TECHNICAL REPORT

Report No. CS2017-01
Date: April 2017

Control-Flow Residual Analysis for Symbolic Automata

Shaun Azzopardi
Christian Colombo
Gordon J. 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

Control-Flow Residual Analysis for Symbolic Automata

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: *Where full static analysis of systems fails to scale up due to system size, dynamic monitoring has been increasingly used to ensure system correctness. The downside is, however, runtime overheads which are induced by the additional monitoring code instrumented. To address this issue, various approaches have been proposed in the literature to use static analysis in order to reduce monitoring overhead. In this paper we generalise existing work which uses control-flow static analysis to optimise properties specified as automata, and prove how similar analysis can be applied to more expressive symbolic automata - enabling reduction of monitoring instrumentation in the system, and also monitoring logic. We also present empirical evidence of the effectiveness of this approach through an analysis of the effect of monitoring overheads in a financial transaction system.*

Control-Flow Residual Analysis for Symbolic Automata*

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: *Where full static analysis of systems fails to scale up due to system size, dynamic monitoring has been increasingly used to ensure system correctness. The downside is, however, runtime overheads which are induced by the additional monitoring code instrumented. To address this issue, various approaches have been proposed in the literature to use static analysis in order to reduce monitoring overhead. In this paper we generalise existing work which uses control-flow static analysis to optimise properties specified as automata, and prove how similar analysis can be applied to more expressive symbolic automata - enabling reduction of monitoring instrumentation in the system, and also monitoring logic. We also present empirical evidence of the effectiveness of this approach through an analysis of the effect of monitoring overheads in a financial transaction system.*

1 Introduction

The need for verification of a system to be able to make some guarantees about execution paths, and going beyond sampling of such paths (as done in testing), is required for critical or sensitive software (e.g. financial software [ACPV16]). The literature can be largely split into two main approaches: (i) full *a priori* verification of all possible execution paths through model checking, static analysis and similar techniques, and (ii) *on the fly* verification of execution paths to ensure that any potential violation can be immediately truncated as in runtime verification.

*This research has received funding from the European Union's Horizon 2020 research and innovation programme under grant number 666363.

The former approaches, tend not to scale to more complex and large software, which is typically addressed by abstraction techniques e.g. verifying the property against an over-approximation of the program (if the over-approximation cannot violate the property, then the program cannot either), due to which the analysis is no longer complete. The latter approaches readily scale up to handle larger systems, and have the additional advantage that they can also deal with constraints on the environment which can only be verified fully at runtime (e.g. two methods of an API are never called in sequence by an unknown client application). The downside is, however, that the additional checks introduced typically add significant runtime overheads [PDE12]. Traditionally, these two approaches have been generally seen as alternatives to each other, although their possible complementarity has started to be explored in recent years [APS12, BLH12, DP07].

Approaches exist that use static analysis to prove parts of a property with respect to a program, such that either the whole property is proved statically or it is pruned such that there is less to monitor at runtime, or vice-versa such that certain parts of the program are proved safe to not monitor. In particular, Clara [BLH12] is one such approach, acting on specifications or properties defined as automata with transitions triggered by method invocations in Java programs. Through an analysis of the source code, Clara can be used to determine whether a property transition can never be taken by the program and whether parts of a program can be safely unmonitored. Removing such transitions from a property creates a residual property that is smaller than, or equivalent to, the original property and since some parts of the program will not activate the monitor any longer, then there will be less overheads at runtime due to monitoring. The approach uses three analysis steps, each equivalent to a comparison of the property automaton with a finer over-approximation of the control-flow of a program. However, in practice, one frequently desires properties which are more expressive than these simple automata.

DATEs (Dynamic Automata with Events and Timers) [CPS09] are symbolic automata that can be seen as an extension of Clara's properties such that transitions' triggering depends on either a method invocation or some timer event, along with a condition on variables specific to the monitor or the program. Moreover, if triggered, a transition can also perform an action (e.g. increase some internal counter), introducing the notion of a transition having side-effect, which needs to be taken into account when producing a residual. DATEs are used within the LARVA (Logical Automata for Runtime Verification and Analysis) tool to specify the security properties for a Java system.

Our contribution in this paper is two-fold: (i) extending Clara's first two analyses to produce both a residual DATE and a residual instrumentation of the program, and (ii) a new analysis that uses the control-flow graph of a program to determine if any transitions in a DATE can (or not) be reached by the program.

In Section 2 we briefly introduce some formal notation and informally discuss Clara, while

we present our generalisation of it for DATEs in Section 3. In Section 4, a case study involving a financial transaction system is considered along with some properties as a backdrop against which we discuss the utility of the presented analysis. We present related work in Section 5, and conclude proposing future work in Section 6.

2 Programs' Runtime Traces and Abstractions

In this section we look at control-flow properties written in the form of automata, with transitions being triggered by method invocation events.

