TECHNICAL REPORT

Report No. CS2018-1
Date: November 2018

Securing Calls to Ethereum Smart Contracts with Static and Dynamic Analysis

Shaun Azzopardi
Christian Colombo
Gordon Pace

Department of Computer Science
University of Malta
Msida MSD 06
MALTA

Tel: +356-2340 2519
Fax: +356-2132 0539
http://www.cs.um.edu.mt

Securing Calls to Ethereum Smart Contracts with Static and Dynamic Analysis

Shaun Azzopardi
University of Malta, Malta
shaun.azzopardi.10@um.edu.mt

Christian Colombo
University of Malta, Malta
christian.colombo@um.edu.mt

Gordon J. Pace
University of Malta, Malta
gordon.pace@um.edu.mt

Abstract: *Blockchains decentralise computation and storage, reducing the possibility that vulnerabilities in one node can affect a whole network. Ethereum is a popular implementation of these that acts as a distributed operating system for programs called smart contracts. Issues with these smart contracts have led to millions of tokens with real-world value being lost, motivating the need for formal analysis. One delicate issue is that once a smart contract is deployed to an address on the blockchain, its behaviour can be dependent on other smart contracts on the blockchain space, not just its own code or state. This creates issues for verification of a smart contract, given sound analysis requires modeling function calls to other smart contracts. In this paper we consider the case of developed smart contracts which can use the functionality of other smart contracts possibly not developed by the same person. We propose the use of assume-guarantee reasoning that allows us to provide static guarantees about call sites in a user's smart contract given assumptions provided by the service provider, assumptions which can then be discharged at runtime. We use symbolic automata as our specification language, defining a strictness relation between them, and a quotient or residual operation that allows us to prune proven parts away from a specification.*

Securing Calls to Ethereum Smart Contracts with Static and Dynamic Analysis

Shaun Azzopardi
University of Malta, Malta
shaun.azzopardi.10@um.edu.mt

Christian Colombo
University of Malta, Malta
christian.colombo@um.edu.mt

Gordon J. Pace
University of Malta, Malta
gordon.pace@um.edu.mt

Abstract: *Blockchains decentralise computation and storage, reducing the possibility that vulnerabilities in one node can affect a whole network. Ethereum is a popular implementation of these that acts as a distributed operating system for programs called smart contracts. Issues with these smart contracts have led to millions of tokens with real-world value being lost, motivating the need for formal analysis. One delicate issue is that once a smart contract is deployed to an address on the blockchain, its behaviour can be dependent on other smart contracts on the blockchain space, not just its own code or state. This creates issues for verification of a smart contract, given sound analysis requires modeling function calls to other smart contracts. In this paper we consider the case of developed smart contracts which can use the functionality of other smart contracts possibly not developed by the same person. We propose the use of assume-guarantee reasoning that allows us to provide static guarantees about call sites in a user's smart contract given assumptions provided by the service provider, assumptions which can then be discharged at runtime. We use symbolic automata as our specification language, defining a strictness relation between them, and a quotient or residual operation that allows us to prune proven parts away from a specification.*

1 Introduction

Services dependent on a central authority are liable to disruption should that authority be compromised. Distributed ledger technologies (DLTs) are a solution to these single points of failure — by replicating work over all the nodes of a network they

remove the need for a central authority. A popular and common usage for DLTs is in the form of blockchains that are mainly, but not only, being used to provide a backbone to cryptocurrencies. These services are provided on blockchains through the use of smart contracts, which can be Turing-complete (e.g. on Ethereum). Users can call these services from outside the blockchain or own smart contracts that use these services. Despite that deployed smart contract code cannot be changed post-deployment, its behaviour can still be updated at runtime, by changing which smart contracts it delegates work to. The potential for change in these dependencies at runtime motivates the use of analysis to ensure the service provided continually meets its users' expectations.

Program verification techniques depend on inferring the program's behaviour by either analysing the code of the program, or by monitoring the program's behaviour at runtime. The first, static analysis, is more powerful in that it attempts to compute the whole state space of a program (or approximations of it) but is difficult for Turing complete programs. On the other hand, runtime monitoring can in general only verify safety properties completely [7], since it does not explore the whole state space of the program but just the execution trace at runtime, while it adds overheads to the program at runtime. The possible variability in a smart contract's control-flow behaviour at runtime reduces the reach of static analysis — behaviour of calls to another smart contract cannot be known statically since they may not even have been created yet, motivating the need for runtime monitoring.

Assume-guarantee reasoning is a possible approach to handling this variability — by making assumptions about the unknown (or variable) dependencies we can then ensure some guarantees about the interaction of our smart contract with the dependency statically, while we discharge these assumptions by verifying them on the dependency at runtime [16]. An issue then is how to acquire these assumptions. Automata learning techniques can be used to automate this [9], however we are not just interested in proving a service smart contract safe to use according to a user's need, but we want some assurances from the service provider about its behaviour. As with real-world services, which come with enforceable terms of service, we can then use these assurances to hold the service provider liable at runtime for any misbehaviour.

Consider the example of a smart contract acting as an agent for an electric car and another acting on behalf of an electric car battery swapping station, as illustrated partially in Listing. 1 and Listing. 2. Here both smart contracts assume an API of the other, for example the car smart contract assumes the station address has a `swap(uint, uint)` function. At a coarse level we can check for incompatibility between different smart contracts by checking the APIs they provide. Here we propose extending this by considering behavioural compatibility, not only through static but also dynamic methods since certain points in a smart contract may have unpredictable

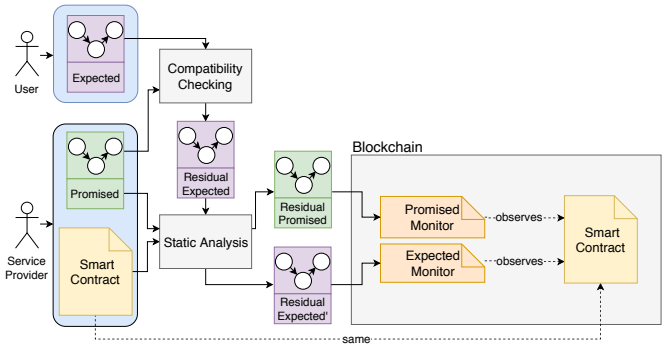


Figure 1: Model-based verification process for smart contracts.

behaviour at runtime (e.g. line 7 in Listing. 1).

```
1 contract Car{
2   Station station;
3   uint batteryId;
4   ...
5   function requestSwap(uint loc) returns (uint)
6   {
7     return station.swap(batteryId,fromLoc);
8   }
9 }
```

Listing 1: Car smart contract.

```
1 contract Station{
2   ...
3   function swap(uint id, uint loc) returns (uint)
4   {
5     ...
6     if(Car(msg.sender).propose(newBatteryID))
7       return newBatteryID;
8   }
9 }
```

Listing 2: Station smart contract.

Inspiring ourselves from assume-guarantee reasoning and real-world provision of services, we thus assume that service smart contracts come with an associated terms of service, or a *promised model* of behaviour (e.g. a station may promise to always give out batteries with charge higher than 90%). We can check for compatibility of the promised model with the behaviour a user expects, encoded similarly in an *expected model* (e.g. a car requires batteries with charge larger than 80% and to be from a certain manufacturer), through model checking techniques. Promising certain behaviour does not however ensure it — a user has no reason to trust the service provider. We propose that the user can use static analysis (prior to allowing a dependency on the service) to prove as much as possible of the service's promised behaviour, leaving any statically unproved parts for runtime. Moreover, certain expected behaviour may not be promised, but a user may still want to use the service. In such a case we can try to enforce this expected but not promised behaviour, if we limit ourself to safety properties [5]. This process is illustrated in Figure 1.

In this scenario the user is motivated by the need to reduce vulnerabilities in the services they use, which may have negative effects on their own smart contract. This

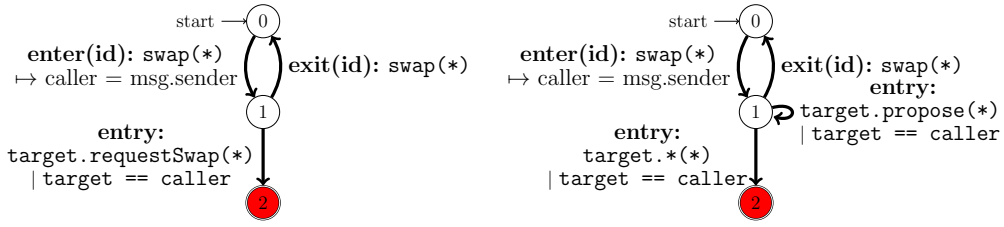
holds regardless of a promised model, however such a model allows users to find services that are promised to be appropriate for their need. Moreover, the service provider can be held accountable in case the service does not behave as promised, which acts as a further incentive for the user. For the service provider this framework provides them with a way to incentivize users to use their service as opposed to just copying the public code or create their own implementation.

Our contributions here include a sound approach to verification combining model checking, static analysis, and runtime verification. We propose defining behavioural models in terms of *Dynamic Event Automata* [4] (DEAs), a kind of symbolic automata that can be used to runtime verify smart contracts [6], and present a method to check for strictness between safety properties as DEAs, along with a quotient or residual operator to allow for combination of different verification methods. We propose these as the formal foundation for a framework to secure dependencies between smart contracts in Ethereum.

In Section 2 we motivate this framework further with examples and define formally DEAs, an abstraction for smart contracts and the required formal tools for analysis. In Section 3 we consider a case study involving a token wallet smart contract, and present experimental results with respect to this and other case studies. We discuss related work in Section 5 and this approach in Section 4, while we conclude in Section 6.

2 Securing Smart Contract Dependencies and Calls

We propose using Dynamic Event Automata (DEAs) both from the perspective of its owner and its user as a language to specify the normative behaviour of a smart contract. These are symbolic automata with transitions triggered by events, some condition and that possibly activate an action, denoted by $state_1 \xrightarrow{event|condition \mapsto action} state_2$. As a motivating example consider Listing. 2, for which we can define re-entrancy properties, a main issue of consideration for the analysis of smart contracts given previous vulnerabilities involving re-entrancy (or recursive calls) [14, 8]. Fig. 2(a) can be used by a car owner to specify that stations are expected not to call the `requestSwap` function of the car, while in the process of swap. Fig. 2(b) on the other is a promise by the station that `swap` may only re-enter the car through its `propose` function. In these examples we use `entry`(id) and `exit`(id') to denote respectively the point of entry and exit of a function, with id being some identifier, e.g. $id \in \mathbb{N}$, where $id == id'$ implies the events refer to the same function call. Red states denote bad states that upon being reached activate a violation, and possibly some reparation code, while we



(a) `swap` is expected not to re-enter its caller through `requestSwap`.

(b) `swap` can only re-enter its caller through `propose`.

Figure 2: Expected and promised model for the battery swapping station.

use $*$[1] to lazily match anything placeable in the position it is in, e.g. the transition from state 1 to state 2 in Fig. 2(b) matches any event except an entry into `propose`. DEAs allow us to define more sophisticated behaviour than this, only limited by which events are observable to the monitor.

We have previously proposed DEAs to allow for safely mutable smart contracts [4], where through a proxy pattern a smart contract is used as an interface for another, while DEAs are instrumented in the interface to impose safety constraints on the behaviour exhibited at runtime. This allows for maintainability of smart contracts while providing a behavioural contract any implementation will satisfy. This presents a use case for the approach we propose here, where the behavioural contract enforced at runtime corresponds to the model promised by the service provider. However, in this case all the implementation is abstracted away with a call to another smart contract, while in other cases some concrete code may be present with statically determinable behaviour.

