A Technique for Automata-Based Verification with Residual Reasoning

Shaun Azzopardi[1] [a], Christian Colombo[1] and Gordon Pace[1]

[1]*Department of Computer Science, Faculty of ICT, University of Malta, Msida, Malta*
{*shaun.azzopardi, christian.colombo, gordon.pace*}*@um.edu.mt*

Abstract: Analysing programs at a high-level of abstraction reduces the effort required for verification, but may abstract away details required for full verification of a specification. Working at a lower level, e.g. through model checking or runtime verifying program code, can avoid this problem of abstraction, at the expense of much larger resource requirements. To reduce the resources required by verification, analysis techniques at decreasing levels of abstraction can be combined in a complementary manner through *partial verification* or *residual analysis*, where any useful partial information discovered at a high-level is used to reduce the verification problem, leaving an easier residual problem for lower-level analyses. Our contribution in this paper is a technology-agnostic symbolic-automata-based framework to project verification effort onto different verification stages. Properties and programs are both represented as symbolic automata, with an event-based view of verification. We give correctness conditions for residual analysis based on equivalence with respect to verification of the original problem. Furthermore we present an intraprocedural residual analysis to identify parts of the property respected by the program, and parts of the program that cannot violate the property.

1   INTRODUCTION

Verification techniques are used to attempt to prove that a program satisfies or violates a certain specification. Such an attempt may fail to give a verdict when not enough resources have been allocated, or when the analysis ignores pertinent details necessary for the problem at hand. Traditionally in the face of such a failure the only option is to attempt a different kind of verification technique on the same problem. Often however failing verification attempts can be used to reduce the original problem, leaving a smaller (and hopefully easier) problem for subsequent attempts [Dwyer and Purandare, 2008].

Techniques for this kind of *partial verification* fall largely into two categories: *program-based* and *property-based*. Program-based techniques identify parts of the program proven safe, leaving a smaller part of the program to be explored by subsequent techniques. Property-based techniques instead transform the property that the program should be compliant with, encoding certain aspects of the specific program being verified to reduce the problem.

These two categories are dual to each other. Consider that a model checker may exhaust all the resources available before traversing all the program state space. One can then take a program-based approach and slice the verified partial state space away, leaving a reduced program exhibiting only the possibly unsafe behaviour [Beyer et al., 2018, Lal et al., 2007]. Dually, a property-based approach can predicate the original property by a condition that only holds true for program states that have not yet been verified safe [Beyer et al., 2012]. This work exists largely for *state-based* specification formalisms that can specify predicates about the acceptable concrete variable state of the program.

Another popular approach for verification, especially for runtime verification [Falcone et al., 2018], abstracts program state changes in terms of events. *Event-based* specifications specify properties in terms of program actions (e.g. program function calls or state modification), with verification involving event instrumentation of the program and properties compared in some manner with this instrumented program to determine compliance. Program-based approaches in this context remove instrumentation from proven safe parts of the program. CLARA is one such approach [Bodden and Lam, 2010, Azzopardi et al., 2020], analysing Java source code with finite-state properties using control-flow analysis (ignoring data aspects of the program), with extensions for symbolic automata [Azzopardi et al., 2017]. Property-based ap-

proaches for event-based formalisms are limited, focused on Java and involving only summarisation of deterministic parts of the program [Dwyer and Purandare, 2007], or refining of pre- and post-conditions of events [Ahrendt et al., 2012].

We can then identify gaps in literature with regards to existing event-based approaches to partial verification: (i) they are largely technology-specific; and (ii) no approaches combine both control-flow and data-oriented aspects of a program. In this paper we seek to remedy these gaps by a technology-agnostic approach that encodes both control-flow and data-oriented aspects of the program in a symbolic automata-based model. We give appropriate conditions for reductions of verification problems in this framework, and present three such novel complementary intraprocedural residual analyses. We evaluate the approach in a Solidity smart contract case study by measuring the reduction of resources required by a subsequent runtime verification phase.

In Section 2 we present a technology-agnostic framework for verification with forms of symbolic automata acting both as models for programs and properties. In Section 3 we discuss correctness conditions for residual analysis in the presented framework, and present three novel residual operations illustrated through examples. In Section 4 we present an evaluation of these approaches, while discussing the presented results in Section 5. We further consider related work in Section 6 and conclude in Section 7.

2 PROGRAM VERIFICATION

Verification is the process of checking that a program satisfies a property. Here we are interested in *event-based* verification techniques, where program behaviour and properties can both be represented as a set of event traces, and verification being the process of checking that the program traces are a subset of the property traces. However working directly with programs represented by sets of traces, as in previous work [Azzopardi et al., 2017], ignores both the program structure and data-based decisions, which directly determine the sequences of events produced. Here we attempt to remedy this.

In this section we present our formalisms for programs and properties, both being similar forms of symbolic automata, with program automata producing events that drive property automata. As a running example we use the Solidity (a language designed to write smart contracts for the Ethereum blockchain) smart contract in Listing. 1, which implements an interface for an auction house. We also assume an event

```
1  contract SmartAuctionHouse{
2    address owner;
3    uint goings = 0;
4    bool ongoing;
5    mapping(int => bool) finished;
6    mapping(int => address payable) winner;
7    mapping(int => uint) offers;
8    int id;
9
10   constructor() public {
11     owner = msg.sender;
12   }
13   modifier isOwner(){
14     require(owner == msg.sender);
15     -;
16   }
17   function start(int itemID, uint startOffer)
           public isOwner{
18     require(!ongoing);
19     require(!finished[id]);
20     offers[itemID] = startOffer;
21     id = itemID;
22     ongoing = true;
23   }
24   function offer() public payable{
25     require(ongoing);
26
27     if(offers[id] < msg.value){
28       if(winner[id] != address(0)){
29         winner[id].transfer(offers[id]);
30       }
31       winner[id] = msg.sender;
32       offers[id] = msg.value;
33     }
34   }
35   function declare() public isOwner{
36     require(ongoing);
37     require(goings >= 3);
38     finished[id] = true;
39     ongoing = false;
40     id = -1;
41     goings = 0;
42   }
43   function going() public isOwner{
44     require(ongoing);
45     goings++;
46   }
47 }
```

Listing 1: Auction smart contract implementation.

set Σ, in this paper containing two kinds of program events: (i) events wrapping around a function call, either before (e.g. **before**(function)) or after (e.g. **after**(function)); and (ii) we use the notation **@**var to match after a variables modification (we also allow reference to the previous value of a variable through var$_{\textbf{pre}}$). We also assume Σ contains an empty event ε.

2.1 Control-flow Automata

A common representation for programs is the corresponding control-flow graph, with transitions representing execution of statements or conditional branching. Based on this notion, here we define *control-flow automata* (CFAs). To represent statement execution, we tag transitions by a statement (st : STMT), while to represent conditions on the state (required to represent `if-then-else` and looping constructs) we guard these by a condition (c : Cond). Here we are interested in event-based verification, which assumes the program emits events that are processed by the monitor, thus we let transitions trigger an event (e : Σ). A transition is then of the form $s \xrightarrow{c \triangleright st \blacktriangleright e} s'$.