Definition 1 (Property automata)**.** *A* property automaton π *is a tuple* $\langle Q, \Sigma, q_0, B, \delta \rangle$*, where* Q *is a finite set of states,* Σ *is a set of events,* q_0 *is the initial state (*$q_0 \in Q$*),* B *is the set of bad states (*$B \subseteq Q$*), and* δ *is the transition relation (*$\delta \subseteq Q \times \Sigma \times Q$*), which is deterministic and total with respect to* $Q \times \Sigma$*.*

We write $q \xrightarrow{e}_\pi q'$ *for* $(q, e, q') \in \delta$*,* $q \overset{es}{\Rightarrow}_\pi q'$ *for the transitive closure of* δ *(with* $es \in \Sigma^*$*), and* $q \hookrightarrow_\pi q'$ *to denote that* q' *is reachable from* q *(i.e.* $\exists es \cdot q_0 \overset{es}{\Rightarrow}_\pi q'$*). We will leave out* π *whenever it is clear from the context.*

Finally, we define the restriction of a property's alphabet to a certain alphabet:

$$\delta \restriction \Sigma' \overset{\text{def}}{=} \{q \xrightarrow{e} q' \mid e \in \Sigma'\}$$

We will write $\pi \restriction \Sigma'$ *and* $\pi \restriction \delta'$ *to denote the property automaton identical to* π *except with the transition relation* $\delta \restriction \Sigma'$ *and* δ' *respectively.*

The problem of verification, given a control-flow automaton property and a program, is to ensure that all traces in the program never transition into a bad property state [LS09]. We call an event trace $t \in \Sigma^*$ a *ground trace*, while we denote the set of events appearing in t by $\Sigma(t) \overset{\text{def}}{=} \{e \mid \exists i \in \mathbb{N} \cdot t(i) = e\}$, overloaded to sets of ground traces $T \subseteq \Sigma^*$.

Definition 2 (Property satisfaction)**.** *A ground trace* $t \in \Sigma^*$ *is said to satisfy property automaton* π *if no prefix of* t *takes the property automaton from the initial state to a bad one:* $t \vdash \pi \overset{\text{def}}{=} \forall t' \in \mathit{prefixes}(t) \cdot \nexists q_B \in B \cdot q_0 \overset{t'}{\Rightarrow} q_B$*. We overload this notation to sets of ground traces* $T \vdash \pi$ *to indicate that all traces in* T *satisfy* π*.*

Consider as an example the property automaton shown in Figure 1, which specifies that the *write* method cannot be called before *open* is called, and that the *read* method is called in pairs. Bad states are marked in red, while an asterisk (*) on a transition is syntactic sugar used to denote that if at that state an event happens for which no other transition matches, then the asterisk transition is taken.

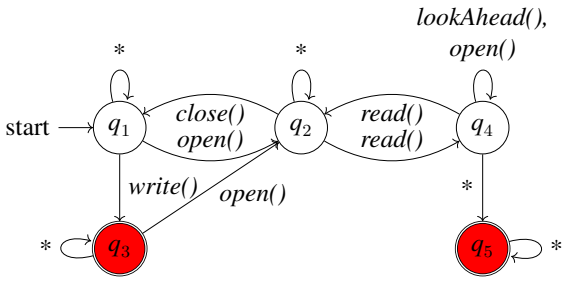

Figure 1: Property disallowing writing on a closed stream, and writing or closing while in the middle of an odd number of reads.

In practice, one would want to instantiate a property automaton for every instance of the object being verified. For example, the property automaton shown in Figure 1 should ideally be monitoring for every stream in use. To enable such replication of property automata, we extend them to parametric properties [JMGR11]. To generalise property automata to handle parametrisation, we start by extending traces to parametrised traces in which each event is associated with an identifier of the object[1] to which the event pertains. Note that we allow an alphabet to be parametrised by a set of identifiers α by: $\Sigma_\alpha \stackrel{\text{def}}{=} \alpha \times \Sigma$. Traces over such alphabets will be referred to as parametrised traces. At runtime it is also possible to consider when two identifiers refer to the same object, which we simulate through an equivalence relation between identifiers. Thus, at runtime, if *open* and *write* are called in succession we cannot immediately conclude non-violation of the property, since they may have been called on different objects (Java objects or with respect to some data).

Satisfaction of property automata can be lifted over parametrised traces by checking for the satisfaction of each possible projection. In such cases our property automata are effectively typestate property automata [BLH12].

Definition 3 (Parametrised traces)**.** *The projection of a parametrised trace $rt \in \Sigma^*_\alpha$ with respect to an identifier $x \in \alpha$ and an equivalence relation between objects $\equiv \in \alpha \leftrightarrow \alpha$, written $rt \downarrow x$, is defined to be the sequence of items in rt equivalent to x. [2]:*

$$\langle\rangle \downarrow x \stackrel{\text{def}}{=} \langle\rangle$$

$$((x', p) : rt) \downarrow x \stackrel{\text{def}}{=} \begin{cases} (x', p) : (rt \downarrow x) & \text{if } x \equiv x' \\ rt \downarrow x & \text{otherwise} \end{cases}$$

A parametrised trace $rt \in \Sigma^*$ is said to satisfy a property automaton π, written $rt \Vdash \pi$, if for each identifier $x \in \alpha$, the projection of rt onto x satisfies the property:

[1] Although we will be using the term 'object' in this paper, events can be parametrised with respect to an identity other than the object on which the method is invoked.