Here then we propose a framework that uses both the concrete code of a smart contract and the assumption that it will obey a certain specification at runtime (as ensured through such a monitor) as the basis of analysis for users who want to verify the smart contract. If we assume that the rest of the code hidden in Listing. 2 is delegated to another smart contract, e.g. to allow updateable behaviour, then we cannot verify either of the models against it. However, if the car owner expects Fig. 2(a) to be true of the station we can check this against the promised model Fig. 2(b), which allows to determine compatibility between the requirements of the user and promised behaviour, requiring only the promised model to be verified at runtime.

In the literature assume-guarantee reasoning is prevalent when attempting the verification of systems with multiple components, in an attempt to reduce the state explosion associated with considering the synchronous behaviour of components (e.g.

[1] We use this as syntactic sugar to allows us to write smaller properties here, but it is not part of the language.

[9]). Here we do not consider the interactive behaviour of these smart contracts, given it can be complex, and leave it for future work, but we focus on the service's behaviour. Given a dependency to be filled, we consider a *promised model* (*PM*) a service provider promises of their service, and an *expected model* (*EM*) that the user expects out of the dependency. *PM* then functions as the assumption about the dependency which we use to ensure the guarantees required, as encoded in *EM*. Formally, given a compliance operator $\vdash$ and a strictness operator (or refinement) between models $\succeq_s$, if we can prove that the assumption is stricter than the guarantee, $PM \succeq_s EM$, and we manage to prove that the smart contract respects the assumption, $SC \vdash PM$ using a combination of static and dynamic analysis, then we have proven that $SC \vdash EM$.

If $PM \succeq_s EM$ and $SC \vdash PM$ are cheaper to compute directly than $SC \vdash EM$ we gain from this approach, otherwise the approach is still sound and the user is still motivated to analyse for compliance with the promised model to prevent bad behaviour and receive reparations. For example, for Fig. 2(b) this can be monitored for at runtime, where reparations can include reverting the bad behaviour and the payment of some *a priori* agreed fine.

2.1 Preliminaries

2.1.1 Models

As a formalism for models we use Dynamic Event Automata (DEA) [6], which are symbolic automata. Their transitions are triggered by method calls or variable changes, are conditional on the DEA's or the smart contract's variable state, and also possibly activate some action.

Definition 1 *A* dynamic event automata *over a set of symbolic monitoring states* Θ*, and a set of symbolic program states* Ω *is a tuple* $\pi = \langle \Sigma, Q, q_0, B, \theta_0, R, \rightarrow \rangle$*, where: (i)* Σ *is a set of events; (ii)* Q *is the set of explicit monitoring states; (iii)* $q_0 \in Q$ *is the initial state; (iv)* $B \in 2^Q$ *is a set of bad states; (v)* $\theta_0 : \Theta$ *is the initial symbolic monitoring state; and (vi)* $\rightarrow$*:* $Q \times \Sigma \times (\Omega \times \Theta \rightarrow Bool) \times (\Omega \times \Theta \rightarrow \Theta) \times Q$ *is the* deterministic *transition function with conditions that predicate over symbolic program and monitoring states and actions that instead act on a symbolic monitoring state, where bad states do not have outgoing transitions.*

We write $q \xrightarrow{e|c \mapsto a} q'$ *for* $(q, e, c, a, q') \in \rightarrow$*, and* $q \overset{e}{\not\rightarrow}$ *for* $\nexists c, a, q' \cdot q \xrightarrow{e|c \mapsto a} q'$*. We also write* $\Rightarrow$ *for its transitive closure, and* $q \Rightarrow q'$ *for* $\exists\, t \in \Sigma^* \cdot q \overset{t}{\Rightarrow} q'$*. We use* `true` *and* `skip` *to respectively denote the condition that always holds true* ($\lambda\, x.\, \lambda\, y.true$) *and the*

identity action $(\lambda\, x.\, \lambda\, y.y)$.

We give the operational semantics of DEAs in terms of configurations of explicit and symbolic DEA states, where their evolution is driven by events and program state pairs. This allows us to identify the program traces that drive a DEA to a bad state.

Definition 2 *The* operational semantics *of a DEA* π *is a labeled transition system over configurations of type* $\Theta \times Q$ *with transitions labeled by events from* $\Sigma \times \Omega$ *and characterized by the following rules: (i) a configuration* (q, θ) *evolves with a label* (e, ω) *if there is a corresponding transition* $q \xrightarrow{e|c \mapsto a} q'$ *in the DEA where the condition* c *holds true for* (θ, ω) *and the symbolic state evolves to* $(q', a(\theta, \omega))$*; and (ii) if there is no applicable transition in the DEA the configuration does not change.*

$$\frac{q \xrightarrow{e|c \mapsto a} q' \qquad c(\theta, \omega) \qquad q \notin B}{(q, \theta) \xrightarrow{e,\omega} (q', a(\theta, \omega))} \qquad \frac{q \xrightarrow{e,\omega} \not\rightarrow}{(q, \theta) \xrightarrow{e,\omega} (q, \theta)}$$

We overload $\Rightarrow$ *to be its transitive closure.*

The violating traces *of a DEA* π *are all the traces that reach a bad state of the DEA:* $V(\pi) = \{t : (\Sigma \times \Omega)^* \mid \exists\, q_B \in B_\pi \cdot \exists\, \theta \cdot (q_0, \theta_0) \overset{t}{\Rightarrow} (q_B, \theta)\}$.

By case analysis of the rules premises it follows easily that a small-step can always be taken

Proposition 1 *Given any configuration, and a smart contract event and symbolic state, there is always a rule in the semantics that can be applied:* $\forall\, e, \omega, \theta, q \cdot \exists\, q', \theta' \cdot (q, \theta) \xrightarrow{e,\omega} (q', \theta')$

Proof

This follows easily from the fact that the condition of the second rule is the negation of the first, and thus all possible conditions are covered. □

We can extend this result to traces.

Proposition 2 *Any smart contract trace induces a certain DEA configuration:* $\forall\, ews \cdot \exists\, q, \theta \cdot (q_0, \theta_0) \overset{ews}{\Longrightarrow} (q, \theta)$

Proof

This follows easily on induction on the trace, by applying Proposition. 1 as needed. □

With these semantics DEAs can be used to specify safety properties. These DEAs can also be used to encode reparations when a violating trace is detected at runtime [6], by associating bad states with a piece of code to be executed upon reaching it. This code can, for example, simply cancel a transaction, thereby avoiding violation. More involved reparation logic can be executed after violation, as determined *a priori* by the parties, but we do not consider this further since we are only interested in analysing the behaviour of the service provider for compliance (see [5, 4] for more detail on how non-compliance at runtime can be dealt with using DEAs).

We use DEAs to encode both promised and expected models, while to allow us to compare them we define what it means for a DEA to be stricter than another in terms of violating traces inclusion.

Definition 3 *A DEA D is said to be* stricter *than a DEA D' iff the violating traces of D contain the violating traces of D': $D \succeq_s D' \stackrel{\text{def}}{=} V(D) \supseteq V(D')$*

We can attempt to compute this using synchronous composition of two DEAs. We define this by considering every possible combination of transitions and condition truth from the DEAs' states.

Definition 4 *The* synchronous composition *of two DEAs D_1 and D_2, $D_1||D_2$, over the same alphabet Σ is the DEA with states of the form $Q_1 \times Q_2$, initial explicit state (q_{0_1}, q_{0_2}), initial symbolic state $(\theta_{0_1}, \theta_{0_2})$, and bad states from the set $\{(q, q') \mid q \in B_{D_1} \wedge q' \in B_{D_2}\}$. The transitions are generated by the following rules when considering a state (q_1, q_2) and an event e: (i) if q_1 does not transition with e but q_2 does with some transition to q_2' then the transition is replicated in the composition to (q_1, q_2'), and (ii) similarly for when q_1 transitions but not q_1; while (iii) if both transition then every possible productive combination of conditions is considering for transitioning:*

$$\frac{q_1 \overset{e}{\not\rightarrow}, q_2 \xrightarrow{e|c' \mapsto a'} q_2'}{(q_1, q_2) \xrightarrow{e|c' \mapsto a'} (q_1, q_2')} \qquad \frac{q_1 \xrightarrow{e|c \mapsto a} q_1', q_2 \overset{e}{\not\rightarrow}}{(q_1, q_2) \xrightarrow{e|c \mapsto a} (q_1', q_2)}$$

$$\frac{q_1 \xrightarrow{e|c \mapsto a} q_1', q_2 \xrightarrow{e|c' \mapsto a'} q_2'}{(q_1, q_2) \xrightarrow{e|c \wedge c' \mapsto \{a;\, a'\}} (q_1', q_2'), \quad (q_1, q_2) \xrightarrow{e|c \wedge \neg c' \mapsto a} (q_1', q_2), \quad (q_1, q_2) \xrightarrow{e|\neg c \wedge c' \mapsto a'} (q_1, q_2')}$$

We assume conditions and actions act only piecewise on their respective automata's state, i.e. if $q_1 \xrightarrow{e|c\mapsto a} q_1'$ *is in* D_1 *then* $c((\theta_1, \theta_2), \omega) = c(\theta_1, \omega)$*, and* $a((\theta_1, \theta_2), \omega) = a(\theta_1, \omega)$*.*

2.1.2 Smart Contracts

A smart contract is not necessarily a closed system, but may communicate with other smart contracts on the blockchain. A blockchain state here can be thought of map from addresses to smart contracts, where smart contracts have some variable state and some function API. Here we give a formal characterisation of this. We start by using automata to represent functions, which allows us to define blockchain states and give a semantics for smart contract executions.

Our representation for smart contract functions is similar to DEAs, except here this automaton is meant to model the behaviour exhibited by the function rather than the function it should exhibit. Here we abstract away the variable state with symbolic states taken from set Ω. We assume the function has been instrumented with the events either from the property event set or an empty action, i.e. from the set $\Sigma_\epsilon \stackrel{\text{def}}{=} \Sigma \cup \{\epsilon\}$[2]. Smart contracts functions may also call each other, which we model by associating some states with address and smart contract function (meant to refer to the function at the given address in the current blockchain state). Moreover, we have two different kinds of calls here, `call` and `delegatecall`. Here we do not go into the semantics of smart contracts since we do not need it to present our approach, however in Appendix. A.1 we define a semantics for smart contracts that also takes into account the different control-flow effects of calls and delegate calls.

Definition 5 *A* smart contract function automaton *instrumented with a property event set* Σ*, of type* $\mathcal{SCF}$*, is a tuple* $SCF = \langle S, s_0, F, R, stmt, call, dcall \rightarrow \rangle$*, where: (i)* S *is a set of states, (ii)* $s_0 \in S$ *is the initial explicit smart contract state, (iii)* $F, R \subseteq S$ *are respectively the non-empty sets of final and revert*[3] *explicit states, (iv)* $stmt : S \mapsto \texttt{STMT}$ *associates explicit states with statements, (v)* $call, dcall : S \times \Omega \mapsto \texttt{Addr} \times \mathcal{SCF}$ *associates states, given some symbolic state, with function calls at some address, and (vi)* $\rightarrow \in S \times \Sigma \times (\Omega \mapsto Bool) \mapsto S$ *is the transition function which is complete and deterministic with respect to the condition on the symbolic state, where final and revert states do not have outgoing transitions. We write* $s \xrightarrow{e|c} s'$ *for* $(s, e, c, s') \in \rightarrow$*.*

[2]Empty actions represent program transitions that do not match a property event.