To keep the semantics technology-agnostic we assume a set of symbolic states Ω, with conditions as predicates over Ω and statements as transformations of Ω. Moreover, consider that a well-designed program is not often a single monolithic piece but consists of different methods[1]. To model this we allow concrete CFA states to be *call states*, which encapsulate other CFAs, allowing for a form of hierarchy. We characterise this formally.

Definition 2.1. *A control-flow automaton (CFA) is parametrised by a type of symbolic program states Ω; and a finite set of concrete program states S.*

*A CFA, of type **CFA**, is a tuple $\langle s_0, E, \text{calls}, \rightarrow \rangle$, where:*

(a) *$s_0 \in S$ is the initial concrete program state;*

(b) *$E \subseteq S$ is the set of end states, we use s_E for an element of this set;*

(c) *$\text{calls} : S \nrightarrow (\Omega \rightarrow \textbf{CFA})$ identifies states associated with dynamic method calls; and*

(d) *$\rightarrow : S \times \text{Cond} \times \text{STMT} \times \Sigma \rightarrow S$ is the deterministic transition relation (where $\text{Cond} = \Omega \rightarrow Bool$, and $\text{STMT} = \Omega \rightarrow \Omega$).*

We write $s \xrightarrow{c \triangleright st \blacktriangleright e} s'$ for $(s, c, st, e, s') \in \rightarrow$. We use P or M for CFAs. We use the function $\text{methods} : \textbf{CFA} \rightarrow 2^{\textbf{CFA}}$ for the set of methods transitively called by M[2].

Figure. 2 illustrates CFAs corresponding to methods in Listing. 1. In these examples we use several shorthands: (i) $*$ is used in place of the *otherwise*[3] transition; and (ii) we leave out the *true* condition, the identity statement, and ε (and their markers).

An interesting aspect of Listing. 1 is that it does not have a main method but instead simply offers a set

---

[1] We use the term *methods* to refer to program functions, to distinguish them from the formal object.

[2] In this paper we will be limiting our analysis to when this is finite.

[3] Corresponding to the transition with the negation of the disjunction of the alternate transitions' conditions.
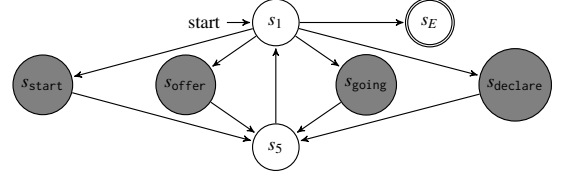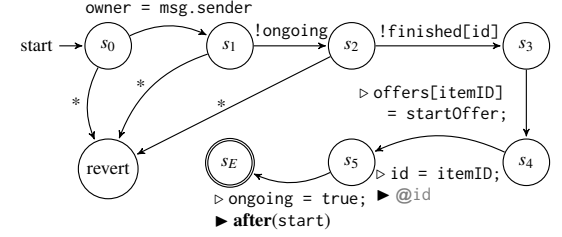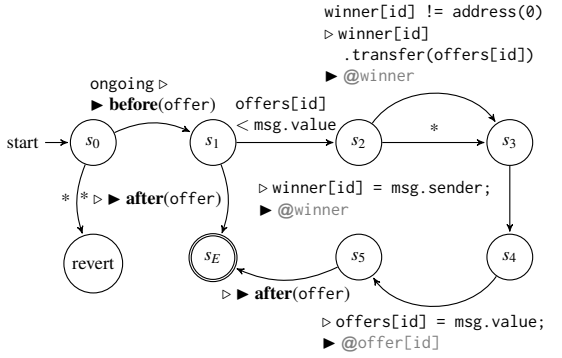

Figure 1: Automata modelling the interface of the smart contract, with shaded states marking call states.

(a) `start` method.

(b) `offer` method.

Figure 2: CFAs of methods from Listing. 1 and the underlying interface[4].

of methods that can be called in any sequence, while a CFA expects a single entry-point in the program. Consider that such a single entry-point can be modeled with a CFA based on the automaton in Figure. 1, with appropriate transition labels that allow the interface methods to be called in any sequence with arbitrary inputs depending on the initial symbolic state.

To be able to reason about CFAs we give the formalism an operational semantics that produces the execution traces associated with the CFA. This semantics traverses a CFA given an initial symbolic state. This symbolic state evolves based on the available transitions, producing a symbolic event pair from $\Sigma \times \Omega$. When at call states there are two different behaviours. If an end state can be reached in the called method then control is yielded back to the call-

---

[4] Here events in gray will be removed by applying the residual techniques presented in Section 3.

ing CFA, encoding the produced execution trace on a transition. If on the other hand the called method cannot reach an end state (with the input symbolic state) control is yielded back with no recorded events. For example, looking at Figure. 2a note how a `revert` state is not an end state (and cannot reach an end state), and thus any trace ending at this state is not recorded as an execution trace of the program.

To model method calls, states are extended with a *direction* ($S^{\uparrow\downarrow} \stackrel{\text{def}}{=} S \times \{\uparrow,\downarrow\}$): $s^{\uparrow}$ denotes *exiting* from a call of s, and $s^{\downarrow}$ denotes *entering* into a call of s.

Transitions in the semantics are tagged by traces containing pairs of events and symbolic states (to be consumed by the monitor). For simplicity we abuse notation and assume the transitive closure of the operational semantics transition is also tagged by these traces, rather than traces of traces, i.e. $\rightarrow, \Rightarrow : (S^{\uparrow\downarrow} \times \Omega) \times (\Sigma \times \Omega)^* \times (S^{\uparrow\downarrow} \times \Omega)$, with this corresponding to the standard transitive closure with concatenation of the traces.

Definition 2.2. *The* intraprocedural operational semantics *of a CFA is given with configurations as a pairs of directed states and symbolic states ($S^{\uparrow\downarrow} \times \Omega$), transitions labeled by traces of pairs of events and symbolic states ($(\Sigma \times \Omega)^*$), and characterised by:*

(i) *Given a transition $s_1 \xrightarrow{c\triangleright st \blacktriangleright e} s_2$, with c holding on ω, and s_2 not being an end state, then a configuration $(s_1^{\uparrow},\omega)$ transitions to $(s_2^{\downarrow}, st(\omega))$ with $\langle e, st(\omega)\rangle$:*

$$\frac{s_1 \xrightarrow{c\triangleright st \blacktriangleright e} s_2 \qquad s_2 \notin E \qquad c(\omega)}{(s_1^{\uparrow},\omega) \xrightarrow{\langle e,st(\omega)\rangle} (s_2^{\downarrow}, st(\omega))}$$

(ii) *If s_1 is not a call state, then a configuration with a state $s_1^{\downarrow}$ simply transitions to a configuration with the same symbolic state and the exiting state $s_1^{\uparrow}$:*

$$\frac{s_1 \notin dom(calls)}{(s_1^{\downarrow},\omega) \xrightarrow{\langle\rangle} (s_1^{\uparrow},\omega)}$$