[2] We use the standard notation $x : xs$ to denote the list with head x and tail xs.

$$rt \Vdash \pi \stackrel{\text{def}}{=} \forall x \in \alpha \cdot rt \downarrow x \vdash \pi$$

We overload this notation over sets of parametrised traces, $RT \subseteq \Sigma_\alpha^*$.

$$RT \downarrow x \stackrel{\text{def}}{=} \{rt \downarrow x \mid rt \in RT\}$$
$$RT \Vdash \pi \stackrel{\text{def}}{=} \forall rt \in RT \cdot rt \Vdash \pi$$

At runtime, we have perfect knowledge of the equivalence relation between parametrised events. However, when using static analysis, this is not always possible. Alias analysis [BLH08] can give partial information whether two method calls in the code always correspond to the same object. In such cases, we have three possible outcomes: (i) the two event generators always refer to the same object; (ii) the two event generators always refer to different objects; and (iii) neither of the previous two cases can be concluded. In the literature, this information is typically encapsulated in two relations — a *must* relation $\equiv$, which relates two event generators if their objects always (must) match, and a *may* relation $\equiv_{\text{may}}$, which relates two event generators if their objects may match, with the former relation being a subset of the latter.

Definition 4 (Statically Parametrised Traces)**.** *A statically parametrised trace is a parametrised $st \in \Sigma_\alpha^*$, with two relations over α: (i) a must-alias equivalence relation $\equiv \ \in \alpha \leftrightarrow \alpha$, and (ii) a may-alias relation $\equiv_{\text{may}} \in \alpha \leftrightarrow \alpha$, such that $\equiv \ \subseteq \equiv_{\text{may}}$. We define the projection[3] of a statically parametrised trace st with respect to parameter x, written $st \Downarrow x$, as follows:*

$$\langle\rangle \Downarrow x \stackrel{\text{def}}{=} \{\langle\rangle\}$$
$$((x', e) : ps) \Downarrow x \stackrel{\text{def}}{=} \begin{cases} \{e : es' \mid es' \in ps \Downarrow x\} & \text{if } x \equiv x' \\ ps \Downarrow x & \text{if } x \not\equiv_{\text{may}} x' \\ ps \Downarrow x \cup \{e : es' \mid es' \in ps \Downarrow x\} & \text{otherwise} \end{cases}$$

Trace st is said to satisfy property automaton π , written $st \Vdash\!\!\vdash \pi$, if for each identifier $x \in \alpha$ the projection of st onto x satisfies the property:

$$st \Vdash\!\!\vdash \pi \stackrel{\text{def}}{=} \forall x \in \alpha \cdot st \Downarrow x \vdash \pi$$

We overload this notation over sets of statically parametrised traces $ST \subseteq \Sigma_\alpha^*$:

$$ST \Downarrow x \stackrel{\text{def}}{=} \{t \mid st \in ST \cdot t \in st \Downarrow x\}$$
$$ST \Vdash\!\!\vdash \pi \stackrel{\text{def}}{=} \forall st \in ST \cdot st \Vdash\!\!\vdash \pi$$

[3]This corresponds to the notion of program slicing, by slicing the instrumented program statements according to whether they can associated with the same object or not.

It is worth noting that although we are only considering parametrisation over a single object, a typestate property over multiple objects can be reduced to a single identifier by looking at them as a tuple of identifiers with point-wise must and may-alias relations.

We now turn our view from individual traces to the programs which generate them.

Definition 5 (Programs)**.** *For a program P over an alphabet Σ with a set of objects Obj and object identifiers ObjId, (i) we will write P^R_Σ to denote the set of parametrised traces over Obj i.e. $P^R_\Sigma \subseteq \mathbb{\Sigma}^*_{Obj}$ with equivalence $\equiv$ over Obj; (ii) we will write P^S_Σ to denote the set of statically parametrised traces over ObjId i.e. $P^S_\Sigma \subseteq \mathbb{\Sigma}^*_{ObjId}$ with relations $\equiv$ and $\equiv_{\mathsf{may}}$.*

Object identifiers in *ObjId* are assumed to contain information about the instrumentation points in the code which triggers the associated event: $\mathcal{I} \in \mathit{ObjId} \to \mathit{InstrPnt}$. The set of *shadows* of a static approximation trace *st*, written $\mathit{shadows}(st)$, is defined to be the set of instrumentation points which appears somewhere along the trace. Given this, we can define what it means to silence part of a trace.

