 2021-04-16.

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

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. *PACMPL* 3, OOPSLA (2019), 185:1–185:30. https://doi.org/10.1145/3360611

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

Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 67–82. https://doi.org/10.1145/3243734.3243780

Uniswap. 2019. Uniswap version 1. https://github.com/Uniswap/uniswap-v1 Accessed on 2021-04-16.

Fabian Vogelsteller and Vitalik Buterin. 2015. EIP-20: ERC-20 Token Standard. *Ethereum Improvement Proposals* 20 (2015). https://eips.ethereum.org/EIPS/eip-20

Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark W. Barrett, and David L. Dill. 2020. The Move Prover. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 137–150. https://doi.org/10.1007/978-3-030-53288-8_7

A RESOURCE GHOST COMMANDS AND EFFECTS

In this section, we give a more detailed description of the ghost commands that can be used to modify the resource state, and the effects they cause.

Creating and Destroying Resources. Initially, every address has a balance of zero for every resource. Resource can be created by executing the ghost command $\mathsf{create}_R(e_c, e_t, e_a)$, which means that e_c creates e_a amount of resource R and allocates it to e_t, and which has the effect $\mathsf{create}_R(e_t, e_a)$. Since this command states that e_c is the one creating the resource, this command may be performed only if e_c is the current msg.sender, i.e. the caller of the function that contains this ghost command (modulo trust). Creating a resource additionally requires owning the *right* to do so, which we represent by a special resource $\mathsf{creator}(R)$. In the constructor of the contract, the caller of the constructor automatically gets the right to create such creator resources (i.e. they get a resource $\mathsf{creator}(\mathsf{creator}(R))$ for every resource type R declared by the contract) and can then set up the contract so that the parties that are intended to have minting rights each own a creator resource at the end of the constructor.

In function mint of the token contract, which increases the number of tokens in the contract state, we would have to insert the command $\mathsf{create}_R(\mathsf{self.minter}, \mathsf{to}, \mathsf{amount})$ to preserve the relation between $\mathsf{balances}_{token}$ and the contract state. Verifying mint then requires showing that the caller is self.minter, which we know because of the assertion in the first line, and that self.minter owns a creator resource for token. To show the latter, we must create such a resource in the contract's constructor (not shown in Fig. 2), and record the fact that the minter owns it in an additional invariant.

Conversely, when executing the command $\mathsf{destroy}_R(e_f, e_a)$, e_f destroys e_a amount of its reserves of resource R, which requires that e_f actually has that amount of resource, and, as before, that e_f is (or trusts) the msg.sender. This ghost command has the effect $\mathsf{destroy}_R(e_f, e_a)$.

Transfers. The ghost command $\mathsf{transfer}_R(e_f, e_t, e_a)$ transfers e_a amount of resource R from e_f to e_t, and requires that e_f owns that amount and is (or trusts) the msg.sender. This last requirement, along with the similar requirement for destroying resources, is what guarantees that once a contract owns a resource, no other contract can take it away or destroy it. The command has the effect $\mathsf{transfer}_R(e_f, e_t, e_a)$.

In the transfer function of the token example, we would have to insert the ghost command $\mathsf{transfer}_{token}(\textbf{from}, \mathsf{to}, \mathsf{amount})$ before the call to notify in order to re-establish the invariant. Verifying the function requires proving that **from** is the msg.sender (modulo trust) and that **from** owns at least amount tokens, both of which we can prove because of the assertion in the first line of the function.

Offers and Exchanges. The resource transfers just shown can only happen directly at the command of the sender, at the moment the sender requests them. In practice, this is not always sufficient to model different kinds of resources. Therefore, as we explain in the main body of the paper, we allow two exceptions from this general requirement; the first is that contracts can *offer* to perform specific resource *exchanges* with other contracts at some later point in time. If both parties have offered an exchange, the exchange can then be performed at any point, without requiring further agreement from the involved parties.

The command $\mathsf{offer}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, e_n)$ creates e_n offers from e_f to contract e_t to exchange e_{a_1} amount of R_1 against e_{a_2} amount of R_2. The command $\mathsf{revoke}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, e_n)$ revokes e_n offers from e_f to contract e_t to exchange e_{a_1} amount of R_1 against e_{a_2} amount of R_2. Both commands can only be executed by e_f or someone trusted by e_f. However, they do *not* require that e_f actually *owns* these amounts of the specified resources: contracts can offer exchanges that they could not actually perform at the time of making the offer, which can then potentially be performed at a later point once they have the necessary resources. Like the previous ghost commands, both of these ghost commands have corresponding effects with the same form.

The command $\mathsf{exchange}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2})$ performs an exchange between e_{a_1} amount of R_1 from e_f and e_{a_2} amount of R_2 from e_t. Executing it requires that both e_f and e_t have made an offer to perform such an exchange, and consumes the offer. There is one exception to this: if either e_{a_1} or e_{a_2} are zero, i.e. the exchange simply gives a resource to one party without requiring anything in return, no offer is required from the party receiving the resource.

Unlike any other ghost command, there are no requirements as to who can execute this command (i.e. who the msg.sender is), since its effect is one that all affected parties have already agreed to previously. The effect $\mathsf{exchange}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2})n$ states that n such exchanges have happened.

Note that, once a contract makes an offer to exchange some amount of a resource, it is no longer guaranteed that no other contract can take that resource away from it; instead, the resulting guarantee is that if some other contract takes resources away, this happens only in the form of an exchange the initial contract has previously agreed to.