(iii) *If s_1 is a call state, then a configuration $(s_1^{\downarrow},\omega)$ transitions with ews to a configuration $(s_1^{\uparrow},\omega')$ if the method called by s_1 reaches an end configuration with ω' when starting with ω:*

$$\frac{s_1 \in dom(calls) \qquad M = calls(s_1)(\omega)}{\exists s_E \in E_M, \omega' \in \Omega \cdot (s_{0_M},\omega) \stackrel{ews}{\Rightarrow} (s_E,\omega')}{(s_1^{\downarrow},\omega) \xrightarrow{ews} (s_1^{\uparrow},\omega')}$$

(iv) *If s_1 is a call state, then a configuration $(s_1^{\downarrow},\omega)$ transitions with $\langle\rangle$ to a configuration $(s_1^{\uparrow},\omega)$ if the method called by s_1 reaches no end configuration when starting with ω:*

$$\frac{s_1 \in dom(calls) \qquad M = calls(s_1)(\omega)}{\nexists s_E \in E_M, \omega' \in \Omega \cdot (s_{0_M},\omega) \Rightarrow (s_E,\omega')}{(s_1^{\downarrow},\omega) \xrightarrow{\langle\rangle} (s_1^{\uparrow},\omega')}$$

From this operational semantics we can characterise the execution behaviour of a CFA by identifying the prefixes induced by an initial symbolic state.

Definition 2.3. *The behaviour of CFA P starting with symbolic state ω of length $i \in \mathbb{N}$ is the trace $t_{P,\omega,i}$ defined formally as follows[5]:*

$$t_{P,\omega,i} \stackrel{\text{def}}{=} \begin{cases} pre(ews,i) & (s_{0_P}^{\downarrow},\omega) \stackrel{ews}{\Rightarrow} (s_{E_P}^{\uparrow},\omega') \wedge i < len(ews) \\ \langle\rangle & otherwise \end{cases}$$

We use CFAs to represent the actual behaviour of a program, in fact note how in CFAs there is a direct one-to-one correlation with states and lines in code. Its low-level nature however makes it unviable for high-level specification. In the next section, we consider a more appropriate variant for specification.

2.2 Dynamic Event Automata

Common formalisms for property specification largely fall into logic-based or automata-based. Here we focus on *dynamic event automata* (DEAs), expressive symbolic automata encoding allowed behaviour both through control-flow and data-oriented aspects, allowing for more succinct representations and power than the more traditional finite-state automata. DEAs are similar to CFAs, with symbolic states (Θ), and transitions tagged by events (e), guards (g) on both the program and property symbolic states, and actions (a) on the property symbolic state, written $q \xrightarrow{e|g \mapsto a} q'$.

Definition 2.4. *A* dynamic event automaton *(DEA) is parametrised by: (i) a type of symbolic monitor states Θ; (ii) a type of symbolic program states Ω; and (iii) a finite set of concrete states Q.*

A DEA is then a tuple $\pi \stackrel{\text{def}}{=} \langle q_0, \theta_0, B, A, \rightarrow \rangle$, where:

(a) *$q_0 \in Q$ is the initial concrete monitor state;*

(b) *$\theta_0 \in \Theta$ is the initial monitor symbolic state;*

(c) *$B, A \subseteq Q$ are respectively the set of concrete bad and accepting states, we use q_B for a bad state; and*

(d) *$\rightarrow: Q \times (\Sigma \setminus \{\varepsilon\}) \times Guard \times Act \rightarrow Q$ is the set of transitions (where $Guard = \Theta \times \Omega \rightarrow Bool$, and $Act = \Theta \times \Omega \rightarrow \Theta$).*

We write $q \xrightarrow{e|g \mapsto a} q'$ for $(q,e,g,a,q') \in \rightarrow$.

Figure. 3 illustrates some DEAs for Listing. 1.

Monitors at runtime evolve depending on the behaviour of the program. Here the behaviour of the program at runtime is encoded as a trace of pairs of events and program symbolic states. Then we define

---

[5]Where $pre : (\Sigma \times \Omega)^* \times \mathbb{N}$ is the prefix of the input trace of the input length if existent, otherwise the empty trace, and $len : (\Sigma \times \Omega)^* \rightarrow \mathbb{N}$ gives the length of a trace.
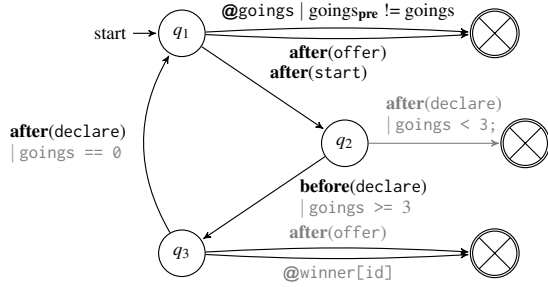
(a) *The business process of an auction smart contract: (i) no offers can be made before an auction is started; (ii) during an auction the winner cannot be declared before going thrice; (iii) no offer can be made and no winner set just as the auction is declared; and (iv) after the auction has been declared the goings reset.*
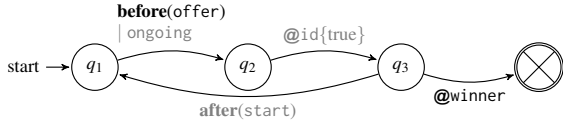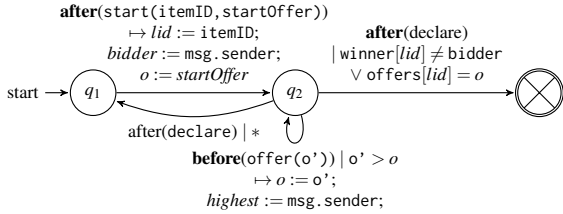

(b) *When an offer is made during an ongoing action and the item being auctioned is changed, no winner is set until the auction starts.*

(c) *Whenever the current winning offer for an item is modified, the new value must be larger than the old value.*

(d) *Keep track of the current highest bid. This should be the winning bid at the end of an auction.*

Figure 3: Several properties expected of the auction smart contract in Listing. 1, with bad states marked with a cross.

the operational semantics of DEAs to allow DEAs to evolve based on these symbolic events. A transition is taken if its event matches the event triggered while the guard holds on the symbolic states. While it transforms the monitoring symbolic state with the action accordingly. The semantics halts once a verdict (a bad or accepting state) has been reached.

Definition 2.5. *The operational semantics of a DEA is given with configurations of type $Q \times \Theta$, with transitions labeled by elements of $\Sigma \times \Omega$, and characterised by the following rules:*

(i) *A configuration (q, θ), given a transition $q \xrightarrow{e|g \mapsto a} q'$, evolves to a configuration $(q', a(\omega, \theta))$ only if the guard g holds on θ:*

$$\frac{q \xrightarrow{e|g \mapsto a} q' \qquad q \notin A \cup B \qquad g(\omega, \theta)}{(q, \theta) \xrightarrow{e, \omega} (q', a(\omega, \theta))}$$

(ii) *if the previous rule does not hold, then the configuration does not evolve:*

$$\frac{\nexists q', \theta' \cdot \theta' \neq \theta \land (q, \theta) \xrightarrow{e, \omega} (q', \theta')}{(q, \theta) \xrightarrow{e, \omega} (q, \theta)}$$

We overload $\Rightarrow$ for the transitive closure of $\rightarrow$.