[3]When reached revert states cancel any changes to state since the start of the transaction and end the execution.

Smart contracts can then be represented as sets of functions with some symbolic variable state.

Definition 6 *A* smart contract abstraction *instrumented with a property event set Σ, of type $\mathcal{SC}$, is a tuple $SC = \langle \omega, fs \rangle$, where ω is the symbolic smart contract state, $fs \subseteq \mathcal{SCF}$ is a set of functions. We write ω_{SC} to refer to the symbolic state of the smart contract SC, and fs_{SC} to refer to its functions. We use $L(SC) \subseteq (\Sigma \times \Omega)^*$ to refer to the language of the instrumented smart contract at runtime*[4].

Here we consider over-approximations of the semantics of these automata in terms of event traces, using finite-state automata with transitions labeled by events from Σ_ϵ. We consider a possible abstraction that identifies the possible successful control-flow behaviour (ignoring traces that reach a revert state, that are never recorded to the blockchain), allowing for functions to be called in any order and any behaviour at call sites.

Definition 7 *The* coarse control-flow abstraction *of a smart contract SC, denoted $CF(SC)$, is the finite-state automaton with: (i) initial state s_0 being also the only possible final state; (ii) states including the union of the function states: $S_{SC} \stackrel{\text{def}}{=} \{s_0\} \cup (\bigcup_{f \in SCFs} S_f)$*[5]*; and (iii) transitions characterized by the following rules: (a) the initial state can transition into any of the functions; (b) function transitions are replicated without conditions; (c) functions' final states transition back to the initial/end state, while (d) any event trace can happen at a call site:*

$$\frac{f \in SCFs}{s_0 \xrightarrow{\epsilon} s_{0_f}} \qquad \frac{s_f \xrightarrow{e|c} s'_f}{s_f \xrightarrow{e} s'_f} \qquad \frac{s_f \in F_f}{s_f \xrightarrow{\epsilon} s_0} \qquad \frac{s_f \in dom_f(allCalls) \quad e \in \Sigma}{s_f \xrightarrow{e} s_f}$$

We overload $\Rightarrow$ as its transitive closure.

We further define the successful transition relation $\rightsquigarrow$ *as the smallest relation that only allows transitioning to states that cannot revert (i.e. that can reach s_0 in the abstraction*[6]*), while skipping ϵ events: for $e \in \Sigma$ we write $s \overset{e}{\rightsquigarrow} s'$ if $\exists\, t \cdot s' \overset{t}{\Rightarrow} s_0 \wedge (s \xrightarrow{e} s' \vee (\exists\, s'' \cdot s \xrightarrow{\epsilon} s'' \wedge s'' \overset{e}{\rightsquigarrow} s'))$, and use $\overset{*}{\rightsquigarrow}$ as its transitive closure. Then we define the abstract language of a smart contract in terms of the traces from Σ^* reaching the end (or initial) state in this semantics: $AL(SC) \stackrel{\text{def}}{=} \{t \in \Sigma^* \mid s_0 \overset{t}{\rightsquigarrow}_{CF(SC)} s_0\}$.*

This abstraction of smart contracts is similar to our chosen representation for models, which allows for interoperability. However automata here act as descriptions of actual

[4]See Appendix. A.1

[5]We assume each function uses unique states.

[6]Note that in the abstraction revert states will have no outgoing transitions.

behaviour, while DEAs are used as normative specifications. In Appendix. A.2 we show how this abstract language approximates the language of the smart contract.

We will attempt to verify a smart contract statically by simulating the monitoring of a smart contract by a DEA with synchronous composition of the coarse abstraction and the DEA.

Definition 8 *The synchronous composition* $\triangleright$ *of a smart contract abstraction SCA and a dynamic event automaton* D *over the same alphabet* Σ *has* $q_0 = (s_0, q_{0_D}), \theta_0 = \theta_{0_D}$*, and* $B = \{(s, q') \mid q' \in B_D\}$*, with:*

$$\frac{s \overset{e}{\rightsquigarrow} s', q \xrightarrow{e|c \mapsto a} q'}{(s,q) \xrightarrow{e|c \mapsto a} (s', q')} \qquad \frac{s \overset{e}{\rightsquigarrow} s', q \overset{e}{\not\mapsto}}{(s,q) \xrightarrow{e|true \mapsto \texttt{skip}} (s', q)}$$

We can show that the synchronous composition DEA behaves equivalently to the original DEA, in that given a trace in the language of the smart contact then the configuration reachable in the composition has the same property explicit state and symbolic state as would be induced in the original DEA. First we show this for any prefix, and then specialise the result for violation traces.

Lemma 1 *For any trace in the language of a smart contract SC, then any prefix of it will lead to the same property explicit and symbolic state in the monitoring semantics of a property* D *and property produced by its synchronous composition with* SC*,* $D \triangleright SC$*:* $\forall\, ews \in L(SC), ews' \in \mathit{prefixes}(ews) \cdot \exists\, q, \theta \cdot ((q_0, \theta_0) \overset{ews'}{\Longrightarrow}_D (q, \theta)) \Leftrightarrow (\exists\, s \cdot ((s_0, q_0), \theta_0) \overset{ews'}{\Longrightarrow}_{D \triangleright SC} (s, (q, \theta)))$.

Proof

We show this by induction on the length of ews.

BC: $\langle\rangle$

Proof:

$((q_0,\theta_0) \stackrel{\langle\rangle}{\Rightarrow}_D (q_0,\theta_0)) \Leftrightarrow (((s_0,q_0),\theta_0) \stackrel{\langle\rangle}{\Rightarrow}_{D \triangleright SC} ((s_0,q_0),\theta_0))$ holds trivially by definition of $\Rightarrow$

IH: Holds for $ews' \in \mathit{prefixes}(es)$

IC: Prove for $ews' \texttt{++} \langle(e,\omega)\rangle \in \mathit{prefixes}(es)$

Proof:

Suppose $(q_0,\theta_0) \stackrel{ews'}{\Longrightarrow}_D (q,\theta) \xrightarrow{\langle(e,\omega)\rangle} (q',\theta')$

$\Leftrightarrow$ (*by inductive hypothesis*)

$(q,\theta) \xrightarrow{\langle(e,\omega)\rangle} (q',\theta') \wedge \exists\, s \cdot ((s_0,q_0),\theta_0) \stackrel{ews'}{\Longrightarrow}_{D \triangleright SC} ((s,q),\theta)$

$\Leftrightarrow$ (*by Corollary. 1*)

$(q,\theta) \xrightarrow{\langle(e,\omega)\rangle} (q',\theta') \wedge \exists\, s, s' \cdot s \stackrel{e}{\rightsquigarrow} s' \wedge ((s_0,q_0),\theta_0) \stackrel{ews'}{\Longrightarrow}_{D \triangleright SC} ((s,q),\theta)$

$\Leftrightarrow$ (*by definition of synchronous composition and DEA semantics*)

$\exists\, s, s' \cdot ((s_0,q_0),\theta_0) \stackrel{ews'}{\Longrightarrow}_{D \triangleright SC} ((s,q),\theta) \xrightarrow{\langle(e,\omega)\rangle} ((s',q'),\theta')$ □

Lemma 2 *Given a smart contract's trace, then if it is in the violation traces of D then it is also in the violation traces of the residual of D with respect to the smart contract: $ews \in L(SC) \Rightarrow (ews \in V(D) \Leftrightarrow ews \in V(D \triangleright SC))$*

Proof

Suppose $ews \in L(SC)$.

$ews \in V(D)$

$\Leftrightarrow$ (by definition of $V(D)$)

$\exists\, \theta, q_B \in B \cdot (q_0,\theta_0) \stackrel{ews}{\Longrightarrow} (q_B,\theta)$

$\Leftrightarrow$ (*by Lemma. 1*)

$\exists\, \theta, s, q_B \in B \cdot ((s_0,q_0),\theta_0) \stackrel{ews}{\Longrightarrow} ((s,q_B),\theta)$

$\Leftrightarrow$ (by definition of $V(D)$)

$ews \in V(\triangleright SC)$ □

2.2 Combining Model Checking and Runtime Verification

We wish to be able to prove the promised model to be stricter than the expected model, while showing that the service smart contract respects the promised model. We may not always be able to do this statically, since we have to consider over-approximations of the smart contract's behaviour (given call sites at runtime may point to any smart contracts). Our solution here is then to attempt to prove as much as possible statically and leave the rest for runtime.

Statically we are generating an over-approximation of the smart contract's behaviour, by analysing its code, and model checking this against the DEA.

Definition 9 *A smart contract SC is* statically compliant *with a DEA D if its abstract language has no violating trace:* $SC \vdash_{SA} D \stackrel{\text{def}}{=} AL(SC) \cap T(V(D)) = \varnothing$.

On the other hand, runtime verification (RV) checks for set membership of the execution trace against the violating language of the DEA — we define this by assuming an oracle function $run : SC \mapsto (\Sigma \times \Omega)^*$ that returns the execution trace of the run chosen at runtime, that satisfies $run(SC) \in L(SC)$.

Definition 10 *A smart contract SC is said to be* runtime compliant *with a DEA D if the execution trace of SC satisfies D:* $SC \vdash_{RV} D \stackrel{\text{def}}{=} run(SC) \notin V(D)$.

Note then how finding the smart contract to be compliant through static analysis means that smart contract will be compliant at runtime.

Theorem 1 *If a smart contract SC is statically compliant with a DEA D, then SC is also compliant with D at runtime:* $SC \vdash_{SA} D \Rightarrow SC \vdash_{RV} D$.

Proof

$SC \vdash_{SA} D$
$\Rightarrow$ (by its definition)
$AL(SC) \cap T(V(D)) = \varnothing$
$\Rightarrow$ (by Thm.5 $run(SC) \in L(SC) \Rightarrow T(run(SC)) \in AL(SC)$)
$T(run(SC)) \notin T(V(D))$
$\Rightarrow$ (contrapositive of Proposition.4)
$run(SC) \notin V(D)$
$\Rightarrow$ (by its definition)
$SC \vdash_{RV} D$ □

However leaving property checking for runtime may be costly, since deploying monitoring code and calling a function has associated fees, creating disincentives against RV. Here we propose the use of quotient or residual operators [2, 3, 18] to identify what has been partially proven statically (or what is ensured by the part of the code whose behaviour cannot change). The part of the property that could not be verified statically is then be moved to runtime, which is always possible for DEAs, since they define safety properties which are monitorable [7].

In previous work we have already considered techniques to prune a DEA using a trace-based static program over-approximation [3], here we extend this to work directly on a finite-state representation of the program (or smart contract). This reduction removes from a DEA transitions that are not used by a smart contract or which the smart contract cannot take to a violation.

Definition 11 *The residual of DEA D with respect to a smart contract SC, $D \setminus SC$, is defined to be D reduced to D's transitions used in the synchronous composition with an outgoing state that is both reachable from the initial state and can reach a bad state:* $\rightarrow_{D \setminus SC} \stackrel{\mathsf{def}}{=} \{q \xrightarrow{e|c \mapsto a} q' \mid \exists\, s, s', s'' \in S, q_B \in B \cdot (s_0, q_0) \Rightarrow (s, q) \xrightarrow{e|c \mapsto a} (s', q') \wedge (s, q) \Rightarrow (s'', q_B)\}$.