Trust and Delegation. The second exception that allows resources to be managed by someone that is not the owner is, as we briefly hint at in the main body of the paper, that a contract can *trust* another contract to act (e.g. to transfer resources or make offers) in its place. One common use case for this is that multiple contracts collaborate and represent the same party, and need to be able to act in each other's name. Some token contract standards, e.g. ERC721 [Entriken et al. 2018], have a built-in mechanism to enable exactly this pattern.

$$
\begin{array}{rcl}
\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models \mathsf{emp} & \Leftrightarrow & \mathcal{R} = \mathcal{R}_{empty} \land \mathcal{E} = \emptyset^{\#} \\
\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models e & \Leftrightarrow & [\![e]\!]_{(\sigma, \mathcal{H}, \mathcal{R}, \mathcal{O})} \land \mathcal{E} = \emptyset^{\#} \\
\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models P * Q & \Leftrightarrow & \exists \mathcal{R}_1, \mathcal{R}_2, \mathcal{E}_1, \mathcal{E}_2. \mathcal{R} = \mathcal{R}_1 \uplus \mathcal{R}_2 \land \mathcal{E} = \mathcal{E}_1 \cup^{\#} \mathcal{E}_2 \land \\
 & & \langle \mathcal{H}, \mathcal{R}_1, \mathcal{E}_1, \mathcal{O}, \sigma \rangle \models P \land \langle \mathcal{H}, \mathcal{R}_2, \mathcal{E}_2, \mathcal{O}, \sigma \rangle \models Q \\
\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models P \land Q & \Leftrightarrow & \langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models P \land \langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models Q \\
\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models P \mathrel{-\!\!*} Q & \Leftrightarrow & \forall \mathcal{R}', \mathcal{E}'. \langle \mathcal{H}, \mathcal{R}', \mathcal{E}', \mathcal{O}, \sigma \rangle \models P \Rightarrow \\
 & & \langle \mathcal{H}, \mathcal{R} \uplus \mathcal{R}', \mathcal{E} \cup^{\#} \mathcal{E}', \mathcal{O}, \sigma \rangle \models Q \\
\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models P \lor Q & \Leftrightarrow & \langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models P \lor \langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models Q \\
\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models \mathsf{perf}(E) & \Leftrightarrow & \mathcal{R} = \mathcal{R}_{empty} \land \mathcal{E} = E \\
\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models \mathsf{owns}_R(e_o, e_a) & \Leftrightarrow & \mathcal{R} = \mathcal{R}_{empty}[\mathsf{balances}_R := b] \land \mathcal{E} = \emptyset^{\#} \\
\text{where} & & b = [o \mapsto a, \_ \mapsto 0], \\
 & & o = [\![e_o]\!]_{(\sigma, \mathcal{H}, \mathcal{R}, \mathcal{O})}, a = [\![e_a]\!]_{(\sigma, \mathcal{H}, \mathcal{R}, \mathcal{O})} \\
\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models \mathsf{offers}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, e_n) & \Leftrightarrow & \mathcal{R} = \mathcal{R}_{empty}[\mathsf{offered}_{R_1 \leftrightarrow R_2} := o] \land \mathcal{E} = \emptyset^{\#} \\
\text{where} & & o = [(f, t, a_1, a_2) \mapsto n, \_ \mapsto 0], \\
 & & f = [\![e_f]\!]_{(\sigma, \mathcal{H}, \mathcal{R}, \mathcal{O})}, t = [\![e_t]\!]_{(\sigma, \mathcal{H}, \mathcal{R}, \mathcal{O})} \\
 & & a_1 = [\![e_{a_1}]\!]_{(\sigma, \mathcal{H}, \mathcal{R}, \mathcal{O})}, a_2 = [\![e_{a_2}]\!]_{(\sigma, \mathcal{H}, \mathcal{R}, \mathcal{O})}, \\
 & & n = [\![e_n]\!]_{(\sigma, \mathcal{H}, \mathcal{R}, \mathcal{O})} \\
\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models \mathsf{trusts}(e_{c_1}, e_{c_2}, e_v) & \Leftrightarrow & \mathcal{R} = \mathcal{R}_{empty}[\mathsf{trusted} := t] \land \mathcal{E} = \emptyset^{\#} \\
\text{where} & & t = [(c_1, c_2) \mapsto v], v = [\![e_v]\!]_{(\sigma, \mathcal{H}, \mathcal{R}, \mathcal{O})}, \\
 & & c_1 = [\![e_{c_1}]\!]_{(\sigma, \mathcal{H}, \mathcal{R}, \mathcal{O})}, c_2 = [\![e_{c_2}]\!]_{(\sigma, \mathcal{H}, \mathcal{R}, \mathcal{O})}
\end{array}
$$

Fig. 8. Definition of assertion truth in a state. If r is a record, $r[f := v]$ updates field f of the record to value v. Operator $\uplus$ for resource states is defined s.t. it performs pointwise addition for balance and offer maps, and combination of partial functions with disjoint domains for the trusted map. $\mathcal{R}_{empty}$ denotes an empty resource state (i.e. all balances are zero, there are no offers, and the domain of the partial trust map is empty). We write $\emptyset^{\#}$ for the empty multiset and $\cup^{\#}$ for multiset union.

Trust is *local*, i.e. A can trust B to act in its name *only* when interacting with contract C. Like for offers and balances, every contract has a ghost map trusted that represents which contracts trust which other contracts when interacting with the current contract. For all commands we have previously discussed, the requirement is therefore not actually that the msg.sender *is* the respective source address/creator/..., but that the msg.sender is *trusted* by the source address/... (since every contract implicitly trusts itself).