We use DEAs to explicitly specify the behaviour that the program is prohibited from exhibiting, characterised as the set of traces reaching a bad state.

Definition 2.6. *The* bad traces *of property π are those traces that reach a bad state in π: $B(\pi) \stackrel{\text{def}}{=} \{t \in (\Sigma) \times \Omega \mid \exists q_B \in B, \theta \in \Theta \cdot (q_0, \theta_0) \stackrel{t}{\Rightarrow} (q_B, \theta)\}$.*

Looking at the operational semantics consider that states unreachable from the initial state cannot participate in a violation. Moreover states that cannot reach a bad state can be marked as accepting, since when reaching such states any continuation cannot violate. We can reduce a DEA accordingly.

Definition 2.7. *A DEA is in* optimal form *if: (i) there are no states unreachable from the initial state: $\nexists q \in Q \cdot q_0 \not\Rightarrow q$; and (ii) states that cannot reach a bad state are accepting and have no outgoing transitions: $\forall q \in Q, q_B \in B \cdot q \not\Rightarrow q_B \implies (q \in A \land \nexists q' \in Q \cdot q \Rightarrow q' \land q \neq q')$.*

We assume every DEA used here is optimal.

Finally, we can characterise compliance of a program with a property when every execution prefix exhibited at runtime is not in the property bad language.

Definition 2.8. *A CFA P is said to satisfy a property π if every execution prefix it generates is not a bad trace of π: $P \vdash \pi \stackrel{\text{def}}{=} \forall \omega \in \Omega, i \in \mathbb{N} \cdot t_{P, \omega, i} \notin B(\pi)$.*

Based on this background, we introduce our framework for partial verification of event-based properties in the next section. Our objective is to prove $P \vdash \pi$ or to reduce it to another problem $P' \vdash \pi'$.

3 RESIDUAL STATIC ANALYSIS

Exploring the whole runtime state space of a program exhaustively is expensive for large enough programs. An alternative is to perform *sound static analysis*, by over-approximating the behaviour of a program and verifying this sound abstraction rather than the actual

program behaviour. We can then conclude satisfaction when no violating prefix is found in the abstraction. However the lack of completeness means that locating a violating prefix does not allow us to conclude that the program is violating, leaving an inconclusive result. This motivates the need for partial verification techniques where instead, inconclusive results are framed as a residual version of the original problem.

Here we discuss and give correctness conditions for reducing the verification problem, and then present an intraprocedural program-property analysis combining both control- and data-flow abstractions upon which we define three novel residual analyses that soundly: (i) silence CFA events; (ii) remove DEA transitions; and (iii) remove DEA transition guards. The full proofs of the claims in this section can be found in a corresponding technical report [Azzopardi et al., 2019].

3.1 Residual Correctness

Reducing the verification problem involves transformations of the program instrumentation and of the property. Care must be taken to ensure that these transformations do not cause a change in verdict to the program, i.e. that they are equivalent. There are different levels to which this can be ensured.

The most basic condition that is required of a residual analysis is to simply ensure that the original verification problem $(P \vdash \pi)$ is equivalent to the residual verification problem $(P' \vdash \pi')$. However this may be considered too weak. Consider that we may be interested in runtime verifying the residual problem, where each trace is analysed individually. Consider that P may violate π with multiple traces, and then a sufficient P' is one that exhibits only one of the original violating traces and a sufficient π' is π. This means that there is some input with which P violates π but with which P' does not violate π', although $P \vdash \pi$ is still equivalent to $P' \vdash \pi'$.

Instead we can characterise a deeper level of equivalence, one that exists at the level of inputs. This ensures that at runtime an execution trace of P will violate iff the corresponding execution trace of P' will violate. However, consider that P' may be transformed such that its execution trace violates later than that of P. In applications such as enforcement, this behaviour would not be ideal since detecting a violation as early as possible is ideal. To ensure that P and P' violate at the same time, we consider also prefixes of the CFAs of the same length. We term this equivalence at the level of input and prefix length *lockstep verification equivalence*.

Definition 3.1. *(P, π) is said to be in* lockstep *with the pair (P', π') iff execution traces of P and P' from the same symbolic state ω are given the same verdict by π and π' for every prefix length i:*
$$\forall \omega \in \Omega, i \in \mathbb{N} \cdot t_{P,\omega,i} \in B(\pi) \iff t_{P',\omega,i} \in B(\pi').$$

Our objective is then to create residual analyses that produce a verification problem that is in lockstep with the original, ensuring detection of a violating trace occurs at the same point. Note how this leaves us open to detect non-violating traces before the original monitor.

3.2 Intraprocedural Analysis

The problem of verification here is the checking that the language of the program is not contained in the bad language of the property, as defined in Definition. 2.8. A standard manner of checking this is to analyse the composition of the program with the property [Vardi, 2007]. In our case CFAs may contain calls to other CFAs, making composition non-straightforward. Moreover, for complex programs, dealing with interprocedural behaviour (i.e. the way data- and control- flows through multiple procedures) can prove to be expensive. Instead the analyses we present here are *intraprocedural*, considering the behaviour of each method individually, while over-approximating outside behaviour. Working at this level allows for less resource-intensive analysis. Another aspect necessary for this analysis is that execution traces depend on the manner data flows through a CFA, which is determined by the transitions and their labels. A viable analysis then does not only take into account the control-flow structure of a CFA, but also the conditions that determine data-flow, and the statements that determine how data is transformed.

In this section we present both data- and control-flow abstractions of a program. We first characterise data abstractions of runtime symbolic states using conditions, and then define an intraprocedural composition of CFAs and DATEs that takes into account a data abstraction to avoid impossible runtime paths. This composition will be the basis upon which we define our residuals.

3.3 Data Abstraction

In an attempt to maintain a light analysis, we shall not be enumerating all the possible runtime configurations of a CFA. Instead, we shall be abstracting these configurations by the corresponding concrete program state (e.g. state s abstracts runtime configuration (s, ω)). To determine which transitions from s can actually be taken at runtime, we wish to associate

State	Data Abstraction
s_0	`true`
s_1	`ongoing`
s_2	`ongoing & offers[id] < msg.value`
s_3	`ongoing & offers[id] < msg.value`
s_4	`ongoing & offers[id] < msg.value` `& winner[id] == msg.sender`
s_5	`ongoing & offers[id] == msg.value` `& winner[id] == msg.sender`
s_E	`ongoing`

Table 1: A data abstraction for the `offer` CFA in Figure. 2b.

with s some information about the symbolic states it can be associated with at runtime. We characterise such a *data abstraction* as a function that associates a condition with a state that holds true on every corresponding symbolic state at runtime. We keep this intraprocedural.