Definition 6. *Given a static approximation trace, st, and a set of instrumentation points $ips \subseteq \mathit{InstrPnt}$, the silencing of ips in st, written $\mathit{silence}(st, ips)$, is defined to be the original trace st except for elements mapped by $\mathcal{I}$ to an instrumentation point in ips:*

$$
\begin{array}{lcl}
\mathit{silence}(\langle\rangle, ips) & \stackrel{\text{def}}{=} & \langle\rangle \\
\mathit{silence}((oid, e) : ps, ips) & \stackrel{\text{def}}{=} & \begin{cases} \mathit{silence}(ps, ips) & \text{if } \mathcal{I}(oid) \in ips \\ (oid, e) : \mathit{silence}(ps, ips) & \text{otherwise} \end{cases}
\end{array}
$$

We overload the silencing operator to sets of static approximation traces, and static approximations of programs.

Given the definition of property satisfaction of static traces, we can immediately show that satisfaction of runtime and static programs can be expressed in terms of ground traces satisfaction.

Proposition 1. *The runtime parametrised traces of a program P satisfy a typestate property if and only if the ground traces satisfy it:*

$$P^R_\Sigma \Vdash \pi \iff P^R_\Sigma \downarrow \mathit{Obj} \vdash \pi$$

Proof. Follows immediately from Definitions 5, and 3. □

Proposition 2. *The static parametrised traces of a program P satisfy a typestate property if and only if the ground traces satisfy it:*

$$P^S_\Sigma \Vdash \pi \iff P^S_\Sigma \Downarrow \mathit{ObjId} \vdash \pi$$

Proof. Follows immediately from Definitions 5, and 9. □

In what follows, we make the assumption that the statically parametrised trace generator is correct, i.e. a statically parametrised program actually generates the ground traces generated by the program at runtime.

Assumption 1. $P^R_\Sigma \downarrow \mathit{Obj} \subseteq P^S_\Sigma \Downarrow \mathit{ObjId}$

This allows us to show that satisfaction of the static program implies that of the runtime program.

Proposition 3. *Given program P and property π, if its static over-approximation of a program, P^S_Σ satisfies property π, then the runtime behaviour of the program, P^R_Σ also satisfies the property:*

$$P^S_\Sigma \Vdash\!\vdash \pi \implies P^R_\Sigma \Vdash \pi$$

Proof.

$$\begin{aligned}
& P^S_\Sigma \Vdash\!\vdash \pi \\
& \quad \text{(by Proposition 2)} \\
\implies & P^S_\Sigma \Downarrow \mathit{ObjId} \vdash \pi \\
& \quad \text{(by assumption that } P^R_\Sigma \downarrow \mathit{Obj} \subseteq P^S_\Sigma \Downarrow \mathit{ObjId}\text{)} \\
\implies & P^R_\Sigma \downarrow \mathit{Obj} \vdash \pi \\
& \quad \text{(by Proposition 1)} \\
\implies & P^R_\Sigma \Vdash \pi
\end{aligned}$$

□

In general, a statically parametrised program can be acquired by considering a control-flow graph of a program.

Definition 7. *A control-flow graph (CFG) $\mathcal{C}$ over event alphabet Σ and object identifiers ObjId is a tuple: $\langle Q,\ \Sigma_{\mathit{ObjId}},\ q_0,\ F,\ \delta,\ \equiv,\ \equiv_{\mathsf{may}},\ \mathcal{I}\rangle$, Q is a finite set of states, Σ_{ObjId} is the alphabet, q_0 is the initial state ($q_0 \in Q$), F is the set of final states ($F \subseteq Q$) and δ is a transition function allowing for τ-transitions ($\delta \subseteq Q \times (\Sigma_{\mathit{ObjId}} \cup \{\tau\}) \times Q$). In addition, the CFG includes the must-alias relation (over ObjId) $\equiv$, may-alias relation (also over ObjId) $\equiv_{\mathsf{may}}$, and function $\mathcal{I}$ which associates object identifiers with the relevant instruction pointer ($\mathcal{I} \in \mathit{ObjId} \to \mathit{InstrPnt}$).*

As we did with property automata, we will write $q \xrightarrow{p}_{\mathcal{C}} q'$ to denote $(q,p,q') \in \delta$, $q \overset{ps}{\Rightarrow}_{\mathcal{C}} q'$ for the transitive closure of δ (with $ps \in \Sigma^*_{\mathit{ObjId}}$). We will also omit $\mathcal{C}$ whenever it is clear from the context.

Note that CFGs allow for τ-transitions, corresponding to internal steps in the program which do not trigger visible events (and will thus not activate a property transition). Also, a CFG can be transformed into a set of static traces corresponding to all strings taken along paths in the CFG structure and ignoring τ transitions, based on which we can define what it means for a CFG to satisfy a parametrised property.

$$P^S_\Sigma(\mathcal{C}_\Sigma) \stackrel{\text{def}}{=} \{ps \mid \exists q_F \in F \cdot q_0 \stackrel{ps}{\Rightarrow} q_F \wedge \nexists q'_F \in F, ps', ps'' \cdot ps = ps'; ps'' \wedge q_0 \stackrel{ps'}{\Rightarrow} q'_F\}$$

CFGs can be constructed directly from concrete programs using techniques as used, for instance, in [BLH12], in which the CFGs of individual methods are constructed, and extended by flattening behaviour outside the method itself. We have also extended Bodden's approach to create these over-approximate CFGs in [ACP17].