Trust can be modified via the ghost command $\mathsf{trust}(e_c, e_v)$, which sets the current caller's trust to the contract at address e_c (when interacting the current contract) to the boolean value e_v. The effect $\mathsf{trust}_{e_{ctr}}(e_{c_1}, e_{c_2}, e_v)$ states that e_{c_1} has set its trust to the contract at address e_{c_2} when interacting with contract e_{ctr} to e_v; as a result, the ghost command $\mathsf{trust}(e_c, e_v)$ causes the effect $\mathsf{trust}_{\mathsf{self}}(\mathsf{msg.sender}, e_c, e_v)$.

Note that the ability to extend trust cannot be delegated, i.e. $\mathsf{trust}(e_c, e_v)$ always modifies the trust of the msg.sender, not of anyone else.

B DEFINITIONS AND PROOF RULES

Assertion truth in a state is defined by a judgement $\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle \models P$ whose cases are given in Fig. 8. As commented in Sec. 6, in contrast to traditional separation logics [Reynolds 2002], we do not use the *linear/separating* aspects of the $*$ and $\mathrel{-\!\!*}$ connectives to govern access to the (already-encapsulated) *heap*, but rather for the resource state and effects concepts added by our methodology. The separating conjunction $P * Q$ splits the resource state and the effects into two parts; the first described by P and the second by Q. Descriptions of constituent parts of the resource state come via assertions such as $\mathsf{owns}_R(e_o, e_a)$ that prescribe that $\mathcal{R}$ is empty (no offers, no trust,

and all balances are zero) *except* for the balance of e_o, which owns exactly e_a of resource R, and that $\mathcal{E}$ is empty (no effects). The (multiplicative) separating conjunction builds up larger descriptions of these states; e.g. $\mathsf{owns}_R(e_o, e_{a_1}) * \mathsf{owns}_R(e_o, e_{a_2})$ is equivalent to $\mathsf{owns}_R(e_o, e_{a_1} + e_{a_2})$. The assertions $\mathsf{offers}_{R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, e_n)$ and $\mathsf{trusts}(e_{c_1}, e_{c_2}, e_v)$ form the analogous base cases prescribing offers and trust. Note that $\mathsf{owns}_R(e_o, e_a)$ can be used alongside assertions containing references to the $\mathsf{balances}_R$ map; both are useful in different contexts. For example, an invariant that states that the allocation of *token* in the resource state is stored in the contract in the field self.tokens can be easily expressed as $\mathsf{balances}_{token} = \mathsf{self.tokens}$, whereas the proof rules for framing and resource commands, which we will show later, are much easier to express using the exact assertion $\mathsf{owns}_R(e_o, e_a)$.

We write CS_a to refer to the entire state, including the resource state, of the contract at address a. We denote (the conjunction of) the primary contract's transitive segment constraints (which may refer to other contracts' states) by TSC, its segment constraints by SC, and its function constraints by FC. The latter two may only refer to the primary contract's state. By ITSC, we denote the conjunction of the transitive segment constraints of all known interfaces. By $\mathsf{PC}(e_1, e_2)$, we denote the conjunction of the reflexive and transitive assertions P from the privacy constraints of all known interfaces *except* those in set e_2, instantiated for e_1. In all those specification constructs, the old state is referred to without a label (simply as $\mathbf{old}(e)$), since the kind of specification construct determines which old state it refers to.

The proof rules are shown in Fig. 9 and Fig. 10. $FV(P)$ are the free variables in P, $mods(c)$ are the variables modified by c. We write $e[e_1/e_2]$ to substitute all occurrences of e_2 in e by e_1.

The rules for ordinary statements are standard; the bulk of the work happens in the rule for calls, as well as in the rules for resource commands. We will explain the checks for every element of our methodology step by step.

Ordinary and transitive segment constraints must hold at the end of every local segment. We therefore require them to hold at the end of each function as well as before every call (in a state where the Ether that is about to be sent with the call has already been deducted), i.e. in the premise of the call rule (we will discuss the resource assertions here later), w.r.t. the old state at the beginning of the local segment (i.e. the old state labelled "last"). Crucially, what this last state is does not stay constant throughout a function, but it changes every time a new local segment starts, i.e. after each call. To model the fact that after a call, the new "last" state is the current state, the postcondition of the call rule allows assuming that any expression e_N that is true in the current state is also true in the last state. Similarly, the rule for functions allows doing this at the beginning of each function.

The frame rule ensures that no information about the last state or the current state can be framed around calls; this represents the fact that the entire contract state can change with every call. After a call, one may assume the transitive segment constraints w.r.t. to the call's pre-state. To remember information about said pre-state, we use the same trick as before, and allow assuming any expression e_O after a call about its pre-state that was known to be true before the call. In constructors, no calls are allowed, and we check at the end that transitive segment constraints hold in the current state w.r.t. itself, which ensures that all single-state invariants contained in the transitive segment constraints are established.

The proof obligations for function constraints work in a similar way: They must be shown to hold at the end of each function w.r.t. to its pre-state (which may again be remembered by assuming that everything that holds in the beginning of the function holds in its pre-state), and may be assumed after a call w.r.t. to the call's pre-state.