We can show that monitoring for the original DEA and this structurally smaller DEA is equivalent for SC. We show this by showing that a trace of SC is violating in the residual iff it is violating in the synchronous composition, which then allows us to conclude that it is also violating in the original property.

Lemma 3 *The residual maintains the same violating traces as the synchronous composition:* $ews \in L(SC) \Rightarrow (ews \in V(D \setminus SC) \Leftrightarrow ews \in V(D \triangleright SC))$

Proof

Suppose $ews \in L(SC)$.

$$
\begin{array}{ll}
 & ews \in V(D \triangleright SC) \\
\Leftrightarrow & (\textit{by definition of } V(D \triangleright SC)) \\
 & \exists\, \theta, s, q_B \in B \cdot ((s_0, q_0), \theta_0) \stackrel{ews}{\Longrightarrow} ((s, q_B), \theta) \\
\Leftrightarrow & (\textit{by definition of } \Rightarrow \textit{ and the previous line}) \\
 & \exists\, \theta, s, q_B \in B \cdot ((s_0, q_0), \theta_0) \stackrel{ews}{\Longrightarrow} ((s, q_B), \theta) \wedge \\
 & \forall\, ews' \in \textit{prefixes}(ews) \cdot ((s_0, q_0), \theta_0) \stackrel{ews'}{\Longrightarrow} ((s', q'), \theta') \wedge \\
 & \quad (s', q') \Rightarrow (s, q_B) \\
\Leftrightarrow & (\textit{by definition of } D \setminus SC \textit{ then, since every transition here is from a state} \\
 & \textit{that can reach a bad state, each of the transitions used are kept in } D \setminus SC) \\
 & \exists\, \theta, s, q_B \in B \cdot (q_0, \theta_0) \stackrel{ews}{\Longrightarrow}_{D \setminus SC} (q_B, \theta) \\
\Leftrightarrow & (\textit{by definition of } V(D \setminus SC)) \\
 & ews \in V(D \setminus SC) \qquad \square
\end{array}
$$

Theorem 2 *The residual maintains the same violating traces as the original property, for traces of the smart contract:* $ews \in L(SC) \Rightarrow (ews \in V(D) \Leftrightarrow ews \in V(D \setminus SC))$

Proof

This follows immediately from Lemma.3 and Lemma.2.

Then we can show that at runtime monitoring either for D or for the residual is equivalent.

Theorem 3 *A smart contract SC is compliant at runtime with DEA D if and only if it is compliant at runtime with residual* $D \setminus SC$*:* $SC \vdash_{RV} D \Leftrightarrow SC \vdash_{RV} D \setminus SC$.

Proof

$SC \vdash_{RV} D$
$\Rightarrow$ (by its definition)
$run(SC) \notin V(D)$
$\Rightarrow$ (by contrapositive of Lemma.2)
$run(SC) \notin V(D \setminus SC)$
$\Rightarrow$ (by its definition)
$SC \vdash_{RV} D \setminus SC$ □

For example, if we show that the smart contract static over-approximation of a battery swaaping engiine never calls the `requestSwap` function in the station smart contract, then we do not need to listen to calls for this function when monitoring for Fig. 2(b). In effect residual operations allow us to decompose DEAs in two parts: the part proven statically and the part to be proven at runtime. Since the residual we defined considers only a subset of the transitions of the original DEA then monitoring this residual always requires the same or less business logic to be instrumented in the code.

We have then identified the tools necessary to verify a promised model against a smart contract, possibly leaving a residual to prove at runtime. What remains for our framework is the formal tools to be able to compare DEAs with each other. We compare two DEAs with synchronous composition, and compute a residual DEA that includes the part of the first DEA not to be determined to be ensured by the second DEA. If this residual is the always satisfied property then we can determine strictness between the two DEAs as per Defn. 3.

Assuming the synchronous composition has been pruned soundly to remove non-viable transitions (i.e. with contradictory conditions), then consider that a trace is violating for both D_1 and D_2 if it can reach a state tagged with bad states of both DEAs in their composition. If there is a trace that reaches a state tagged by a bad state of D_1 and a normal state of D_2 then $V(D_1) \not\subseteq V(D_2)$, i.e. D_2 is not stricter than D_1. Such traces are then exactly those violating D_1 that do not violate D_2. Then we keep a transition $q_1 \xrightarrow{e|c \mapsto a} q_1'$ of D_1 in its residual w.r.t. to D_2 if it is used in the reachable part of the synchronous composition, and from a state (q_1, q_2) to some state (q_1', q_2'), and where (q_1, q_2) can also reach a state composed of a bad state of D_1 but not of D_2. Note that if (q_1', q_2') never violates the transition is still kept because of the latter condition, this ensures that any non-violating trace in D_1 remains non-violating in its residual.

Definition 12 *The* residual DEA *of D_1 with respect to D_2, $D_1 \setminus D_2$, is D_1 with the transitions of D_1 used in the synchronous compositions from states that can reach bad states:* $\rightarrow_{D_1 \setminus D_2} \stackrel{\text{def}}{=} \{q_1 \xrightarrow{e|c \mapsto a} q_1' \mid \exists\, q_2, q_2' \cdot (q_{0_1}, q_{0_2}) \Rightarrow (q_1, q_2) \xrightarrow{e|c \wedge c' \mapsto a;\ a'} (q_1', q_2') \wedge \exists (q_b, q_2'') \in B_1 \times (Q_2 \setminus B_2) \cdot (q_1, q_2) \Rightarrow (q_b, q_2'')\}$.[7][8]

To talk about DEAs without enumerating every possible smart contract trace we consider an abstraction function from such traces to symbolic traces, corresponding to the event-condition-action triples on DEA transitions.

Definition 13 *The* symbolic abstraction *of a smart contract trace $ews \in (\Sigma \times \Omega)^*$ is a trace of event-condition-action triples of the same length, $A : (\Sigma \times \Omega)^* \times \mathbb{D} \times Q \mapsto (\Sigma_\epsilon \times (\Theta \times \Omega \mapsto \mathit{Bool}) \times (\Theta \times \Omega \mapsto \Omega))^9$:*

$$A_{D,(q,\theta)}(\langle\rangle) \stackrel{\text{def}}{=} \langle\rangle$$

$$A_{D,(q,\theta)}(\langle e, \omega\rangle +\!\!+ ews) \stackrel{\text{def}}{=} \begin{cases} \langle e \mid c \mapsto a\rangle +\!\!+ A_{D,(q',a(\theta\ \omega))}(ews) & \text{if } \exists\, q' \cdot q \xrightarrow{e|c \mapsto a} q' \wedge \\ & \quad c(\theta, \omega) \\ \langle \epsilon \mid \mapsto\rangle +\!\!+ A_{D,(q',a(\theta\ \omega))}(ews) & \text{otherwise} \end{cases}$$

Then we define $A_D(ews) \stackrel{\text{def}}{=} A_{D,(q,\theta)}(ews)$. We overload the transitive closure of DEAs to ignore the transitions marked by an epsilon: $q \xrightarrow{\epsilon|c \mapsto a} q' \Rightarrow q = q'$.

Then we can show that a concrete trace can be associated with an abstract trace.

Lemma 4 *If a concrete trace reaches a certain explicit state and symbolic state configuration in the semantics, then there is a symbolic path from the initial state of the DEA with the counterpart abstract trace to the same explicit state: $\exists\, \theta, ews \cdot (q_0, \theta_0) \stackrel{ews}{\Longrightarrow} (q, \theta) \Leftrightarrow q_0 \xrightarrow{A(ews)} q$*

---

[7] Here c' can be the empty condition that always returns *true*.

[8] Note knowing $(q_1, e, c, a, q_1') \in \rightarrow_{D_1}$ and $(q_1, q_2) \xrightarrow{e|c \wedge c' \mapsto a;\ a'} (q_1', q_2')$ then we can infer than $(q_2, e, c', a', q_2') \in \rightarrow_{D_2}$.

[9] We use ϵ to mark a step where no symbolic transition applies, and *epsilon* $\mid\mapsto$ for $e \mid \mathit{true} \mapsto \mathit{skip}$

Proof

By induction on the length of the trace:

BC: $ews = \langle\rangle$
trivially then $(q_0, \theta_0) \stackrel{ews}{\Longrightarrow} (q_0, \theta_0)$, but $A(ews) = \langle\rangle$, and then also trivially then $(q_0, \theta_0) \stackrel{A(ews)}{\Longrightarrow} (q_0, \theta_0)$

IH: $(q_0, \theta_0) \stackrel{ews'}{\Longrightarrow} (q', \theta') \Leftrightarrow q_0 \stackrel{A(ews')}{\Longrightarrow} q'$

IC: To show: $(q_0, \theta_0) \stackrel{ews' +\!\!+ \langle e \rangle}{\Longrightarrow} (q, \theta) \Leftrightarrow q_0 \stackrel{(A(ews' +\!\!+ \langle e, \omega \rangle))}{\Longrightarrow} q$

Proof: $(q_0, \theta_0) \stackrel{ews' +\!\!+ \langle e, \omega \rangle}{\Longrightarrow} (q, \theta)$

$\Leftrightarrow$ (*by definition of* $\Rightarrow$)

$(q_0, \theta_0) \stackrel{ews'}{\Longrightarrow} (q', \theta') \wedge (q', \theta') \xrightarrow{\langle e, \omega \rangle} (q, \theta)$

$\Leftrightarrow$ (*by inductive hypothesis*)

$q_0 \stackrel{A(ews')}{\Longrightarrow} q' \wedge (q', \theta') \xrightarrow{\langle e, \omega \rangle} (q, \theta)$

$\Leftrightarrow$ (*by definition of the DEA semantics*)

Case 1: $\exists\, q' \xrightarrow{e|c \mapsto a} q \wedge c(\theta', \omega) \wedge q' \notin B$

$\Leftrightarrow$ (*by above case condition and semantics*)

$q_0 \stackrel{A(ews')}{\Longrightarrow} q' \wedge q' \xrightarrow{e|c \mapsto a} q$

$\Leftrightarrow$ (*by definition of* $\Rightarrow$)

$q_0 \stackrel{A(ews') +\!\!+ (e|c \mapsto a)}{\Longrightarrow} q$ □

Case 2: $\neg(\exists\, q' \xrightarrow{e|c \mapsto a} q \wedge c(\theta', \omega) \wedge q' \notin B)$

$\Leftrightarrow$ (*by above case condition and semantics*)

$q_0 \stackrel{A(ews')}{\Longrightarrow} q' \wedge q' \xrightarrow{\epsilon|\mapsto} q \wedge q = q'$

$\Leftrightarrow$ (*by definition of* $\Rightarrow$)

$q_0 \stackrel{A(ews') +\!\!+ (\epsilon|c \mapsto a)}{\Longrightarrow} q$ □

Using this notion of abstraction, that allows us to move between concrete and abstract transitioning, we can then characterize the language of the residual.

We can show that the residual accepts at least the violating part of D_1 that is not violating in D_2, and at most all the violating traces of D_1.

Lemma 5 *The violation language of the property residual includes all the violation traces of* D_1 *not violating in* D_2*, and a subset of the violating traces of* D_1*:* $V(D_1) \setminus V(D_2) \subseteq V(D_1 \setminus D_2) \subseteq V(D_1)$.