Definition 3.2. *A function* $\sim: S \to$ Cond *is said to be a data abstraction of a CFA if for any explicit state s, it associates with s a condition that is always true on symbolic states associated with configurations of s that are reachable from the initial configuration and reaching an end configuration:* $\forall \omega_0, \omega, \omega_E \in \Omega, s^- \in S^{\uparrow\downarrow} \cdot (s_0^\downarrow, \omega_0) \Rightarrow (s^-, \omega) \Rightarrow (s_E^\uparrow, \omega_E) \wedge s \sim c \implies c(\omega).$

A data abstraction here essentially is a *state invariant* relation. Consider that a trivial abstraction is one that associates *true* with every state. Another more useful abstraction that we use in a prototype of this work is created by intraprocedurally propagating statement effects. For example, if a state s is only accessible through a transition that performs a statement $x = 7$ then $s \sim (x == 7)$ is an appropriate abstraction for s. See Table. 1 for this logic applied to `offer` (Figure. 2b).

To utilise these abstractions during our analysis, we can employ an *SMT solver* (specifically the Z_3 solver [de Moura and Bjørner, 2008]) that can be exploited to check whether a DEA guard is compatible ($\top$), or not ($\bot$) with a condition. We assume this is sound, i.e. it may be inconclusive, since certain background theories for symbolic states may be undecidable.

Definition 3.3. *A satisfaction procedure sat :* $Guard \times$ Cond $\to \{\top, \bot, ?\}$ *is a procedure that satisfies the following conditions:*

(i) **Compatibility** *If the procedure returns $\top$ then whenever the condition is true on a symbolic state, the guard is also true on the same state:*
$sat(g,c) = \top \implies (\forall \omega \in \Omega, \theta \in \Theta \cdot c(\omega) \implies g(\theta, \omega));$ *and*

(ii) **Incompatibility** *If the procedure returns $\bot$, then whenever the condition is true on a symbolic state, the guard is false on the same state:*

$sat(g,c) = \bot \implies (\forall \omega \in \Omega, \theta \in \Theta \cdot c(\omega) \implies \neg g(\theta, \omega)).$

For example, we can easily conclude that for any guard y, $sat(\text{ongoing} \vee y, \text{ongoing}) = \top$ for any y, while for a guard y independent of the CFA symbolic state, $sat(y, c) = ?.$

Furthermore consider that a DEA transition $q \xrightarrow{e|g \mapsto a} q'$ may only execute after a CFA transition $s \xrightarrow{c \triangleright st \blacktriangleright e} s'$. To check whether the DEA transition may be triggered, we have the option of checking whether g is incompatible with the data abstraction of s'. However the data abstraction of s' may be very weak, since s' may have multiple incoming transitions (consider s_E in Table. 1). Instead, since here we know which transition is being taken, we can compute a stronger abstraction, by updating the abstraction of s with the transition taken, i.e. with the condition c and statement st.

Definition 3.4. *A procedure that computes the data abstraction of a state with respect to a certain transition, update :* $S \times$ Cond $\times$ STMT $\to$ Cond, *obeys the condition that the produced predicate holds at runtime on the symbolic state after the transition is taken:* $\forall \omega, \omega_E \in \Omega \cdot (s_0^\downarrow, \omega_0) \Rightarrow (s^\uparrow, \omega) \Rightarrow (s_E^\uparrow, \omega_E) \wedge c(\omega) \implies update(s,c,st)(st(\omega)).$

Consider that a valid value for update(s_5, `true`, `skip`) is the abstraction of s_5. This allows us to consider the effect of taking the transition from s_5 to s_E (see Figure. 2b), rather than using the weaker abstraction associated with s_E (see Table. 1).

We shall be using this data abstraction of states to identify on-the-fly incompatible transitions in a CFA-DEA composition, as presented in the next section.

3.3.1 Control-flow Abstraction

Since we have limited ourselves by intraprocedurality, in the interest of tractability, we cannot simply unfold P and compose it against the property. Instead our approach is based on the intuition that if an execution trace of P is violating, then the violation must occur in some method M of P. Our analysis then will involve attempting to find such an M, by composing each method against the property, while abstracting the behaviour before and after the method is called, and during calls of the method.

The composition will give us a view of every possible execution of P that contains an execution of M. To achieve this we can abstract the behaviour before the method is called chaotically, allowing any event to occur before the initial state of M. This models any possible call to M from any method. With similar justification, we can do the same for end and call states.

Consider however that a method M may contain special events that only it can trigger, e.g. only

the start method triggers the **before**(start) event. Then, a more precise abstraction only allows initial, end, and call states to behave chaotically with events that may occur outside of M, while allowing end and call states to re-enter M with artificial transitions to its initial state. This re-entering will allow us to make more precise inferences about execution traces of the program.

We then proceed by comparing transitions of M with those of π such that: (i) any available DEA transition with the same event and compatible guard is taken synchronously (a *concrete match*); and (ii) if the guards are not mutually exclusive with the abstracted CFA state, or if the transition's event is ε, then the configuration evolves asynchronously in M. This reflects the CFA and DEA operational semantics.

Definition 3.5. *The* abstract intraprocedural composition *of a CFA M and a property π is the transition system with states of type $S \times Q$, transitions labeled by pairs of CFA and DEA transition labels, with possibly one label missing represented by the $\square$ symbol:* $((\mathsf{Cond} \times \mathsf{STMT} \times \Sigma) \cup \{\square\}) \times ((\Sigma \times Guard \times Action) \cup \{\square\})$, *and characterised by the following rules:*

1. *Initial and end configurations take any available DEA transition with events used outside of M[6], while end and call configurations may transition into M and back:*

$$\frac{s \in \{s_0\} \cup E \quad e \in out(M)}{q \xrightarrow{e|g \mapsto a} q'} \qquad \frac{s \in E \cup dom(\mathsf{calls})}{q, q' \in Q}$$
$$\frac{}{(s_0, q) \xrightarrow[e|g \mapsto a]{\square} (s_0, q')} \qquad \frac{}{(s, q) \xrightarrow[\square]{\square} (s_0, q)}$$
$$(s_E, q') \xrightarrow[\square]{\square} (s, q')$$

2. *(s, q) transitions to (s', q') if there are transitions between the respective states in the CFA and DEA, with the same event, and if the abstraction of s updated with the CFA transition is not incompatible with the DEA guard:*

$$\frac{s \xrightarrow{c \triangleright st \blacktriangleright e} s' \quad q \xrightarrow{e|g \mapsto a} q' \quad sat(update(s, c, st), g) \neq \bot}{(s, q) \xrightarrow[e|g \mapsto a]{c \triangleright st \blacktriangleright e} (s', q')}$$

3. *(s, q) transitions to (s', q) if there are transitions between the respective CFA states, and if the data abstraction of s updated with the CFA transition is not incompatible with the conjunction of the negation of each guard associated with a transition from q:*

$$\frac{s \xrightarrow{c \triangleright st \blacktriangleright e} s' \quad sat(update(s, c, st), \bigwedge_{q \xrightarrow{e|g' \mapsto a} q'} \neg g') \neq \bot}{(s, q) \xrightarrow[\square]{c \triangleright st \blacktriangleright e} (s', q)}$$

---

[6]*out* : **CFA** $\to 2^\Sigma$ is the function that returns the set of events used shallowly in CFAs' of P that are not the input CFA.