The rule for constructors ensures that all transitive segment constraints and function constraints are, in fact, reflexive and transitive, and that the resource state is precisely determined by the transitive segment constraints (i.e. they precisely determine the resource state in terms of the

$$\frac{}{\vdash \{\mathsf{true}\}\ \mathbf{assert}\ e\ \{e\}}\ (\mathit{Assert}) \qquad \frac{}{\vdash \{Q[e/x]\}\ x := e\ \{Q\}}\ (\mathit{Assign})$$

$$\frac{}{\vdash \left\{Q[(e' = e_1 ? e_2 : e'.f)/e'.f]\right\}\ e_1.f := e_2\ \{Q\}}\ (\mathit{Write})$$

$$\frac{\begin{array}{c}\mathsf{TSC}' = \mathsf{TSC}[\mathbf{old}_{last}(\_)/\mathbf{old}(\_)] \\ [(e' = \ \mathsf{self}\,?(\ \mathsf{self}\,.\mathsf{balance} - e_a) : e'.\mathsf{balance})/e'.\mathsf{balance}] \\ \mathsf{SC}' = \mathsf{SC}[\mathbf{old}_{last}(\_)/\mathbf{old}(\_)][(e' = \ \mathsf{self}\,?(\ \mathsf{self}\,.\mathsf{balance} - e_a) : e'.\mathsf{balance})/e'.\mathsf{balance}] \\ e_r : T \quad T.\mathsf{fun}(x)\ \mathsf{ensures}\ Q\ \mathsf{performs}\ S \quad S' \subseteq S'' \\ S'' = S[e_r/\ \mathsf{self}\,][x/\ \mathsf{result}\,][\ \mathsf{self}\,/\mathsf{msg}.\mathit{sender}][e_a/x] \\ \mathit{secondary}(e_r) \Rightarrow\ \mathsf{stable}\,(\mathsf{TSC} \wedge e_S, \mathsf{ITSC} \wedge \mathsf{PC}(\ \mathsf{self}\,, \{e_r\}) \wedge \mathsf{CS}_{\mathsf{self}}\ = \mathbf{old}(\mathsf{CS}_{\mathsf{self}})) \\ \mathit{secondary}(e_r) \Rightarrow\ \mathsf{stable}\,(\mathsf{TSC} \wedge e_S, \mathsf{ITSC} \wedge \mathsf{FC})\end{array}}{\vdash \left\{\begin{array}{c}\mathsf{derDestroyed}(S'')* \\ (\mathsf{derCreated}(S'') \mathrel{-\!\!*} \\ (\mathsf{TSC}' \wedge \mathsf{SC}' \\ \wedge e_O \wedge e_S))\end{array}\right\}\ x := e_r.\mathsf{fun}\ (e_a, \mathsf{value} = e_v)\ \left\{\begin{array}{c}\mathsf{perf}(\mathsf{derPerformed}(S''))* \\ \mathsf{perf}(S') * (\mathbf{old}_{call}(e_O) \wedge \\ \mathsf{TSC}[\mathbf{old}_{call}(\_)/\mathbf{old}(\_)] \wedge \\ \mathsf{FC}[\mathbf{old}_{call}(\_)/\mathbf{old}(\_)] \wedge \\ Q[e_r/\ \mathsf{self}\,][x/\ \mathsf{result}\,] \\ [\ \mathsf{self}\,/\mathsf{msg}.\mathit{sender}][e_a/x] \\ [\mathbf{old}_{call}(\_)/\mathbf{old}(\_)] \wedge \\ e_N \Rightarrow \mathbf{old}_{last}(e_N))\end{array}\right\}}\ (\mathit{Call})$$

$$\frac{\vdash \{P\}\ c_1\ \{R\} \qquad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1; c_2\ \{Q\}}\ (\mathit{Seq}) \qquad \frac{\vdash \left\{P'\right\}\ c\ \left\{Q'\right\} \qquad P \models P' \qquad Q' \models Q}{\vdash \{P\}\ c\ \{Q\}}\ (\mathit{Cons})$$

$$\frac{\begin{array}{c}\mathit{FV}(R) \cap \mathit{mods}(c) = \emptyset \qquad \vdash \{P\}\ c\ \{Q\} \\ R \text{ is stateless if } c \text{ contains a call}\end{array}}{\vdash \{P * R\}\ c\ \{Q * R\}}\ (\mathit{Frame})$$

$$\frac{\vdash \left\{\begin{array}{c}\exists l.\, \mathsf{TSC}[\mathbf{old}_l(\_)/\mathbf{old}(\_)] \wedge \\ e_{N_1} \Rightarrow \mathbf{old}_{last}(e_{N_1}) \wedge \\ e_{N_2} \Rightarrow \mathbf{old}_{pre}(e_{N_2})\end{array}\right\}\ c\ \left\{\begin{array}{c}Q[\mathbf{old}_{pre}(\_)/\mathbf{old}(\_)] \wedge \\ \mathsf{FC}[\mathbf{old}_{pre}(\_)/\mathbf{old}(\_)] \\ \wedge \mathsf{TSC}[\mathbf{old}_{last}(\_)/\mathbf{old}(\_)] \\ \wedge \mathsf{SC}[\mathbf{old}_{last}(\_)/\mathbf{old}(\_)] \\ * \mathsf{perf}(S)\end{array}\right\}}{\mathbf{def}\ f(x) : T\ \{c\}\ \mathsf{ensures}\ Q\ \mathsf{performs}\ S}\ (\mathit{Func})$$