Proof

$$
\begin{array}{ll}
 & ews \in V(D_1) \setminus V(D_2) \\
\Rightarrow & (\textit{by definition of } V \textit{ and } \setminus) \\
 & \exists\, \theta_1, q_{B_1} \in B_1 \cdot (q_{0_1}, \theta_{0_1}) \stackrel{ews}{\Longrightarrow}_1 (q_{B_1}, \theta_1) \wedge \\
 & \quad \nexists \theta_2, q_{B_2} \in B_2 \cdot (q_{0_2}, \theta_{0_2}) \stackrel{ews}{\Longrightarrow}_2 (q_{B_2}, \theta_2) \\
\Rightarrow & (\textit{re-writing the second conjunct, and by Proposition. 2}) \\
 & \exists\, \theta_1, q_{B_1} \in B_1 \cdot (q_{0_1}, \theta_{0_1}) \stackrel{ews}{\Longrightarrow}_1 (q_{B_1}, \theta_1) \wedge \\
 & \quad \exists\, \theta_2, q_2 \in Q_2 \setminus B_2 \cdot (q_{0_2}, \theta_{0_2}) \stackrel{ews}{\Longrightarrow}_2 (q_2, \theta_2) \\
\Rightarrow & (\textit{abstracting using Lemma. 4}) \\
 & \exists\, q_{B_1} \in B_1 \cdot q_{0_1} \stackrel{A(ews)}{\Longrightarrow}_1 q_{B_1} \wedge \\
 & \quad \exists\, q_2 \in Q_2 \setminus B_2 \cdot q_{0_2} \stackrel{A(ews)}{\Longrightarrow}_2 q_2 \\
\Rightarrow & (\textit{by definition of synchronous composition}) \\
 & \exists\, q_{B_1} \in B_1, q_2 \in Q_2 \setminus B_2 \cdot (q_{0_1}, q_{0_2}) \stackrel{A(ews)}{\Longrightarrow}_{1||2} (q_{B_1}, q_2) \\
\Rightarrow & (\textit{by definition of } D_1 \setminus D_2) \\
 & \exists\, q_{B_1} \in B_1 \cdot q_{0_1} \stackrel{A(ews)}{\Longrightarrow}_{1\setminus 2} q_{B_1} \\
\Rightarrow & (\textit{concretizing using Lemma. 4}) \\
 & \exists\, \theta, q_{B_1} \in B_1 \cdot (q_{0_1}, \theta_{0_1}) \stackrel{ews}{\Longrightarrow}_{1\setminus 2} (q_{B_1}, \theta) \\
\Rightarrow & (\textit{by definition of } V) \\
 & ews \in V(D_1 \setminus D_2)
\end{array}
$$

Then we have proven that $V(D_1) \setminus V(D_2) \subseteq V(D_1 \setminus D_2)$. We are left to prove that $V(D_1 \setminus D_2) \subseteq V(D_1)$.

$$
\begin{array}{ll}
 & ews \in V(D_1 \setminus D_2) \\
\Rightarrow & (\textit{by definition of } V) \\
 & \exists\, \theta_1, q_{B_1} \in B_1 \cdot \quad (q_{0_1}, \theta_{0_1}) \stackrel{ews}{\Longrightarrow}_{D_1 \setminus D_2} (q_{B_1}, \theta_1) \\
\Rightarrow & (\textit{abstracting using Lemma. 4}) \\
 & \exists\, q_{B_1} \in B_1 \cdot \quad q_{0_1} \stackrel{A(ews)}{\Longrightarrow}_{D_1 \setminus D_2} q_{B_1} \\
\Rightarrow & (\textit{all transitions in } D_1 \setminus D_2 \textit{ are also in } D_1 \textit{ by its definition}) \\
 & \exists\, q_{B_1} \in B_1 \cdot \quad q_{0_1} \stackrel{A(ews)}{\Longrightarrow}_{D_1} q_{B_1} \\
\Rightarrow & (\textit{concretising using Lemma. 4}) \\
 & \exists\, \theta_1, q_{B_1} \in B_1 \cdot \quad (q_{0_1}, \theta_0) \stackrel{ews}{\Longrightarrow}_{D_1} (q_{B_1}, \theta) \\
\Rightarrow & (\textit{by definition of } V) \\
 & ews \in V(D_1)
\end{array}
$$

□

If this residual has no state that can reach a bad state then we can conclude that the second DEA is stricter than the first.

Proposition 3 *Given the residual $D_1 \setminus D_2$, if it only contains the initial state, then D_2 is stricter than D_1: $Q_{D_1 \setminus D_2} = \{q_{0_1}\} \Rightarrow D_2 \succeq_s D_1$*

Proof

$$
\begin{array}{ll}
 & Q_{D_1 \setminus D_2} = \{q_{0_1}\} \\
\Rightarrow & (\textit{by definition of } V \textit{ and since } q_{0_1} \textit{ is not a bad state}) \\
 & V(D_1 \setminus D_2) = \varnothing \\
\Rightarrow & (\textit{by Lemma.5 and standard set-theoretic results}) \\
 & V(D_1) \setminus V(D_2) = \varnothing \\
\Leftrightarrow & (\textit{by standard set-theoretic results}) \\
 & V(D_1) \subseteq V(D_2) \\
\Leftrightarrow & (\textit{by definition of strictness}) \\
 & D_2 \succeq_s D_1
\end{array}
$$

□

This is not a necessary condition (since we are abstracting away using symbolic transitions), and we may not be able to prove this strictness statically. Then we allow for the user to make the choice to leave the residual to prove (or enforce) at runtime. Monitoring for this residual is then enough for verifying D_1, if D_2 has been verified.

Theorem 4 *If SC is shown to be compliant with D_2 then monitoring for $D_1 \setminus D_2$ is equivalent to monitoring D_1, for SC: $SC \vdash_{RV} D_2 \Rightarrow (SC \vdash_{RV} D_1 \Leftrightarrow D_1 \setminus D_2)$.*

Proof

Assuming $SC \vdash_{RV} D_2$, and then $run(SC) \notin V(D_2)$, then:

$$
\begin{array}{ll}
 & SC \vdash_{RV} D_2 \\
\Leftrightarrow & (\textit{by definition of} \vdash_{RV}) \\
 & run(SC) \notin V(D_2) \\
\Leftrightarrow & (\textit{by standard set-theoretic results}) \\
 & run(SC) \in V(D_1) \setminus (D_2) \Leftrightarrow run(SC) \in V(D_1) \\
\Rightarrow & (\textit{by Lemma.5 and standard set-theoretic results}) \\
 & run(SC) \in V(D_1 \setminus D_2) \Leftrightarrow run(SC) \in V(D_1) \\
\Leftrightarrow & (\textit{by definition of} \vdash_{RV}) \\
 & SC \vdash_{RV} D_1 \setminus D_2 \Leftrightarrow SC \vdash_V D_1
\end{array}
$$

□

The proposed algorithm then consists of: (a) proving as much as possible of the expected model against the promised model, and leave the expected residual for runtime, and (b) verifying the promised model against the smart contract, statically and/or at runtime, as previously illustrated in Figure 1.

3 Evaluation

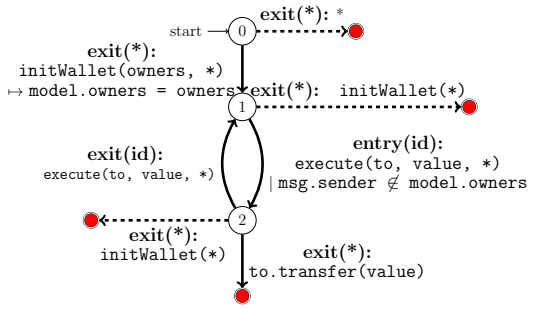
We have considered several case studies in investigating this approach. We then evaluated them in terms of the overheads that monitoring adds to their deployment, and the effects our residual analysis has. The case studies include: (i) a token wallet depending on a library to provide its logic; (ii) an electric car smart contract interacting with a battery swapping station in an internet of things scenario (a variant of the example in Section 2); and (iii) a courier service that interacts with a smart contract regulating a procurement contract [6], by notifying it upon delivery of some ordered goods[10].

3.1 Wallet Case Study

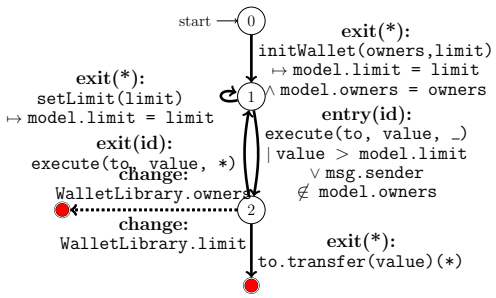
In Fig. 3(a) and Fig. 3(b) we specify an expected and promised model of a wallet library. Fig. 3(a) specifies that a wallet must be initialised with some owners and following that cannot be initialised again, while if a transaction is initiated (i.e. `execute` is called) then it is only successful if it is initiated by one of the owners. Fig. 3(b) similarly specifies that if a wallet is initialised with some owners and a transaction limit, then transactions initiated for values over the limit or by non-owners will fail (and that during a transaction the owners and limit cannot change). Clearly there is some intersection between the two, in fact if we consider the expected model in Fig. 3(a) without the dashed transitions then the promised model includes this smaller model's bad traces. With this reduced model our analysis would consider the promised model as stricter than the expected model and attempt to verify the promised model using a combination of static and dynamic analysis.

However, if we consider the full expected model with the dashed transitions then this is clearly not the case: the promised model does not prohibit the smart contract from being re-initialised (this was in fact a problem with a version of the Parity wallet, see [15]), while the full expected model does. If we consider that the user still wants to use the offered service (perhaps we cannot find a better alternative), we then enforce

[10]The case studies code can be found here: `https://github.com/shaunazzopardi/interactive-smart-contracts-case-study`.



(a) Wallet is required to be initialised only once, and only allows owners to send ether.



(b) A transaction below the limit, or not initiated by an owner fails, while owners and the limit cannot change.

Figure 3: Wallet case study.

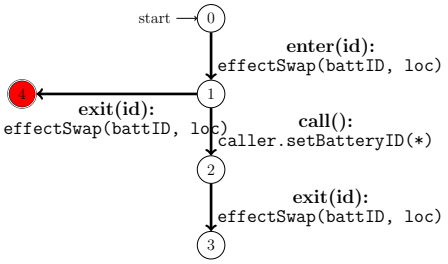
the parts of the expected model not prohibited by the promised model at runtime. Moreover we can statically analyse the promised model against the smart contract, and conclude that the dotted transitions in Fig. 3(b) will not affect the smart contract variables owners and limit (and thus can be ignored at runtime).

3.2 Battery Swapping

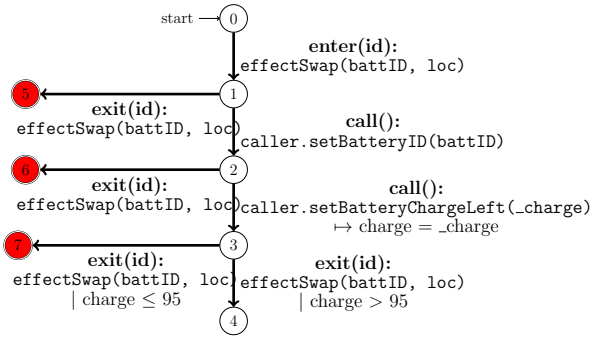
Although widely used to implement crypto-currencies, blockchains are being proposed for many other uses, among them to increase trust between devices in the Internet of Things (IoT) [1]. We consider a case study in this promising line of work.

As proposed in [17] blockchains provide a transparent and effective environment in which electric cars and battery stations can interact to facilitate charging and/or swapping of batteries at appropriate stations. Here we consider an implementation of such a battery swapping scenario in the form of a car smart contract that interacts with a station smart contract to query the availability of compatible batteries and to effect the battery swap.