4. *Any configuration with a program state with an outgoing ε-transition asynchronously transitions with that CFA transition:*

$$\frac{s \xrightarrow{c \triangleright st \blacktriangleright \varepsilon} s'}{(s, q) \xrightarrow[\square]{c \triangleright st \blacktriangleright \varepsilon} (s', q)}$$

We overload $\Rightarrow$ as its transitive closure. We call a transition that does not use $\square$ a concrete *transition, and use $\boxslash$ for a label that is not $\square$.*

Note how we are using $\square$ as a placeholder value whenever either a DEA transition does not match or a CFA transition is not used.

As described before, we want to use this composition to identify when DEA transitions can be used on the way to a violation. We will do this by iterating over all concrete transitions of a composition, and gathering those whose source configurations is reachable from both the initial configuration, and directly from a configuration that can reach a bad end configuration. This will allow to choose all those DEA transitions that may be used on the way to a violation, i.e. those DEA transitions that may both be used as part of a violating trace and those transitions that may be used to avoid a violating verdict.

Definition 3.6. *We write $(s, q) \xrightarrow[y]{x} (s', q')$, when:*

(i) the transition is in the composition:

$$(s, q) \xrightarrow[y]{x} (s', q'); \text{ and}$$

(ii) (s, q) is reachable from the initial configuration and can reach a bad end configuration:

$$(s_0, q_0) \Rightarrow (s, q) \Rightarrow (s_E, q_B).$$

Note how we require (s, q) to reach a bad end configuration and not (s', q') since q' may be effectively accepting and then $(s, q) \xrightarrow[y]{x} (s', q')$ would be used on the way to an accepting verdict.

Then we claim that if we cannot find a method of a program P for which there is no such useful transition, we can conclude that P satisfies the DEA in question.

Theorem 3.1. $\forall \pi \in \Pi \cdot \nexists M \in methods(P), s, s' \in S_M, q_B \in B, q \in Q \setminus B \cdot (s, q) \xrightarrow[\boxslash]{\boxslash} (s', q_B) \implies P \vdash \pi.$

Proof Sketch. Synchronous steps of the CFA P and DEA π at runtime have corresponding transitions in a composition of π with a method of P. This allows to conclude that if at runtime there is a point at which the program is not violating and takes a step that causes a violation, then this step must be reflected in concrete transition of some composition. $\square$

This is a strong requirement for satisfaction, given the chaotic abstraction we are using, and often will fail. Instead we can produce residuals.

3.3.2 Residual Analysis

The characterising features of residual problems are twofold, they are: (i) reductions of the original problem; and (ii) in lockstep with the original problem. The first is required to ensure that the residual problem requires the same or less work than the original problem, rather than increasing the workload. To ensure the second, we must be careful in the way we choose sub-structures of a DEA to remove. Our principle here will be that we only remove objects that cannot affect the verdict of a given trace. We will present three residuals that obey this principle, all relying on the notion of a useful transition.

Looking at Theorem. 3.1, consider that any property transitions not participating in a useful transition are either unused or used when an accepting verdict has already been given. This ensures that removing them will not cause a runtime verdict to change.

Definition 3.7 (Control-Flow Residual). $\pi \backslash P$ *is the property with the transitions of π participating in a useful transition in a method of P:*

$$\rightarrow_{\pi \backslash P} \overset{\text{def}}{=} \left\{ q \xrightarrow{e|g \mapsto a} q' \mid \begin{array}{c} \exists M \in methods(P), s, s' \in S_M \cdot \\ (s,q) \xrightarrow[e|g \mapsto a]{\boxslash} (s',q') \end{array} \right\}.$$

Consider for example that in a composition of `start` (Figure. 2a) and Figure. 3b a bad state is never reached — a change in `id` is always followed by a **before**(`start`) event that avoids reaching the bad state. Moreover, since these events only occur in this method, the respective transitions can be removed from Figure. 3b. Note how the limitation for end states in the first composition rule is essential to prove this.

Dually, we can re-formulate this for CFA transitions. Consider that any CFA transition not used in a useful transition can be silenced (i.e. transformed into ε-transitions), since the events they trigger are only triggered when a verdict can already be given.

Definition 3.8. $P \backslash \pi$ *is the program:*

(i) with transitions of P participating in a concrete transition:

$$\rightarrow_0 \overset{\text{def}}{=} \left\{ s \xrightarrow{c \triangleright st \blacktriangleright e} s' \mid \begin{array}{c} \exists M \in methods(P), s, s' \in S_M \cdot \\ (s,q) \xrightarrow[\boxslash]{c \triangleright st \blacktriangleright e} (s',q') \end{array} \right\};$$

and

(ii) with the rest of the transitions of P silenced:

$$\rightarrow_1 \overset{\text{def}}{=} \left\{ s \xrightarrow{c \triangleright st \blacktriangleright \varepsilon} s' \mid (s,c,st,e,s') \in \rightarrow_\pi \backslash \rightarrow_0 \right\}.$$

Then $\rightarrow_{P \backslash \pi} \overset{\text{def}}{=} \rightarrow_0 \cup \rightarrow_1$. The reduction applies transitively to calls: $calls_{P \backslash \pi} = \lambda s, \omega \cdot calls(s)(\omega) \backslash \pi$.

Consider that in Figure. 2a we can turn off the `id` modification event between state s_4 and s_5 since as determined previously, the transition can never be used towards a violation.

We can show that the residual problem pair thus created is equivalent to the original problem.

Theorem 3.2. $(P \backslash \pi, \pi \backslash P)$ *is in lockstep with* (P, π).

Proof Sketch. Consider that any step towards a violation at runtime must be reflected in a composition. These steps are exactly those that have corresponding useful transitions in a composition, and thus remain in the residual problem. Any transitions removed or silenced had no effect on the verdict in the original problem and being removed in the residual problem thus does not affect the verdict given, allowing us to easily conclude the theorem. $\qquad \square$

Using the composition, we can furthermore define another residual operation that operates on DEA guards. Consider that looking at the composition we have an over-approximation of when DEA transitions match CFA transitions. A special case is when we find that a certain DEA transition may *always be used when it is an option*, i.e. from any configuration that can trigger such a transition then all outgoing transitions concretely match that transition. In this case, since the composition is sound, we can conclude that this also holds at runtime and thus we can avoid computing the guard of such a transition by transforming it into the *true* guard.