$$\frac{\begin{array}{c}\vdash \{\mathsf{CREATORS} * \ \mathsf{default}\,(\mathsf{CS}_{\mathsf{self}})\}\ c\ \left\{\begin{array}{c}\mathsf{TSC}[\_/\mathbf{old}(\_)] \wedge Q \\ * \mathsf{perf}(S)\end{array}\right\} \\ \mathsf{TSC} \text{ precisely determines resource state.} \\ \mathsf{TSC}, \mathsf{FC} \text{ are reflexive and transitive.} \\ c \text{ does not contain any calls.} \\ \mathsf{stable}\,(\mathsf{TSC}, \mathsf{ITSC} \wedge \mathsf{PC}(\ \mathsf{self}\,, \emptyset) \wedge \mathsf{CS}_{\mathsf{self}}\ = \mathbf{old}(\mathsf{CS}_{\mathsf{self}}))\end{array}}{\mathbf{def}\ \ \mathsf{init}\ (x)\ \{c\}\ \mathsf{ensures}\ Q\ \mathsf{performs}\ S}\ (\mathit{Init})$$

Fig. 9. Statement, function and constructor proof rules.

contract state). Similarly, the rule for the constructor ensures that transitive segment constraints are stable under the transitive segment constraints and privacy constraints of all known interfaces. For every call to a secondary contract, the call rule requires that the transitive segment constraints, potentially *strengthened* by conjoining them with some arbitrary information e_S that is known

$$\frac{R \neq \text{wei} \qquad R \text{ is not derived}}{\vdash \left\{ \begin{array}{c} (e_a \neq 0 \Rightarrow \\ \text{trusts}\,(e_c, \text{msg.sender}, \text{true})) * \\ \text{owns}_{\text{creator}\,(R)}(e_c, 1) \\ \wedge e_a \geq 0 \end{array} \right\} \text{create}_R(e_c, e_t, e_a) \left\{ \begin{array}{c} (e_a \neq 0 \Rightarrow \\ \text{trusts}\,(e_c, \text{msg.sender}, \text{true})) * \\ \text{owns}_R(e_t, e_a) * \\ \text{perf}(\,\text{create}_R(e_t, e_a)) * \\ \text{owns}_{\text{creator}\,(R)}(e_c, 1) \end{array} \right\}} \;(\textit{Create})$$

$$\frac{R \neq \text{wei} \qquad R \text{ is not derived}}{\vdash \left\{ \begin{array}{c} (e_a \neq 0 \Rightarrow \\ \text{trusts}\,(e_f, \text{msg.sender}, \text{true})) * \\ \text{owns}_R(e_f, e_a) \wedge e_a \geq 0 \end{array} \right\} \text{destroy}_R(e_f, e_a) \left\{ \begin{array}{c} (e_a \neq 0 \Rightarrow \\ \text{trusts}\,(e_f, \text{msg.sender}, \text{true})) * \\ \text{perf}(\,\text{destroy}_R(e_f, e_a)) \end{array} \right\}} \;(\textit{Destroy})$$

$$\frac{}{\vdash \left\{ \begin{array}{c} e_a \geq 0 * \\ (a \neq 0 \Rightarrow \\ \text{trusts}\,(e_f, \text{msg.sender}, \text{true})) * \\ \text{owns}_R(e_f, e_a) \wedge e_a \geq 0 \end{array} \right\} \text{transfer}_R(e_f, e_t, e_a) \left\{ \begin{array}{c} \text{owns}_R(e_t, e_a) * \\ (e_a \neq 0 \Rightarrow \\ \text{trusts}\,(e_f, \text{msg.sender}, \text{true})) \\ * \text{perf}(\,\text{transfer}_R(f, t, a)) \end{array} \right\}} \;(\textit{Transfer})$$

$$\frac{}{\vdash \left\{ \;\text{trusts}\,(\text{msg.sender}, e_c, \_)\; \right\} \text{trust}\,(e_c, e_v) \left\{ \begin{array}{c} \text{trusts}\,(\text{msg.sender}, e_c, e_v) \\ * \text{perf}(\,\text{trust}_{\,\text{self}}\,(\text{msg.sender}, e_c, e_v)) \end{array} \right\}} \;(\textit{Trust})$$

$$\frac{R_1 \neq \text{wei} \qquad \text{No resource derived from } R_1.}{\vdash \left\{ \begin{array}{c} e_{a_1} \geq 0 \wedge e_{a_2} \geq 0 \\ \wedge e_n \geq 0 \wedge \text{emp} * \\ (e_n \neq 0 \Rightarrow \\ \text{trusts}\,(e_f, \text{msg.sender}, \text{true})) \end{array} \right\} \begin{array}{c} \text{offer}_{\,R_1 \leftrightarrow R_2} \\ (e_f, e_t, e_{a_1}, e_{a_2}, e_n) \end{array} \left\{ \begin{array}{c} \text{offers}_{\,R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, e_n) \\ * \text{perf}(\,\text{offer}_{\,R_1 \leftrightarrow R_2} \\ (e_f, e_t, e_{a_1}, e_{a_2}, e_n)) \\ * (e_a \neq 0 \Rightarrow \\ \text{trusts}\,(e_f, \text{msg.sender}, \text{true})) \end{array} \right\}} \;(\textit{Offer})$$