In Fig. 4(a) we specify an expected model required by a car's smart contract, where it requires that upon the car calling the station's `effectSwap` function then the battery station calls in turn the car's `setBatteryID` function, setting the new battery's id in the car's smart contract. In Fig. 4(b) we can see that this is already ensured by the station, while moreover it ensures that the new battery's charge level is also sent to the car, while that this is also bigger than 95%. Note how if the transition between states 1 and 2 was not present in Fig. 4(b) then Fig. 4(a) would not be ensured, but by analysing the smart contract of the station Fig. 4(a) can still be verified statically.



(a) Effecting a swap calls the caller's (`msg.sender`) *setBatteryID* function.



(b) Effecting a swap sets the caller's new battery ID and its charge, while it is ensured that the new battery has more than 95% charge.

Figure 4: Example expected model and promised model for the battery swapping example.

Here the residual $EM \setminus PM$ is thus the trivially satisfied property, and we only need to verify PM at runtime[11].

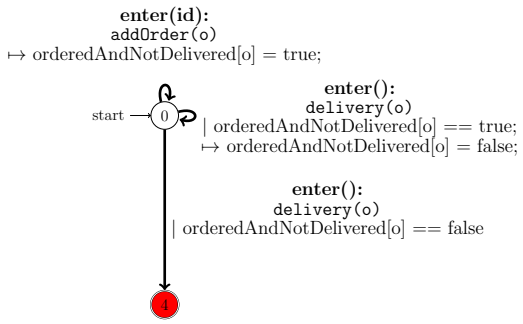
3.3 Courier Service

A promising use case for distributed ledgers is that of serving as a trusted record. Here we consider a case study where a buyer and a seller agree to use a procurement smart contract on the Ethereum blockchain as the environment where they interact.

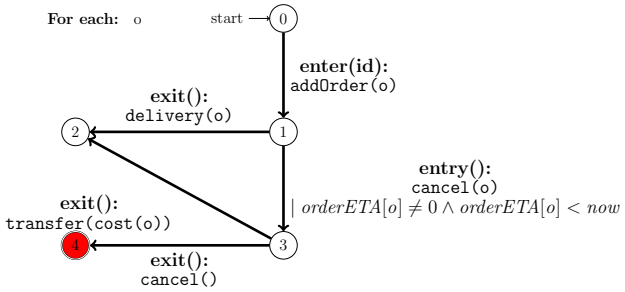
The buyer places orders by calling an `addOrder` function in the smart contract, with enough ether to pay for the cost of the order. Then off-chain the seller is obliged to deliver the order in the estimated time of arrival (ETA), upon which the money paid into the smart contract by the buyer is released to the seller. To automate this the seller engages the service of a courier to perform the delivery, such that the buyer signs for arrival of the goods, by calling some `signature` function in the courier's smart contract. In turn this function calls back into the procurement smart contract, calling the `delivery` function that releases the cost of the order to the seller.

Here we consider behaviour expected by the buyer and other behaviour promised by the seller. In 5(a) the buyer specifies that an order o is certified as delivered (i.e. the `delivery` is called) only for orders that have been created and not delivers. This avoids the courier service requesting the buyer to pay for an order twice, or to pay for

[11]Note, with some further analysis we could also collapse states 1, 2, and 3, since these are ensured by the code.



(a) An order can only be delivered if it has been ordered and if it has not been delivered before.



(b) An order can be canceled if it has not been delivered and if the ETA has elapsed, upon which the cost of the order is released to the buyer.

Figure 5: Courier service case study.

an order they did not create. In 5(b) the seller promises to allow the buyer to cancel an order and refund the buyer successfully if the order ETA has elapsed. We replicate this property for each order, by keeping hashmap of each order's monitor state.

Here we are able to prove the promised model, since we can show that the mapping *orderETA[o]* is initialised only when *addOrder(o)* is called, and *cancel(o)* is not successful if the ETA is less than the current time, after which a refund is always enacted. On the other hand the expected model cannot be proven, since no pre-conditions of the function prevent delivering an order twice. By monitoring and enforcing this expected model the user can ensure that this does not happen at runtime.

3.4 Overhead Measurements

We have evaluated this approach by considering the overheads added by monitoring for these behavioural models, and by testing the hypothesis that residual analysis can reduce these. In RV we usually consider overheads in terms of time and space overheads, but in this context these are neatly captured by the notion of *gas*. In Ethereum each bytecode instruction has a gas cost which is set *a priori* in the Ethereum specification (see [20]), intended to reflect the effort needed to execute the instruction. Any call to a smart contract function must then be also accompanied by some amount of ether. If enough ether is paid the transaction can be successful, otherwise the transaction is reverted once the amount of ether is exhausted.

We consider the unmonitored, naïvely monitored, and residual optimized monitored versions of both smart contracts, considering in gas their deployment cost. We do not

consider added transaction costs since it is not significant (around 5% at most) in our case studies. The results are shown in Table 11. This allows us to conclude that added monitoring costs are proportionally significant, however their cost in real-world value is negligible. Moreover, residual analysis can significantly reduce the overheads associated with monitoring in the presence of significantly overlapping specifications (consider the battery swapping case study) or where the code reflects the specification clearly (see the courier service case study). In the worst case scenario overheads are marginally reduced (in the wallet case study). Moreover, being able to prove some of the specification statically provides more confidence in the service provided and the expenses incurred in monitoring the rest can be reasonable depending on the risk involved (e.g. the cost in ensuring that the wallet behaves correctly at runtime is negligible if it is protecting a large amount of ether we have stored in the wallet).

Table 11: Deployment overheads for our case studies.

	Without Monitoring		Naïvely Monitored		With Residuals	
Case Study	Gas	Percent	Gas	Percent	Gas	Percent
Wallet	2735384	100%	4704167	172%	4640699	170%
Battery Swapping	1049522	100%	2546832	243%	2100367	200%
Courier Service	1141676	100%	2058229	180%	1720458	151%

4 Discussion and Future Work

This work continues on our research into how to combine different analysis techniques, where we defined and explored what residuals are in [2], and in [3] we applied this to define residuals symbolic automata for Java programs. Here we extended this work by defining residuals between symbolic automata, enabling another variety of combination between model checking, static analysis and runtime verification.

Interestingly here we are using assume-guarantee reasoning across different verification methods, instead of across components as usual — by assuming the promised model holds at runtime we use it as our base abstraction of the smart contract pre-deployment and we attempt to provide pre-deployment guarantees in the form of the expected model. By also using residuals we allow for sliding between static analysis and runtime verification.

Other approaches (e.g. [11]) work at the level of bytecode, however we are working at the level of Solidity code here, which allows for better treatment of high level concepts, such as loops and calls, and for easier debugging of the violating smart contracts. We

are currently in the process of implementing a tool to perform the analysis presented in this paper[12], with a simple control-flow analysis with a view to add data-flow analyses and theorem provers to allow for more precise strictness checking.

4.1 Trust and Monitorability

A question that arises from the framework is how to implement it in such a way that the user can trust the verification performed. The strictness check and static analysis steps can be performed off-chain by the user themselves, avoiding any need to trust the service provider. However, an issue for monitoring is that the user does not control the service provider smart contract, while current instrumenting and monitoring implementations for monitoring on the Ethereum blockchain, e.g. [6], require ownership of a smart contract so that the monitoring logic can be inlined. Here we consider this problem of monitorability.

We can characterise models to be of two mutually exclusive kinds: (i) DEAs with all events observable by the user's smart contract; and (ii) DEAs with some events observable only by the service. The models previously considered in Figure 2 are of the second type since the user cannot detect every time the `swap` function is entered. However, if the user adds conditions to the `swap` transitions limiting them to trigger only upon the caller of `swap` being the user themselves, then the model becomes of the first type because the user can observe such events (a caller can always observe themselves).

Models of the first kind are then monitorable from the user's smart contract and then do not require any trust in the service provider. These suffice to specify simple but useful (no) re-entrancy properties. However, more sophisticated properties, e.g. dynamically checking whether some data flows from the service to another third-party smart contract, require observing events in the service. Here instead of assuming monitoring logic has been inlined in the service we can instead assume that event triggering code has been instrumented in the service and that monitor smart contracts can register as listeners for these events. This limits models to runtime observable events to those instrumented in the already existing service. Moreover, given the availability of the service on the blockchain it can be inspected by the user or automated services to ensure this instrumentation has been done in a correct manner and that any listening monitors will return a correct result.

Care has to be taken in event instrumentation in the presence of `delegatecall`s, since these may perform actions on the state but from logic contained in another smart contract, e.g. events matching variable changes cannot be observed synchronously

[12]See `https://github.com/shaunazzopardi/solidity-static-analysis` for progress.

in the presence of `delegatecalls`. Simple Hoare logic type pre- and post-condition contracts, which can be specified using DEAs, are an alternative to more complex control-flow specification that still can be soundly and completely monitored for in the presence of delegation.

5 Related Work

Fröwis et. al. in [8] survey Ethereum smart contracts for the use of calls, for the purpose of identifying which smart contracts can be trusted a priori and which not. Smart contracts not calling other smart contracts are termed *control-flow immutable*. They find that two out of five smart contracts use these calls, motivating the need for analysis to increase trust *a priori*.

Several tools for both static and dynamic analysis in Ethereum exist, with varying levels of precision and automation. Briefly, a symbolic execution engine of EVM bytecode is available, Oyente [14], while KEVM provides an executable semantics of EVM bytecode allowing for deductive verification [11], and EtherTrust a static analysis tool based on an abstract semantics of EVM bytecode [10]. This work however has to assume non-determinism at call sites. Here instead we are assuming behavioural contracts that can be used to make static analysis more precise. Behavioural or session types have been used before to specify behavioural contracts objects must respect [13], our approach instead is automata-based.

On the dynamic side, an ad hoc approach has been used for verifying business processes [19], while a general runtime verification engine has also been developed [6]. This engine inlines checks in the smart contract, allowing a stake-based design pattern such that any party violating the contract pays a stake into the smart contract, that can be used as a fine for any misbehaviour. This technique can be used to implement the envisioned reparations in case of a violation in our approach.

Monitoring has previously been proposed to verify the assumptions for assume-guarantee reasoning. [12] proposes the combination of model checking with runtime verification, such that given an assumption about the system, if we can show that it implies the guarantee then we can just monitor for the assumption at runtime. Other work discharges assumptions on the statically unfixed environment at runtime [16]. Differently, here we also attempt to prove the assumption on the program statically, leaving only a residual for runtime, and similarly for a guarantee. This notion of a residual operator corresponds to quotient operators, which have been used to identify parts of specification that have not yet been implemented [18].

6 Conclusions

Ethereum smart contracts present challenges for verification, in particular given their behaviour at runtime can change through calling different smart contracts. Here we have proposed a framework to manage this by associating a model of behaviour with such a call. A behavioural model would be used by the caller to set bounds on the behaviour of the called smart contract. The service providers promise certain behaviour in a similar model, which can be used by the user to find an appropriate service. The user can use the formal methods presented here to verify the appropriateness of the service against its smart contract statically and at runtime, while verifying or enforcing their own expected behaviour.

From a formal perspective, this method allows us to limit what can happen at smart contract call sites, which we plan to exploit in future work for more complete verification of whole smart contracts, rather than focusing only on functional contracts.