The approaches we will explore in this paper involve property-specific transformations of program monitoring in order to reduce overheads in such a manner that will not affect the verdict with respect to that particular property. Next, we define notions of trace and program equivalence with respect to a property or program.

Definition 8 (Equivalence between properties)**.** *Two properties π and π' are said to be equivalent with respect to a set of ground traces T, written $\pi \cong_T \pi'$, if every trace in T is judged the same by either property:*

$$\pi \cong_T \pi' \stackrel{\text{def}}{=} \forall t \in T \cdot t \vdash \pi \iff t \vdash \pi'$$

This notion is lifted to parametrised and statically parametrised traces, and sets thereof, requiring equivalence up to object identifiers.

$$\pi \cong_{RT} \pi' \stackrel{\text{def}}{=} \forall o \in \mathit{Obj} \cdot \pi \cong_{RT\downarrow o} \pi'$$
$$\pi \cong_{ST} \pi' \stackrel{\text{def}}{=} \forall \mathit{oid} \in \mathit{ObjId} \cdot \pi \cong_{ST\Downarrow \mathit{oid}} \pi'$$

It is straightforward to prove that equivalence up to traces is an equivalence relation.

Proposition 4. *$\cong_\pi$, $\cong_T$,$\cong_{RT}$, and $\cong_{ST}$ are all equivalence relations.*

Proof. This follows from the definition over ground traces using $\iff$. □

Equivalence of traces is then preserved when reducing the set of traces.

Proposition 5. $\forall T' \subseteq T \cdot \pi \cong_T \pi' \implies \pi \cong_{T'} \pi'$

Proof. This follows from the definition of $\cong_T$ and $\vdash$. □

Given this, we can conclude the following proposition.

Proposition 6. *If* $\pi \cong_T \pi'$ *and* $\pi' \cong_{T'} \pi''$*, then* $\pi \cong_{T \cap T'} \pi''$.

We can also show that equivalence with respect to an approximation of a program can be expressed in terms of its projection onto ground traces, following from Proposition 2.

Proposition 7. $\pi \cong_{P^S_\Sigma} \pi' \iff \pi \cong_{P^S_\Sigma \Downarrow ObjId} \pi'$.

Equivalence between programs statically is more complex. We shall be using the silencing operator on a static trace, which will reduce the concrete traces generated from it, and thus we want to ensure correspondence between the original concrete generated traces and the reduced traces (ensuring that a trace is given by the same verdict as its corresponding silenced trace). To verify this correspondence at the low-level of concrete traces, we will require knowledge about the identifier originally associated with each event. To represent this we shall define a projection operator that instead of generating concrete traces, generates another set of statically parametrized traces, with each having a one-to-one correspondence with the concrete traces generated by the concrete projection operator. This in effect is a maximal representative set of static traces needed to represent the concrete traces generated by the original static trace.

Definition 9 (Generating a maximal representative set of static traces)**.** *We generate the maximal representative set of static traces to represent a static trace st as follows:*

$$
\begin{array}{lcl}
\mathit{maxRep}(\langle\rangle, x) & \stackrel{\text{def}}{=} & \{\langle\rangle\} \\
\mathit{maxRep}((x', e) : ps, x) & \stackrel{\text{def}}{=} & \begin{cases}
\{(x', e) : ps' \mid ps' \in \mathit{maxRep}((x', e) : ps, x)\} & \text{if } x \equiv x' \\
\mathit{maxRep}(ps, x) & \text{if } x \not\equiv_{\mathsf{may}} x' \\
\mathit{maxRep}(ps, x) & \\
\quad \cup \{(x', e) : ps' \mid ps' \in \mathit{maxRep}(ps, x)\} & \text{otherwise}
\end{cases}
\end{array}
$$

The must relation is different for each of these static traces, where it relates every identifier used in the trace with every other, given such a static trace st, $x \equiv^{st}_{eq} y \stackrel{\text{def}}{=} \exists n, m \cdot st(n) = (x, e) \wedge st(m) = (y, e')$ *(we will write* $\equiv_{eq}$ *when st is clear from the context), and the may relation is equal to the must relation.*

Note how this definition mirrors exactly the concrete projection operator's definition, with the addition that each event is tagged with the original identifier. Using the equivalence relation as defined in Definition. **??** also ensures that each trace in the maximal representative set generates only one concrete trace, and that the set of concrete traces generated by the original trace is equal to that generated by each of the static traces in *maxRep*.

Proposition 8. *Given a static trace, each static trace in its maximal set generates only one static trace.*

$$\forall st' \in \mathit{maxRep}(st, x) \cdot |st' \Downarrow_{\equiv_{eq}} x| = 1$$

Proof.