$$\frac{}{\vdash \left\{ \begin{array}{c} e_{a_1} \geq 0 \wedge e_{a_2} \geq 0 \\ \wedge e_n \geq 0 \wedge \text{emp} * \\ (e_n \neq 0 \Rightarrow \\ \text{trusts}\,(e_f, \text{msg.sender}, \text{true})) \\ * \,\text{offers}_{\,R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, e_n) \end{array} \right\} \begin{array}{c} \text{revoke}_{R_1 \leftrightarrow R_2} \\ (e_f, e_t, e_{a_1}, e_{a_2}, e_n) \end{array} \left\{ \begin{array}{c} (e_a \neq 0 \Rightarrow \\ \text{trusts}\,(e_f, \text{msg.sender}, \text{true})) * \\ \text{perf}(\text{revoke}_{R_1 \leftrightarrow R_2} \\ (e_f, e_t, e_{a_1}, e_{a_2}, e_n)) \end{array} \right\}} \;(\textit{Revoke})$$

$$\frac{}{\vdash \left\{ \begin{array}{c} e_{a_1} \geq 0 * e_{a_2} \geq 0 \\ * (e_{a_1} > 0 \Rightarrow \\ \text{offers}_{\,R_1 \leftrightarrow R_2}(e_f, e_t, e_{a_1}, e_{a_2}, 1)) \\ * (e_{a_2} > 0 \Rightarrow \\ \text{offers}_{\,R_2 \leftrightarrow R_1}(e_t, e_f, e_{a_2}, e_{a_1}, 1)) \\ * \text{owns}_{R_1}(e_f, e_{a_1}) * \text{owns}_{R_2}(e_t, e_{a_2}) \end{array} \right\} \begin{array}{c} \text{exchange}_{R_1 \leftrightarrow R_2} \\ (e_f, e_t, e_{a_1}, e_{a_2}) \end{array} \left\{ \begin{array}{c} \text{owns}_{R_1}(e_t, e_{a_1}) * \\ \text{owns}_{R_2}(e_f, e_{a_2}) * \\ \text{perf}(\text{exchange}_{R_1 \leftrightarrow R_2} \\ (e_f, e_t, e_{a_1}, e_{a_2}, 1)) \end{array} \right\}} \;(\textit{Exchange})$$

Fig. 10. Rules for resource ghost commands.

about the current state (like the fact that self . lock is set in the example in Sec. 4), are stable under each of the following two assertions:

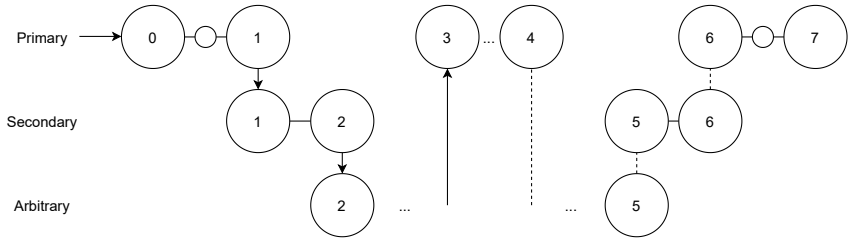

Fig. 11. Example showing a call from a primary to a secondary contract that leads to a re-entrant call to the primary contract.

(1) The conjunction of the transitive segment constraints of all interfaces, the knowledge that the primary contract's state does not change, and the privacy constraints of all secondary contracts *except* the called one (which is now trivial).
(2) The conjunction of the transitive segment constraints of all interfaces and the function constraints of the primary contract.

Fig. 11 illustrates the scenario where the primary contract calls a secondary contract. The strengthened invariant is known to hold in state 1. Whatever local changes the secondary contract performs cannot break the strengthened invariant, since it is stable under the first assertion (e.g. in the example, changes in the secondary contract cannot break the strengthened invariant that remains true as long as self . lock is set). Any contracts executing between states 2 and 5 cannot break the original non-strengthened invariant by the reasoning laid out in Sec. 4. Additionally, because the strengthened invariant is stable under the second assertion, it will also be re-established by the time any re-entrant calls the secondary contract may (transitively) make on the primary contract return (states 3 and 4); in the example, a function constraint guarantees that self . lock will again be set when such a re-entrant call returns. As a result, again because of stability under the first assertion, the secondary contract again cannot break the invariant after the return (between states 5 and 6) either.

The rules for resource commands are mostly intuitive. A transfer, for example, requires that the sender initially has the resources that are to be transferred, and ends in a state where the recipient has them. It also requires that the caller of the function is trusted by the address whose resources are being transferred (everybody always trusts themselves). Finally, the conclusion states that a transfer-effect has occurred. Rules for other resource commands are analogous; for example, the rule for exchanges requires two compatible offers (unless the offered amount of a party is zero) and consumes them, switches ownership of the involved resources, and records that an exchange-effect has occurred. The rule for resource creation requires that e_c (the party who is creating the new resource of type R) owns a creator-resource for R, which represents the right to create new resources. These creator resources can be created and given to arbitrary addresses in the contract's constructor: In the constructor rule, the caller of the constructor is given the right to create such creator-resources for any resource the contract declares. This is denoted by CREATORS, which, for a contract with resources $R_0, \ldots, R_n$, is defined as $\mathsf{owns}_{\mathsf{creator}(\mathsf{creator}(R_0))}(\mathsf{msg.sender}, 1) * \cdots * \mathsf{owns}_{\mathsf{creator}(\mathsf{creator}(R_n))}(\mathsf{msg.sender}, 1)$.

The function and constructor rules ensure that, at the end of the function or constructor, the multiset of recorded effects is exactly that which has been declared.