Definition 3.9 (Guard Residual). $\pi \backslash\backslash P$ *is the property with the union of the following transition sets:*

(i) the DEA transitions that are not always activated:

$$\rightarrow_0 \overset{\text{def}}{=} \left\{ q \xrightarrow{e|g \mapsto a} q' \mid \begin{array}{c} \exists M \in methods(P), s, s', s'' \in S_M, \\ q'' \in Q \cdot (s,q) \xrightarrow[e|g \mapsto a]{c \triangleright st \blacktriangleright e} (s',q') \wedge \\ \exists l \neq (e,g,a) \cdot (s,q) \xrightarrow[l]{c \triangleright st \blacktriangleright e} (s'',q'') \end{array} \right\}$$

and

(ii) the DEA transitions that are always activated with silenced guards:

$$\rightarrow_1 \overset{\text{def}}{=} \left\{ q \xrightarrow{e|true \mapsto a} q' \mid \begin{array}{c} (q,e,g,a,q') \in \rightarrow_{\pi \backslash P} \wedge \\ (q,e,g,a,q') \notin \rightarrow_0 \end{array} \right\}.$$

Then $\rightarrow_{\pi \backslash\backslash P} \overset{\text{def}}{=} \rightarrow_0 \cup \rightarrow_1$.

Consider Figure. 3b: We can easily determine that `ongoing` is always true when **before**(`offer`) occurs, consider Table. 1, and similarly for the silenced guards in Figure. 3a. Note how in the definition of the guard residual, we are analysing the control-flow residual and not the original property (see (ii)). This is necessary since considering the transitions of π can lead us to create a non-deterministic DEA or a DEA with different behaviour. Consider using π naïvely instead of $\pi \backslash P$ in condition (ii): This would lead us to have to consider transitions never activated in the

composition. Silencing their guards could change the behaviour at runtime or clash with transitions from the same state. Using $\pi \backslash P$ avoids this.

Theorem 3.3. $(P, \pi \mathbin{\backslash\!\backslash} P)$ *is in lockstep with* (P, π).

Proof Sketch. Consider how the composition makes explicit any possible intraprocedural behaviour, including no matching (see rule (iii) of the composition). Then, weakening a guard when there is no alternative transition from the same property state in any composition is sound since then there are no alternatives at runtime either. $\square$

This finishes our presentation of our framework for residual analysis using symbolic automata, while we have introduced several residual operators that we evaluate in the next section.

4 EVALUATION

To evaluate the techniques in this paper, we considered the running example and the various properties we defined over it. We runtime verified these properties both before and after residual analysis, measuring the overheads in each case, using the CONTRACTLARVA tool [Azzopardi et al., 2018].

Traditionally these overheads are measured in terms of two aspects: (i) the time taken in both setups; and (ii) the memory consumed by both setups. These are both measures of different aspects of the computation required for monitoring. The running example we use here, Listing. 1, is a smart contract written in the Solidity language intended for the Ethereum blockchain [Wood, 2014]. In Ethereum any computation requested has to be paid for according to a preset deterministic unit called *gas*, which has a variable real-world cost. Thus reducing the computation performed on the blockchain is imperative, to reduce the cost of a service running off the blockchain. Gas is also intended to be proportional to the effort required to store and compute on the blockchain. We then use this measure to evaluate our residual techniques here.

Table. 2 illustrates the results of the evaluation, listing the percentage of gas overheads that (i) monitoring the original property adds; (ii) monitoring the residual property adds; and (iii) savings of the latter over the former. Consider how there are significant reductions to gas costs associated with deployment, except in the last case since no reductions could be made to the property. The results with function calls vary, with significant reductions for the `offer` method in the first partially proven property, with little reductions in the other cases.

	DEA	Monitored % of original gas	Residual % of original gas	Savings % of monitored gas
Deploy.	Figure. 3a	65.87%	36.33%	28.58%
	Figure. 3b	32.55%	0%	100%
	Figure. 3c	21.40%	0%	100%
	Figure. 3d	48.69%	48.69%	0%
start	Figure. 3a	0.92-25.07%	0.92-25.07%	0%
	Figure. 3b	0.02-9.36%	0%	100%
	Figure. 3c	1.30-24.17%	0%	100%
	Figure. 3d	0.94-93.63%	0.94-93.63%	0%
offer	Figure. 3a	3.87-5.68%	2.63-4.01%	27.71-31.89%
	Figure. 3b	14.86-72.15%	0%	100%
	Figure. 3c	1.43-31.65%	0%	100%
	Figure. 3d	0.2-1.40%	0.2-1.40%	0%
declare	Figure. 3a	40.30%	37.75%	6.32%
	Figure. 3b	87.56%	0%	100%
	Figure. 3c	0.62%	0%	100%
	Figure. 3d	26.58%	26.58%	0%
goings	Figure. 3a	3.01-77.57%	3.01-77.57%	0%
	Figure. 3b	0.86-1.33%	0%	100%
	Figure. 3c	0.85-1.33%	0%	100%
	Figure. 3d	0.86-1.33%	0.86-1.33%	0%

Table 2: Evaluation of analysis with running example[7].

These result shows how analysing properties before runtime verification can help avoid some runtime overheads, however these reductions in overheads depend significantly on how much of the property and instrumentation was removed. An issue this evaluation does not tackle is how useful these analyses are when combined with more precise static analyses.

5 DISCUSSION

In this work we made the decision to use a representation of programs that is similar to our specification language, begging the question: why not simply use DEAs as both our program and specification models? The response is that DEAs cannot be used elegantly to represent the actual behaviour of a program. Consider that there are several differences between DEAs and programs: (i) DEAs are passive (they wait for events to occur) whereas programs are active (they trigger events); (ii) programs call other programs sequentially whereas DEAs are monolithic; and (iii) it is not clear how DEAs can be extracted from programs while maintaining a one-to-one connection to program code necessary to turn off some instrumentation as required. CFAs on the other hand are easily extractable from common programming languages like Java while they allow for calls to other CFAs. However CFAs are not appropriate for high-level specifications that intersect across different modules, unlike DEAs. What is missing from our analysis here is the consideration of concurrency, which we leave for future work.

We also made a choice in the definition of CFA operational semantics to define the semantics of calls recursively. Another option would have been to use a stack-based approach, with configurations as pairs

of symbolic states and a stack (or sequence) of explicit states. Upon a call, the initial state of the called method is pushed onto stack, and popped upon reaching an end state. Here we avoided this for simplicity, although we do require a form of interprocedurality to prove our results [Azzopardi et al., 2019].

The residual analyses here are quite weak on two fronts: (i) they are intraprocedural; and (ii) they depend on universal observations about the program (i.e. a transition's guard is removed if it is *always* activated). Adding efficient interprocedurality can be done using some form of weighted pushdown system [Reps et al., 2007]. Moreover the current approach can be extended by easily obtainable interprocedural information, e.g. limiting the chaos at call states by computing the alphabet of the transitively called methods. Analysis of the composition can be further extended to allow for existential observations about the program. This would allow us, perhaps through special transitions, to determine points in the program for which a certain side-effect-free guard can be precomputed. This proposed approach has similarities with work of [Dwyer and Purandare, 2007].