By structural induction over st':

Base Case: $st' = \langle\rangle$

Proof: $st' \Downarrow_{\equiv_{eq}} x$

(by definition of $\Downarrow$)

$=\{\langle\rangle\}$

Inductive Hypothesis: $|st' \Downarrow_{\equiv_{eq}} x| = 1$

Inductive Case: $|((x', e) : st') \Downarrow_{\equiv_{eq}} x| = 1$

Proof: $((x', e) : st') \Downarrow_{\equiv_{eq}} x$

(by definition of $\Downarrow$ and $\equiv_{eq}$)

$=\{e : es' \mid es' \in st' \Downarrow x\}$

(by inductive hypothesis)

$\Longrightarrow |((x', e) : st') \Downarrow_{\equiv_{eq}} x| = 1$

□

Proposition 9. *The projection of a static trace over a parameter generates the same set of concrete traces as the union of the concrete trace generated by each static traces in the maximal set of the original trace.*

$$st \Downarrow_{\equiv} x = \bigcup_{st' \in maxRep(st,x)} st' \Downarrow_{\equiv_{eq}} x$$

Proof.

By structural induction over st:

Base Case: $st = \langle\rangle$

Proof: $\langle\rangle \Downarrow_{\equiv} x$

(by definition of $\Downarrow$)

$=\{\langle\rangle\}$

$$\bigcup_{st' \in maxRep(\langle\rangle,x)} st' \Downarrow_{\equiv_{eq}} x$$

(by definition of *maxRep*)

$$= \bigcup_{st' \in \{\langle\rangle\}} st' \Downarrow_{\equiv_{eq}} x$$

$$
\begin{aligned}
&\text{(by definition of union)}\\
=&\langle\rangle \Downarrow_{\equiv_{eq}} x\\
&\text{(by definition of } \Downarrow)\\
=&\{\langle\rangle\}
\end{aligned}
$$

Inductive Hypothesis: $st \Downarrow_{\equiv} x = \bigcup_{st' \in maxRep(st,x)} st' \Downarrow_{\equiv_{eq}} x$

Inductive Case: $((x', e) : st) \Downarrow_{\equiv} x = \bigcup_{st' \in maxRep(((x',e):st),x)} st' \Downarrow_{\equiv_{eq}} x$

Proof:

Case 1: $x \equiv x'$

$$
\begin{aligned}
&\bigcup_{st' \in maxRep(((x',e):st),x)} st' \Downarrow_{\equiv_{eq}} x\\
&\text{(by definition of } \textit{maxRep})\\
=&\bigcup_{st' \in \{((x',e):st'') \mid st'' \in maxRep(st,x)\}} st' \Downarrow_{\equiv_{eq}} x\\
&\text{(simplifying the expression)}\\
=&\bigcup_{st'' \in maxRep(st,x)} ((x', e) : st'') \Downarrow_{\equiv_{eq}} x\\
&\text{(by definition of } \Downarrow)\\
=&\bigcup_{st'' \in maxRep(st,x)} \{e : es \mid es \in st'' \Downarrow_{\equiv_{eq}} x\}\\
&\text{(pushing in the union)}\\
=&\{e : es \mid es \in \bigcup_{st'' \in maxRep(st,x)} st'' \Downarrow_{\equiv_{eq}} x\}\\
&\text{(by the inductive hypothesis)}\\
=&\{e : es \mid es \in st \Downarrow_{\equiv} x\}\\
&\text{(by definition of } \Downarrow)\\
&((x', e) : st) \Downarrow_{\equiv} x
\end{aligned}
$$

Case 2: $x \not\equiv_{\mathsf{may}} x'$

$$
\begin{aligned}
&((x', e) : st) \Downarrow_{\equiv} x\\
&\text{(by definition of } \Downarrow)
\end{aligned}
$$

$$= st \Downarrow_{\equiv} x$$

(by the inductive hypothesis)

$$= \bigcup_{st' \in maxRep(st,x)} st' \Downarrow_{\equiv_{eq}} x$$

(by definition of *maxRep*)

$$= \bigcup_{st' \in maxRep(((x',e):\, st),x)} st' \Downarrow_{\equiv_{eq}} x$$

Case 3: $x \equiv_{\mathsf{may}} x'$

$$((x', e):\, st) \Downarrow_{\equiv} x$$

(by definition of $\Downarrow$)

$$= st \Downarrow_{\equiv} x \cup \{e:\, es \mid es \in st \Downarrow_{\equiv} x\}$$

(by induction hypothesis)

$$= st \Downarrow_{\equiv} x \cup \{e:\, es \mid es \in \bigcup_{st'' \in maxRep(st,x)} st'' \Downarrow_{\equiv_{eq}} x\}$$

(pushing out the union)

$$= st \Downarrow_{\equiv} x \cup \bigcup_{st'' \in maxRep(st,x)} \{e:\, es \mid es \in st'' \Downarrow_{\equiv_{eq}} x\}$$