Finally, calls can also interact with the resource state. First, any subset of the effects declared by the called function may be recorded by the caller. Second, if a called function declares resource effects w.r.t. a resource R s.t. the current contract has a resource D derived from R, then these

$$
\begin{array}{rcl}
\mathsf{derCreated}(\ \mathsf{transfer}_R(f,\ \mathsf{self}, a)) & = & \mathsf{owns}_D(f, a) \\
\mathsf{derCreated}(\mathsf{exchange}_{R\leftrightarrow R'}(f,\ \mathsf{self}, a_1, a_2, n)) & = & \mathsf{owns}_D(f, n * a_1) \\
\mathsf{derCreated}(\ \mathsf{create}_R(\ \mathsf{self}, a)) & = & \mathsf{owns}_D(\mathsf{msg.sender}, a) \\
\mathsf{derCreated}(\_) & = & \mathsf{emp} \\
\mathsf{derDestroyed}(\ \mathsf{transfer}_R(\ \mathsf{self}, t, a)) & = & \mathsf{owns}_D(t, a) * \mathsf{offers}_{D\leftrightarrow R}(t,\ \mathsf{self}, 1, 1, a) \\
\mathsf{derDestroyed}(\mathsf{destroy}_R(\ \mathsf{self}, a)) & = & \mathsf{owns}_D(\mathsf{msg.sender}, a) * \\
 & & \mathsf{offers}_{D\leftrightarrow R}(\mathsf{msg.sender},\ \mathsf{self}, 1, 1, a) \\
\mathsf{derDestroyed}(\ \mathsf{offer}_{R\leftrightarrow R'}(\ \mathsf{self}, t, a_1, a_2, n)) & = & \mathsf{false} \\
\mathsf{derDestroyed}(\ \mathsf{trust}_c(\ \mathsf{self}, t, v)) & = & \mathsf{false}\ \text{ if } c \text{ declares } R \\
\mathsf{derDestroyed}(\_) & = & \mathsf{emp} \\
\mathsf{derPerformed}(\ \mathsf{transfer}_R(f,\ \mathsf{self}, a)) & = & \{\mathsf{create}_D(f, a)\}^{\#} \\
\mathsf{derPerformed}(\mathsf{exchange}_{R\leftrightarrow R'}(f,\ \mathsf{self}, a_1, a_2, n)) & = & \{\mathsf{create}_D(f, n * a_1)\}^{\#} \\
\mathsf{derPerformed}(\ \mathsf{create}_R(\ \mathsf{self}, a)) & = & \{\mathsf{create}_D(\mathsf{msg.sender}, a)\}^{\#} \\
\mathsf{derPerformed}(\ \mathsf{transfer}_R(\ \mathsf{self}, t, a)) & = & \{\mathsf{destroy}_D(t, a)\}^{\#} \\
\mathsf{derPerformed}(\mathsf{destroy}_R(\ \mathsf{self}, a)) & = & \{\mathsf{destroy}_D(\mathsf{msg.sender}, a)\}^{\#} \\
\mathsf{derPerformed}(\_) & = & \emptyset^{\#}
\end{array}
$$

Fig. 12. Functions describing the implicit consequences of the effects of called functions on derived resources. D is assumed a resource derived from R. We write $\{\ldots\}^{\#}$ for multiset literals.

effects lead to implicit effects on the derived resources. For example, transferring R away to someone implicitly destroys the same number of D they must currently own, and requires that they have offered to exchange that amount of D for the same amount of R. This is captured by the functions derCreated() and derDestroyed(), the former of which defines which derived resources are implicitly created by an effect, and the latter defines which derived resources are implicitly destroyed. Their definitions can be found in Fig. 12. The premise of the call rule ensures that transitive segment constraints etc. hold in a state where destroyed derived resources have already been removed and created derived resources have already been added. The derDestroyed() clause also ensures that no offers are made to give away resource R by a contract that defines D, and that the contract cannot trust someone w.r.t. to the contract that declares R; either of these could lead to some amount of R being removed from the contract without the appropriate checks that the receiver has sufficient amounts of the derived resource D. Finally, the call-rule ensures that all such implicit effects on derived resources (defined by derPerformed()) are also recorded.

C EXTENDED EXAMPLES

Here, we show (extended versions of) the auction and token examples, annotated with different kinds of specifications. The syntax is similar to that of 2Vyper, but simplified for presentation reasons. For the same reason, we have omitted some functions, some parts of the code (in particular, the locks) and some parts of the specification.