References

[1] Arefayne Abadi, F., Ellul, J., Azzopardi, G.: The blockchain of things, beyond bitcoin: A systematic review. In: Blockchain for the Internet of Things (07 2018)

[2] Azzopardi, S., Colombo, C., Pace, G.: A model-based approach to combining static and dynamic verification techniques. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques. pp. 416–430. Springer International Publishing, Cham (2016)

[3] Azzopardi, S., Colombo, C., Pace, G.J.: Control-flow residual analysis for symbolic automata. In: Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017, Torino, Italy, 19 September 2017. pp. 29–43 (2017). https://doi.org/10.4204/EPTCS.254.3, `https://doi.org/10.4204/EPTCS.254.3`

[4] Azzopardi, S., Ellul, J., Pace, G.J.: Monitoring smart contracts: CONTRACTLARVA and open challenges beyond. In: The 18th International Conference on Runtime Verification (2018)

[5] Colombo, C., Ellul, J., Pace, G.J.: Contracts over smart contracts: Recovering from violations dynamically. In: 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018), LNCS (2018)

[6] Ellul, J., Pace, G.: contractLarva. https://github.com/gordonpace/contractLarva (2018), [Online; accessed 02-March-2018]

[7] Francalanza, A., Aceto, L., Achilleos, A., Attard, D.P., Cassar, I., Monica, D.D., Ingólfsdóttir, A.: A foundation for runtime monitoring. In: Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings. pp. 8–29 (2017). https://doi.org/10.1007/978-3-319-67531-2_2, https://doi.org/10.1007/978-3-319-67531-2_2

[8] Fröwis, M., Böhme, R.: In code we trust? In: Garcia-Alfaro, J., Navarro-Arribas, G., Hartenstein, H., Herrera-Joancomartí, J. (eds.) Data Privacy Management, Cryptocurrencies and Blockchain Technology. pp. 357–372. Springer International Publishing, Cham (2017)

[9] Gheorghiu Bobaru, M., Păsăreanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Gupta, A., Malik, S. (eds.) Computer Aided Verification. pp. 135–148. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

[10] Grishchenko, I., Maffei, M., Schneidewind, C.: Foundations and tools for the static analysis of ethereum smart contracts. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 51–78. Springer International Publishing, Cham (2018)

[11] Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Rosu, G.: Kevm: A complete semantics of the ethereum virtual machine (2017), http://hdl.handle.net/2142/97207

[12] Hinrichs, T.L., Sistla, A.P., Zuck, L.D.: Model check what you can, runtime verify the rest. In: HOWARD-60: A Festschrift on the Occasion of Howard Barringer's 60th Birthday. pp. 234–244 (2014), http://www.easychair.org/publications/?page=1000880740

[13] Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P.M., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (Apr 2016). https://doi.org/10.1145/2873052, http://doi.acm.org/10.1145/2873052

[14] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. CCS '16, ACM, New York, NY,

USA (2016). https://doi.org/10.1145/2976749.2978309, http://doi.acm.org/10.1145/2976749.2978309

[15] Palladino, S.: https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7/, [Online; accessed 23-March-2018]

[16] Sokolsky, O., Zhang, T., Lee, I., McDougall, M.: Monitoring assumptions in assume-guarantee contracts. In: Aceto, L., Francalanza, A., Ingólfsdóttir, A. (eds.) PrePost@IFM. EPTCS, vol. 208, pp. 46–53 (2016), http://dblp.uni-trier.de/db/series/eptcs/eptcs208.html#SokolskyZLM16

[17] Sun, H., Hua, S., Zhou, E., Pi, B., Sun, J., Yamashita, K.: Using ethereum blockchain in internet of things: A solution for electric vehicle battery refueling. In: Chen, S., Wang, H., Zhang, L.J. (eds.) Blockchain – ICBC 2018. pp. 3–17. Springer International Publishing, Cham (2018)

[18] Verdier, G., Raclet, J.B.: Quotient of acceptance specifications under reachability constraints. In: Dediu, A.H., Formenti, E., Martín-Vide, C., Truthe, B. (eds.) Language and Automata Theory and Applications. pp. 299–311. Springer International Publishing, Cham (2015)

[19] Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., Mendling, J.: Untrusted Business Process Monitoring and Execution Using Blockchain, pp. 329–347. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-45348-4_19, https://doi.org/10.1007/978-3-319-45348-4_19

[20] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151**, 1–32 (2014)

A Appendix

A.1 Language of Smart Contracts

We define the semantics of a smart contract in the context of blockchain, where a blockchain is a map from addresses to smart contracts.

Definition 14 *A blockchain state is a function from addresses to smart contracts:* $\mathbb{B} : \texttt{Addr} \mapsto SC$. *We use* $B(a)$ *to refer to the smart contract at address* a *in* B*, and* $B[a \mapsto_\Omega \omega]$ *to refer to a blockchain state that is the same as* B *except where the symbolic state at address* a *is* ω*.*

An execution on the blockchain then is an execution of some function call at some address in a blockchain state, that produces a new blockchain state. Configurations here will be similar to DEA configurations, also containing a smart contract state and a blockchain state, however the presence of function calls requires further constructs.

We first require a notion of when an explicit call state has been entered (i.e. the respective call has been executed) and when the call has finished. We add this by tagging explicit states with upwards or downwards arrows, i.e. $q^\downarrow$ denotes a call state that has yet to be entered, and $q^\uparrow$ denotes a state that is not a call state or a call state that has already been entered.

Usually we can make the assumption that an executed statement will only affect the symbolic state of the program it is in (unless it is a function call). This is true here for normal calls, however Ethereum has the notion of `delegatecall`s which behave differently — delegate calls can be used to execute the called function with the symbolic state of its caller. This requires us to add an address label that is used to point to the smart contract whose symbolic state is affected by execution. We call this address the *execution focus*. Moreover delegate calls have an effect only if the called contract's storage is compatible with the caller's storage[13]. We do not go into the detail of this here, but we simply assume the existence of a relation, denoted by $\gg: \Omega \times \Omega$, that relates symbolic states such that $\omega \gg \omega'$ means a smart contract with state ω can be delegated work to by smart contract with state ω'. For example a smart contract with state `uint x;` can delegate work to smart contracts with state `uint x; int y;`, however the second type cannot delegate work to the second.

We label transitions in the semantics by smart contract event-state traces instead of single event-state pairs, since calls may produce traces of events. We then frequently

[13]If the storage is seen as a list of variable declarations, then the caller's storage must be a prefix of the called's storage.

use the *flatten* function that flattens traces of traces into a single trace (we overload this latter also for traces of traces of traces). Transitioning here ends when a configuration with a final or revert state is reached, while we assume the events relating to a single address are being observed.

Definition 15 (Smart Contract Semantics) *The operational semantics of smart contracts is given in terms of configurations of type* $Config = (Q \times \{\uparrow, \downarrow\}) \times (\mathbb{B} \times \texttt{Addr})$, *and with a labeled transition relation* $\rightarrow$: $Addr \times Config \times (\Sigma \times \Omega)^* \times Config$. *For configurations* $c, c' \in Config$, *address* $a_I \in Addr$, *and trace* $ews \in ((\Sigma \times \Omega)^*)^*$ *we write* $c \xrightarrow[a_I]{ews} c'$ *for* $(a_I, c, ews, c') \in \rightarrow$. *We assume the same instrumented address* a_I *throughout and leave it out when clear from the context. We use* $\Rightarrow$ *as the transitive closure of this relation. We then define the operational semantics as follows, where* $allCalls = call \cup dcall$:

(1) *Given a configuration* $(s_1^{\downarrow}, (B, a))$, *suppose the smart contract contains a transition* $s_1 \xrightarrow{e|c} s_2$, *where the condition holds on the symbolic state at* a *in* B $(\omega_{B(a)})$, s_1 *is not a final state, then: (i) the smart contract state at* a *can be updated by the application of the statement at* s_1, *and (ii) the explicit state transitions to* $s_2^{\downarrow}$ *if* s_2 *is not a call or delegate call state, and to* $s_2^{\uparrow}$ *is a call state. The event on the transition is only recorded in the semantics if* a *is the instrumented address.*

$$\frac{s_1 \xrightarrow{e|c} s_1 \qquad c(\omega_{B(a)}) \qquad \omega = stmt(s)(\omega_{B(a)}) \qquad s_1 \notin F \qquad B' = (B[a \mapsto_\Omega \omega], a)}{\begin{array}{ll} s_2 \notin dom(allCalls) \wedge a = a_I \Rightarrow & (s_1^{\downarrow}, (B, a)) \xrightarrow{\langle e,\omega \rangle} (s_2^{\downarrow}, B') \\ s_2 \notin dom(allCalls) \wedge a \neq a_I \Rightarrow & (s_1^{\downarrow}, (B, a)) \xrightarrow{\langle\rangle} (s_2^{\downarrow}, B') \\ s_2 \in dom(allCalls) \wedge a = a_I \Rightarrow & (s_1^{\downarrow}, (B, a)) \xrightarrow{\langle e,\omega \rangle} (s_2^{\uparrow}, B') \\ s_2 \in dom(allCalls) \wedge a \neq a_I \Rightarrow & (s_1^{\downarrow}, (B, a)) \xrightarrow{\langle\rangle} (s_2^{\uparrow}, B') \end{array}}$$

(2) *Given a configuration* $(s^{\uparrow}, (B, a'))$, *suppose call state* s *is associated with function* f *at address* a, *then if calling this function ends in a finite-number of steps in a final state and with blockchain state* B', *then* $(s^{\uparrow}, (B, a'))$ *transitions to a configuration* $(s^{\downarrow}, (B', a'))$, *where the state is marked as already called with* $\downarrow$ *and the blockchain state is updated. The event trace resulting from the successful call is flattened and recorded between the start and end call states.*

$$\frac{a'.f = calls(q)(\omega_{B(a)}) \qquad (s_0', (B, a')) \xRightarrow{ewss} \exists\, B', s_F \in F_{a'.f} \cdot (s_F^{\downarrow}, (B', a'))}{(s^{\uparrow}, (B, a)) \xrightarrow{flatten(ewss)} (s^{\downarrow}, (B', a))}$$

(3) *Given a configuration $(s^{\uparrow}, (B, a'))$, suppose call state s is associated with function f at address a, then if calling this function ends in a finite-number of steps in a revert state, then $(s^{\uparrow}, (B, a))$ transitions to a configuration $(s^{\downarrow}, (B, a))$, where the state is marked as already called with $\downarrow$, but the blockchain state remains the same. The event trace resulting from the call is not recorded since the execution ends in a revert state.*

$$\frac{a'.f = calls(s)(\omega_{B(a)}) \qquad (s'_0, (B, a')) \overset{ewss}{\Longrightarrow} \exists\, B', s_R \in R_{a'.f} \cdot (s_R^{\downarrow}, (B', a'))}{(s^{\uparrow}, (B, a)) \xrightarrow{\langle\rangle} (s^{\downarrow}, (B, a))}$$

(4) *Given a configuration $(s^{\uparrow}, (B, a))$, suppose delegate call state s is associated with function f at address a', and the state of a' is compatible with that of a, then if calling this function ends in a finite-number of steps in a final state with blockchain state B', then $(s^{\uparrow}, (B, a))$ transitions to configuration $(s^{\downarrow}, (B', a))$, where the state is marked as already called with $\downarrow$ and the blockchain state is updated. The event trace resulting from the successful delegate call is flattened and recorded between the start and end delegate call states.*

$$\frac{a'.f = dcalls(s)(\omega_{B(a)}) \quad \omega_{a'} \gg \omega_a \quad (s'_0, (B, a)) \overset{ewss}{\Longrightarrow} \exists\, B', s_F \in F_{a'.f} \cdot (s_F^{\downarrow}, (B', a))}{(s^{\uparrow}, (B, a)) \xrightarrow{flatten(ewss)} (s^{\downarrow}, (B', a))}$$