(inductive hypothesis and definition of $\Downarrow$)

$$= \bigcup_{st' \in maxRep(st,x)\}} st' \Downarrow_{\equiv_{eq}} x \cup \bigcup_{st'' \in maxRep(st,x)} ((x', e):\, st'') \Downarrow_{\equiv_{eq}} x$$

(joining unions)

$$= \bigcup_{st' \in maxRep(st,x) \cup \{(x',e):\, st'' \mid st'' \in maxRep(st,x)\}} st' \Downarrow_{\equiv_{eq}} x$$

(by definition of *maxRep*)

$$= \bigcup_{st' \in maxRep(((x',e):\, st),x)} st' \Downarrow_{\equiv_{eq}} x$$

□

Considering a transformation between static traces, $\alpha : \mathbb{\Sigma}^*_{ObjId} \to \mathbb{\Sigma}^*_{ObjId}$ (note that silencing some instrumentation points is an example of this), we can define what it means for it to be isomorphic with respect to a set of traces, enabling a one-to-one correspondence between the original representative set and the transformed one.

Definition 10. *A transformation is said to be isomorphic with respect to the maximal representative set of a static trace, or* representative isomorphic*, if the maximal representative set of the transformed trace is made up of the application of the transformation on each of the original representatives:* $\mathit{maxRep}(\alpha(st), x) = \{\alpha(st') \mid st' \in \mathit{maxRep}(st, x)\}$

We use these maximal representatives to define equivalence of a static trace with a representative isomorphic transformation. We use this notion of isomorphism to ensure correspondence between each possible concrete trace generated, which will allow us to conclude equivalence between a program at runtime before and after such a transformation.

Definition 11 (Equivalence between static approximations of programs)**.** *Given a static trace st, and a representative isomorphic transformation function* $\alpha : \mathbb{\Sigma}^*_{ObjId} \to \mathbb{\Sigma}^*_{ObjId}$*, then st is said to be equivalent to its transformation with respect to a property* π*, if the transformation preserves the verdict of each representative.*

$$st \cong_\pi \alpha(st) \stackrel{\text{def}}{=} \forall st' \in \mathit{maxRep}(st, x) \cdot st' \Vdash_{\equiv} \pi \iff \alpha(st') \Vdash_{\equiv_{eq}} \pi$$

This notion is lifted to parametrised and statically parametrised traces, and sets thereof, by lifting a representative isomorphic transformation function α *to sets:* $A(ST) \stackrel{\text{def}}{=} \{\alpha(st) \mid st \in ST\}$.

$$ST \cong_\pi A(ST) \stackrel{\text{def}}{=} \forall st \in ST \cdot st \cong_\pi \alpha(st)$$

This definition ensures that the program monitored with the original instrumentation is equivalent to that monitored after the transformation. To show this we shall first characterize more finely the program at runtime as corresponding to a subset of the statically parametrized program.

Theorem 1. *The behaviour of each object at runtime is modelled statically by at least one static trace, moreover by at least one of the trace's maximal representatives.*

$$\forall rt \in P^R_\Sigma, o \in \mathit{Obj} \cdot \exists st \in P^S_\Sigma, x \in \mathit{ObjId}, st' \in \mathit{maxRep}(st, x) \cdot st' \Downarrow_{\equiv_{eq}} x = \{rt \downarrow o\}$$

Proof.

Given $rt \in P^R_\Sigma$ and $o \in \mathit{Obj}$
(by Assumption. 1)
$\implies rt \downarrow o \in P^S_\Sigma \Downarrow \mathit{ObjId}$
(by definition of $\Downarrow$)
$\implies \exists x \in \mathit{ObjId}, st \in P^S_\Sigma \cdot rt \downarrow o \in st \Downarrow x$

(by Proposition. **??**)

$\implies \exists st \in P_\Sigma^S, x \in \mathit{ObjId}, st' \in \mathit{maxRep}(st, x) \cdot rt \downarrow o \in st' \Downarrow x$

(by Proposition. 8)

$\implies \exists st \in P_\Sigma^S, x \in \mathit{ObjId}, st' \in \mathit{maxRep}(st, x) \cdot st' \Downarrow_{\equiv_{eq}} x = \{rt \downarrow o\}$

□

Therefore, given a runtime trace and an object, it's behaviour is represented statically by some static trace, specifically by at least one static trace in the maximal set of another. We denote a mapping from runtime traces to such a static trace representing it by *staticRep* : $(RT \times Obj) \rightarrow ST$. Therefore, there is a set of *productive* maximal representatives of static traces that produce the program at runtime, namely the range of *staticRep*. It follows easily that satisfaction of each object's behaviour with respect to a trace is equivalent to the correspondent representative static trace, with respect to a property.

Proposition 10. *An object's behaviour at runtime satisfies a property if and only if it's static representative satisfies the property statically.*

$$rt \downarrow o \Vdash \pi \iff \mathit{staticRep}(rt, o) \Downarrow_{\equiv_{eq}} x \Vdash \pi$$

Proof.