C.1 Auction example

The following is a slightly simplified version of the auction contract annotated with resource specifications that use a derived resource. Note that in our tool, a derived resource for wei is declared automatically (meaning that by default, the assumption is that wei sent to a contract should still conceptually belong to the sender, i.e. they get a title for it); here, we have explicitly declared it with the name wei_in_auction for illustration purposes.

```
beneficiary: address
highestBid: int256
```

```
highestBidder: address
ended: bool
pendingReturns: map(address, int256)

#@ resource: good()
#@ resource: wei_in_auction() derived from wei

#@ transitive segment constraint: ... # relate contract state and resource state

#@ segment constraint: msg.sender != self.beneficiary ==> self.ended == old(self.ended)
#@ transitive segment constraint: self.beneficiary == old(self.beneficiary)
#@ transitive segment constraint: old(self.ended) ==> self.ended

#@ performs: create[wei_in_auction](msg.value)
#@ performs: offer[wei_in_auction <-> good](msg.value, 1, to=self.beneficiary, times=1)
@public
@payable
def bid():
    assert block.timestamp < self.auctionEnd
    assert not self.ended
    assert msg.value > self.highestBid
    assert msg.sender != self.beneficiary

    #@ offer[wei_in_auction <-> good](msg.value, 1, to=self.beneficiary, times=1)

    self.pendingReturns[self.highestBidder] += self.highestBid
    self.highestBidder = msg.sender
    self.highestBid = msg.value


#@ performs: destroy[wei_in_auction](self.pendingReturns[msg.sender])
@public
def withdraw():
    pending_amount: wei_value = self.pendingReturns[msg.sender]
    self.pendingReturns[msg.sender] = 0
    send(msg.sender, pending_amount)


#@ performs: exchange[wei_in_auction <-> good](self.highestBid, 1, self.highestBidder,
#@                                             self.beneficiary, times=1)
#@ performs: destroy[wei_in_auction](self.highestBid, actor=self.beneficiary)
@public
def endAuction():
    assert block.timestamp >= self.auctionEnd
    assert not self.ended
    self.ended = True

    #@ exchange[wei_in_auction <-> good](self.highestBid, 1, self.highestBidder,
    #@                                   self.beneficiary, times=1)

    send(self.beneficiary, self.highestBid)
```

C.2 Token example

The following is an extended version of the token contract (namely one that is close to a real implementation of ERC20 in Vyper) using resource specifications and effects-clauses. In our tool, and therefore in this example, effects-clauses are introduced with the keyword performs. The code also contains segment constraints that constrain when *events* are triggered; for a brief description, see our evaluation in the main body of the paper.

```
minter: address
balances: map(address, int256)
```

```
allowances: map(address, map(address, int256))

#@ resource: token()

#@ transitive segment constraint: self.minter == old(self.minter)
#@ transitive segment constraint: self.total_supply == sum(self.balances)

#@ transitive segment constraint: allocated[token]() == balanceOf(self)
#@ transitive segment constraint: forall({o: address, s: address},
#@   self.allowances[o][s] == offered[token <-> token](1, 0, o, s))

#@ segment constraint: forall({a: address, b: address},
#@   self.balanceOf[a] > old(self.balanceOf[a]) and self.balanceOf[b] < old(self.balanceOf[b])
#@     ==> event(Transfer(b, a, self.balanceOf[a] - old(self.balanceOf[a]))))
#@ ... (omitted, more specifications for events)


#@ performs: create[token](_value, to=_to)
@public
def mint(_to: address, _value: uint256):
    assert msg.sender == self.minter
    assert _to != ZERO_ADDRESS
    self.total_supply += _value
    #@ create[token](_value, to=_to)
    self.balanceOf[_to] += _value
    log.Transfer(ZERO_ADDRESS, _to, _value)

#@ performs: transfer[token](_value, to=_to)
@public
def transfer(_to: address, _value: uint256) -> bool:
    self.balanceOf[msg.sender] -= _value
    #@ transfer[token](_value, to=_to)
    self.balanceOf[_to] += _value
    log.Transfer(msg.sender, _to, _value)
    return True

#@ performs: exchange[token <-> token](1, 0, _from, msg.sender, times=_value)
#@ performs: transfer[token](_value, to=_to)
@public
def transferFrom(_from: address, _to: address, _value: uint256) -> bool:
    self.balanceOf[_from] -= _value
    self.balanceOf[_to] += _value
    self.allowances[_from][msg.sender] -= _value
    #@ exchange[token <-> token](1, 0, _from, msg.sender, times=_value)
    #@ transfer[token](_value, to=_to)
    log.Transfer(_from, _to, _value)
    return True

#@ performs: revoke[token <-> token](1, 0, to=_spender)
#@ performs: offer[token <-> token](1, 0, to=_spender, times=_value)
@public
def approve(_spender: address, _value: uint256) -> bool:
    #@ revoke[token <-> token](1, 0, msg.sender, _spender,
    #@                         offered[token <-> token](1, 0, msg.sender, spender))
    self.allowances[msg.sender][_spender] = _value
    #@ offer[token <-> token](1, 0, to=_spender, times=_value)
    log.Approval(msg.sender, _spender, _value)
    return True
```

D FURTHER VERIFIED EXAMPLES

Here, we describe the properties we proved for the contracts not explicitly mentioned in the main body of the paper.

ERC721: ERC721 is a more complex token standard than ERC20. We declared that it implements a token resource whose tokens have identifiers (non-fungible tokens, NFTs), and specified its functions in terms of token transfers and exchanges. Like ERC20, this required using offers and exchanges, but in addition, it also required using trust, since ERC721 allows users to name other users as "operators" who can act on their behalf.

Serenuscoin: Here we use segment constraints prove both access control properties (only the owner may change the factory) and that the correct events are triggered under the right circumstances.

Mana: We verified a simplified version of the Mana application from the VerX paper, where we focused on the parts necessary to show inter-contract invariants between the three collaborating contracts that were also verified (non-modularly) by VerX. Some of these are not single-state invariants but two-state inter-contract transitive segment constraints; one example is that once the token contract's owner has been set to be the continuous_sale contract, its owner will never change again.

Safe remote purchase: This smart contract sells a good to an arbitrary buyer and holds the buyer's funds in escrow until they acknowledge that they have received the good. The contract gives both parties an incentive not to block the other party from receiving funds by holding a deposit from each of them. We use a derived resource for wei (which, as we stated above, our tool declares by default) to model the fact both buyer and seller conceptually own their deposits until the transaction is finalized, at which point the buyer's wei-titles are exchanged for the good, and the deposits can be paid back.