(5) *Given a configuration $(s^{\uparrow}, (B, a))$, suppose delegate call state s is associated with function f at address a', and the state of a' is compatible with that of a, then if calling this function ends in a finite-number of steps in a revert state, then $(s^{\uparrow}, (B, a))$ transitions to the revert configuration $(s_R, (B', a))$ ($s_R \in R$ always can be found since R is defined to be non-empty).*

$$\frac{a'.f = dcalls(s)(\omega_{B(a)}) \quad \omega_{a'} \gg \omega_a \quad (s'_0, (B, a)) \overset{ewss}{\Longrightarrow} \exists\, B', s_R \in R_{a'.f} \cdot (s_R^{\downarrow}, (B', a))}{s_R \in R \Rightarrow \qquad (s^{\uparrow}, (B, a)) \xrightarrow{flatten(ewss)} (s_R, (B, a))}$$

(6) *Given a configuration $(s^{\uparrow}, (B, a))$, suppose a call or delegate call state s is associated with function f at address a', and the state of a' is not compatible with that of a, then $(s^{\uparrow}, (B, a))$ transitions to configuration $(s^{\downarrow}, (B, a))$, where the state is marked as already called with $\downarrow$ and the blockchain state remains the same.*

$$\frac{a'.f = allCalls(s)(\omega_{B(a)}) \quad \omega_{a'} \not\gg \omega_a}{(s^{\uparrow}, (B, a)) \xrightarrow{\langle\rangle} (s^{\downarrow}, (B, a))}$$

With this small-step semantics we can then identify successful blockchain executions,

and we define the language of a smart contract in terms of the trace induced by any possible sequence of executions from blockchain states containing the smart contract in the address being observed.

Definition 16 *We define a single* successful blockchain execution*, with a_I as the address under observation, as a function call: $B \xrightarrow[a_I]{ewss}\!\!\!\!\!\rightarrow B' \Leftrightarrow \exists\, a \in \texttt{Addr}, f \in fs_a, s_F \in F_{f_a} \cdot (s_{0_{f_a}}, (B, a)) \xRightarrow{ewss} (s_F, (B', a))$. We use $\Rightarrow\!\!\!\!\!\rightarrow$: Config $\times$ $(((\Sigma \times \Omega)^*)^*)^* \times$ Config as its transitive closure.*

We define the language of a smart contract *in terms of successful executions, given any initial blockchain state with the smart contract deployed at some address: $L(SC) = \{ews \in (\Sigma \times \Omega)^* \mid \exists\, a_I, B, B' \cdot B(a_I) = SC \wedge B \xrightarrow[a_I]{ewsss}\!\!\!\!\!\rightarrow B' \wedge ews = flatten(ewsss)\}$.*

A.2 Abstract Language Over-approximates language of Smart Contracts

We can show that the event trace projection of any smart contract trace is contained in its abstract language. First we define this projection.

Definition 17 *An event-state trace $ews \in (\Sigma \times \Omega)^*$ is projected into the event trace constructed by ignoring the symbolic states in ews: $T(\langle\rangle) \stackrel{\text{def}}{=} \langle\rangle$, and $T(\langle (e, \omega) \rangle \texttt{++} rest) \stackrel{\text{def}}{=} \langle e \rangle \texttt{++} T(rest)$.*

We overload this for sets of traces such that for $Ews \in (\Sigma \times \Omega)^*$ then $T(Ews) = \{es \in \Sigma^* \mid \exists\, ews \in Ews \cdot es = T(ews)\}$.

A property of this projection that follows easily from this definition is that given a trace and a set that contains it, the the projection of the trace is contained in the projection of the set.

Proposition 4 *A smart contract trace $ews \in (\Sigma \times \Omega)^*$ is in a set of traces $Ews \subseteq (\Sigma \times \Omega)^*$ then the event projection of ews is in the event projection of Ews: $ews \in Ews \Rightarrow T(ews) \in T(Ews)$*

Proof

This follows immediately from Defn.17.

Then we want to prove that the event projection of the language of the smart contract is contained in its abstract language. We do this by specifying an abstraction function that abstracts configurations into abstract states, and showing that if some event trace happens between two concrete traces then the event projection of this trace happens between their respective abstract states.

Definition 18 *A configuration* $(s_{\overline{f_a}}, (B, a'))$*, where* $\_ \in \{\uparrow, \downarrow\}$*, is abstracted to state* s_f *if and only if* $s_{\overline{f_a}}$ *is a state in the instrumented address, and the execution is on the same address, while the configuration is abstracted to* s_0 *otherwise:*

$$\alpha((s_{\overline{f_a}}, (B, a')) +\!\!+ rest) \stackrel{\text{def}}{=} \begin{cases} s_f & a = a' = a_I \\ s_0 & otherwise \end{cases}$$

We call a configuration under observation *if its abstraction is not* s_0*.*

The motivation behind this state abstraction is that configurations referring to states under observation have direct counterparts in the smart contract abstraction. Other states are associated with s_0, but since they are not under observation then they will only introduce traces at a call site which must match the trace of a sequence of a function call in the smart contract under observation. Since s_0 is both the initial and final state in the abstraction then transitioning from s_0 to s_0 with such a trace should be possible.

We are then trying to prove that the $\Rightarrow$ relation is a simulation of the $\rightarrow$ relation. We start first by showing this to be true for transitions between configurations under observation, and continue by showing it for other configurations.

Proposition 5 *A single-step transition in the semantics, between configurations under observation and that eventually reach a final state, with some smart contract trace ews is reflected in the abstraction as a transition between the respective abstract states of the configurations, labeled by the event projection of ews:* $(cc \xrightarrow{ews} cc') \wedge (\alpha(cc) \neq s_0 \wedge \alpha(cc') \neq s_0) \wedge \exists\, s_F \in F, B, a \cdot cc \Rightarrow (s_F, (B, a)) \Rightarrow \alpha(cc) \overset{T(ews)}{\rightsquigarrow} \alpha(cc')$.

Proof

We proceed by case analysis over whether the abstract states are equal or not:

Case 1:	$\alpha(cc) = \alpha(cc')$
$\Rightarrow$	(*definition of* α *and the semantics* $\Rightarrow$ *rules 2,3,4,5,6 where used*)
	$\alpha(cc) \in dom(allCalls)$
$\Rightarrow$	(*by definition of the abstraction and* $\twoheadrightarrow$)
	$\forall\, es \cdot \alpha(cc) \overset{es}{\twoheadrightarrow} \alpha(cc)$
$\Rightarrow$	(*by case condition and specialising for* $T(ews)$)
	$\alpha(cc) \overset{T(ews)}{\twoheadrightarrow} \alpha(cc')$
Case 2:	$\alpha(cc) \neq \alpha(cc')$
$\Rightarrow$	(*by definition of* α *and the semantics* $\Rightarrow$ *rule 1 was used*)
	$\alpha(cc) \xrightarrow{e\mid c} \alpha(cc') \wedge \exists\, \omega \cdot ews = \langle e, \omega \rangle$
$\Rightarrow$	(*by definition of the abstraction*)
	$\alpha(cc) \xrightarrow{e} \alpha(cc') \wedge \exists\, \omega \cdot ews = \langle e, \omega \rangle$
$\Rightarrow$	(*by definition of* $\twoheadrightarrow$)
	$\alpha(cc) \overset{T(ews)}{\twoheadrightarrow} \alpha(cc')$

We can extend this result to big-steps in the smart contract.

Proposition 6 *A big-step transition in a smart contract under observation, between states that eventually reach a final state, is reflected in its abstraction:* $(cc \overset{ewss}{\Longrightarrow} cc') \wedge (\alpha(cc) \neq s_0 \wedge \alpha(cc') \neq s_0) \wedge \exists\, s_F \in F, B, a \cdot cc \Rightarrow (s_F, (B, a)) \Rightarrow \alpha(cc) \overset{T(flatten(ews))}{\twoheadrightarrow} \alpha(cc')$.

Proof

This follows immediately by induction of the size of the trace, and applying Proposition. 5.

We prove this also for smart contracts not under observation, since they may still perform calls to the smart contract under observation.

Proposition 7 *A single-step transition in the semantics with some smart contract trace ews, between configurations not under observation and that eventually reach a final state, is reflected in the abstraction as a transition between the respective abstract states of the configurations, labeled by the event projection of ews:* $(cc \xrightarrow{ews} cc') \wedge (\alpha(cc) = \alpha(cc') = s_0) \wedge \exists\, s_F \in F, B, a \cdot cc \Rightarrow (s_F, (B, a)) \Rightarrow \alpha(cc) \overset{T(ews)}{\rightsquigarrow} \alpha(cc')$.

Proof

We proceed by case analysis on whether the trace is empty or not:

Case 1:	$ews = \langle\rangle$
$\Rightarrow$	(*by definition of* $\twoheadrightarrow$)
	$s_0 \overset{\langle\rangle}{\rightsquigarrow} s_0$
$\Rightarrow$	(*since* $\alpha(cc) = \alpha(cc') = s_0$ *and by the case condition*)
	$\alpha(cc) \overset{es}{\rightsquigarrow} \alpha(cc')$
Case 2:	$ews \neq \langle\rangle$
$\Rightarrow$	The semantics imply rule 2 was used, since others either always produce empty traces.
	W.l.g. this reduces to showing that any trace induced from a start state to an end state of an execution under observation occurs between s_0 and s_0 in the abstraction. This is ensured by Proposition. 6 and by rules 1 and 3 of the coarse abstraction.

We can then use these results to show that the coarse abstraction actually over-approximates the event projection of the language of the smart contract.

Theorem 5 *The language of the smart contract SC without state is in the coarsely abstracted language:* $T(L(SC)) \subseteq AL(SC)$.

Proof

$$
\begin{array}{ll}
 & es \in T(L(SC)) \\
\Rightarrow & (\textit{by definition of the event projection}) \\
 & \exists\, ews \in L(SC) \cdot es = T(ews) \\
\Rightarrow & (\textit{by definition of } L(SC)) \\
 & \exists\, ewsss \in (((\Sigma \times \Omega)^*)^*)^*, B', a_I \cdot B \xRightarrow[a_I]{ewsss} B' \wedge ews = \mathit{flatten}(ewsss) \\
\Rightarrow & (\textit{by definition of } \Rightarrow\!\!\!\!\!\Rightarrow \textit{ and Proposition. 5 and Proposition. 7}) \\
 & \exists\, ewsss \in (((\Sigma \times \Omega)^*)^*)^*, B', a_I \cdot s_0 \overset{T(ewsss)}{\rightsquigarrow} s_0 \wedge ews = \mathit{flatten}(ewsss) \\
\Rightarrow & (\textit{by definition of } AL(SC)) \\
 & T(ews) \in AL(SC)
\end{array}
$$

The results in this paper apply for any abstraction that satisfies this theorem.

We can re-formulate this theorem at a lower level.

Corollary 1 *If a trace is in the language of the smart contract then in the coarse abstraction of the smart contract for any prefix of the trace there is a state reachable by the event projection of the trace:* $ews \in L(SC) \Rightarrow \forall\, ews' \in \mathit{prefixes}(ews) \cdot \exists\, s \cdot s_0 \overset{T(ews')}{\rightsquigarrow} s$

Proof

This follows immediately from Thm.5.