The work presented here can also be extended with a notion of *typestate*, i.e. a property replicated for each object (or a binding of multiple objects) of a certain kind, which we do not detail here for conciseness. The extension for typestate is straightforward: (i) associate events in CFA transitions with objects (dynamic on the program symbolic state); (ii) introduce a pointer analysis to identify equivalence classes of events according to the runtime objects they may be associated with; and (iii) project analysis onto each possible event equivalence class.

An observation about partial verification is that it can be seen as an extension of usual verification. Consider that verification is usually framed as an operation that either returns a satisfying, violating, or unknown verdict: $ver : \mathcal{P} \times \Pi \to \{\top, \bot, ?\}$. Partial verification extends this by instead returning a new program and property pair: $partialVer : \mathcal{P} \times \Pi \to \mathcal{P} \times \Pi$. To signal violation the residual property can be the always violating property $\pi_\bot$, while to signal satisfaction we can use the always satisfying property $\pi_\top$.

6   RELATED WORK

We find several approaches towards partial verification in literature. [Bodden and Lam, 2010] introduce a tool for the control-flow analysis of Java programs against finite-state automata properties. This tool incorporates several analyses that safely turn off event instrumentation in the program. An interesting analysis this performs takes into account *continuation-equivalency* [Bodden, 2010], where backwards and forwards analyses are used to determine different points in the program that have the same continuation with respect to the property, allowing the turning off of any redundant events whose silencing does not change the continuation. This analysis proves to be useful for finite-state automata, however given that DEAs contain guarded events it is not clear if this would be as useful. This approach is different from ours in that the focus is solely on reducing instrumentation, while we also encode inconclusive results in a residual property. Differently from our approach this tool takes into account typestate.

Another approach is STARVOORS [Ahrendt et al., 2012] that deals with DEAs extended with pre- and post-conditions for methods called when at certain concrete states. In this work, a theorem prover is used to prove these conditions, where they are either pruned away or refined. This analysis ignores the control-flow of the property, unlike our approach.

A different summary-based approach is taken by [Dwyer and Purandare, 2007]. Instead of removing transitions from the property and instrumentation from the program, [Dwyer and Purandare, 2007] analyse the program to identify sequences of statements that always start and end at the same property states. Instrumentation in this sequence is removed, with new special events added signaling entry and exit into the sequence. A transition is added accordingly in the property. Unlike our approach, the residual property and program here are transformations of the original rather than structural reductions.

[Beyer et al., 2018] also use automata they call control-flow automata, similar to ours, with transitions corresponding to operations on a program's variable state. However they take a state-based approach, where they are interested in verifying assertions at certain points in the program. They encode what is learned about a program in a *condition automaton*, that represents both the safe and potentially unsafe program paths. These automata are used to create residual programs with safe parts pruned away. This approach is quite general and can be applied to the problem of verifying against a DEA. The motivation here is different from our approach since we are interested in residual monitoring of the whole program.

7   CONCLUSIONS

Partial verification is an approach to verification where on analysis failing, its partial results are used to produce a simpler residual problem for subse-

quent passes. In this paper we have presented an event-based framework that can be used as the basis to produce residuals verification problems, while we have presented novel residual operators that analyse the program intraprocedurally to identify unnecessary event instrumentation, property transitions, and property event guards. We evaluated this with a Solidity smart contract, showing some gains depending on how much of a property was proven.

We are working on an implementation [8] of this approach for CONTRACTLARVA [Azzopardi et al., 2018], a tool for runtime verification on the Ethereum blockchain. A next logical step is to consider extensions of our work to when parts of the program are unknown or dynamic at runtime. We believe our approach to dealing with intraprocedural analysis, i.e. by considering call states as chaotic, can be re-used for this purpose. Missing from literature is also residuals of temporal logic based specifications, which are popular formalisms for verification.

REFERENCES

Ahrendt, W., Pace, G. J., and Schneider, G. (2012). A unified approach for static and runtime verification: Framework and applications. In Margaria, T. and Steffen, B., editors, *Leveraging Applications of Formal Methods, Verification and Validation - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, Proceedings, Part I*, volume 7609 of *LNCS*, pages 312–326. Springer-Verlag.

Azzopardi, S., Colombo, C., and Pace, G. (2019). A technique for automata-based verification with residual reasoning. Technical Report CS-2019-02, Department of Computer Science, University of Malta.

Azzopardi, S., Colombo, C., and Pace, G. J. (2017). Control-flow residual analysis for symbolic automata. In Francalanza, A. and Pace, G. J., editors, *Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, Torino, Italy, 19 September 2017*, volume 254 of *Electronic Proceedings in Theoretical Computer Science*, pages 29–43. Open Publishing Association.

Azzopardi, S., Colombo, C., and Pace, G. J. (2020). CLARVA: Model-based residual verification of java programs. In *Model-Driven Engineering and Software Development - 8th International Conference, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020*.

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

Beyer, D., Henzinger, T. A., Keremoglu, M. E., and Wendler, P. (2012). Conditional model checking: A technique to pass information between verifiers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 57:1–57:11, New York, NY, USA. ACM.

Beyer, D., Jakobs, M.-C., Lemberger, T., and Wehrheim, H. (2018). Reducer-based construction of conditional verifiers. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1182–1193, New York, NY, USA. ACM.

Bodden, E. (2010). Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE '10: International Conference on Software Engineering*, pages 5–14, New York, NY, USA. ACM.

Bodden, E. and Lam, P. (2010). Clara: Partially Evaluating Runtime Monitors at Compile Time. In *1st International Conference on Runtime Verification (RV)*, volume 6418 of *LNCS*, pages 183–197. Springer. Tutorial.

de Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

Dwyer, M. B. and Purandare, R. (2007). Residual dynamic typestate analysis exploiting static analysis: Results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 124–133, New York, NY, USA. ACM.

Dwyer, M. B. and Purandare, R. (2008). Residual checking of safety properties. In Havelund, K., Majumdar, R., and Palsberg, J., editors, *Model Checking Software*, pages 1–2, Berlin, Heidelberg. Springer Berlin Heidelberg.

Falcone, Y., Krstić, S., Reger, G., and Traytel, D. (2018). A taxonomy for classifying runtime verification tools. In Colombo, C. and Leucker, M., editors, *Runtime Verification*, pages 241–262, Cham. Springer International Publishing.

Lal, A., Kidd, N., Reps, T., and Touili, T. (2007). Abstract error projection. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, pages 200–217, Berlin, Heidelberg. Springer-Verlag.

Reps, T., Lal, A., and Kidd, N. (2007). Program analysis using weighted pushdown systems. In Arvind, V. and Prasad, S., editors, *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, pages 23–51, Berlin, Heidelberg. Springer Berlin Heidelberg.

Vardi, M. Y. (2007). Automata-theoretic model checking revisited. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'07, pages 137–150, Berlin, Heidelberg. Springer-Verlag.

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

---

[8] https://www.github.com/shaunazzopardi/solidity-static-analysis