$rt \downarrow o \Vdash \pi$

(by Theorem. **??**)

$\iff \exists st \in P_\Sigma^S, x \in \mathit{ObjId}, st' \in \mathit{maxRep}(st, x) \cdot st' \Downarrow_{\equiv_{eq}} x \Vdash\!\Vdash \pi$

(by definition of *staticRep*)

$\iff \mathit{staticRep}(rt, o) \Downarrow_{\equiv_{eq}} x \Vdash\!\Vdash \pi$

□

Considering transformations of a static program, we now characterize their effects on the concrete program. Given that we have already identified a set of maximal representatives that together produce the program at runtime, then after a transformation (that does not change the semantics of a program, but only possibly that of monitoring) the new productive set is the old productive set with the transformation applied to each of its members. We denote the new program at runtime by $A(P_\Sigma^R)$. We can characterize it by the transformation of only all the productive maximal representatives of a static program.

Definition 12. *A representative isomorphic transformation is said to be isomorphic at runtime if each maximal representative of the transformed program at runtime corresponds to a transformation of an original maximal representative.*

$$\forall rt \in A(P_\Sigma^R) \cdot \forall o \in \mathit{Obj} \cdot \exists rt' \in P_\Sigma^R \cdot \mathit{staticRep}(rt, o) = \alpha(\mathit{staticRep}(rt', o))$$

Given this, we can show that equivalence with respect to a property between a statically parametrized program and its transformation implies equivalence with respect to a property of the original program at runtime and the program generated after the transformation.

Theorem 2. *Given a representative isomorphic transformation that is also isomorphic at runtime, if it preserves the verdict of the original statically parametrized program then it also preserves the verdict with respect to the program at runtime.*

$$P^S_\Sigma \cong_\pi A(P^S_\Sigma) \implies P^R_\Sigma \Vdash \pi \iff A(P^R_\Sigma) \Vdash \pi$$

Proof.

$$\begin{aligned}
& P^S_\Sigma \cong_\pi A(P^S_\Sigma) \\
\implies & \forall st \in P^S_\Sigma \cdot st \cong_\pi \alpha(st) \\
\implies & \forall st' \in maxRep(st, x) \cdot st \Vdash\!\!\vdash_{\equiv} \pi \iff \alpha(st) \Vdash\!\!\vdash_{\equiv_{eq}} \pi
\end{aligned}$$

$$\begin{aligned}
& P^R_\Sigma \Vdash \pi \\
& \text{(by definition of } \Vdash\text{)} \\
\iff & \forall rt \in P^R_\Sigma \cdot \forall o \in Obj \cdot rt \downarrow o \Vdash \pi \\
& \text{(by Prop. \textbf{??})} \\
\iff & \forall rt \in P^R_\Sigma \cdot \forall o \in Obj \cdot staticRep(rt, o) \Downarrow x \Vdash\!\!\vdash \pi \\
& \text{(by premise)} \\
\implies & \forall rt \in P^R_\Sigma \cdot \forall o \in Obj \cdot \alpha(staticRep(rt, o)) \Downarrow x \Vdash\!\!\vdash \pi \\
& \text{(by Definition. \textbf{??})} \\
\implies & \forall rt' \in A(P^R_\Sigma) \cdot \forall o \in Obj \cdot staticRep(rt', o) \Downarrow x \Vdash\!\!\vdash \pi \\
& \text{(by Prop. \textbf{??})} \\
\implies & \forall rt' \in A(P^R_\Sigma) \cdot \forall o \in Obj \cdot rt' \downarrow o \Vdash \pi \\
& \text{(by definition of } \Vdash\text{)} \\
\implies & A(P^R_\Sigma) \Vdash \pi
\end{aligned}$$

□

Based on the notions presented in the previous sections, we can now discuss the analysis techniques used in Clara [BLH12], where each of its analyses reduces the points in a program that activate the monitor at runtime. The basic thesis of Clara is then that appropriate silencing of certain events, does not affect satisfiability of the program with respect to the property but reduces the length of the traces to be analysed: Given a static approximation of

a program $P = P^S_\Sigma$ and property π, the reduced program approximation obtained through Clara, $P' = \mathit{Clara}(P^S_\Sigma, \pi)$, is sound with respect to π: $P \cong_\pi P'$.

Clara uses three analysis techniques to reduce the program approximation [BLH12]:

Quick Check. Some events specified by the property may not correspond to any method invocations by the program, e.g. consider that given Figure 1, a program may only open streams and write to them, but never read from them. Also, some events may only appear on loops in the same state, and therefore never cause a change in state (e.g. *lookAhead*). Clara's first analysis can be used to remove these kinds of events from the property, and the corresponding transitions. This may lead to some states becoming unreachable from the initial state, or states that cannot reach a bad state, and thus these can be removed. If a bad state cannot be reached from an initial state then the property is satisfied.

Orphan Shadows Analysis. The first analysis ignores the fact that events are parametrised. Consider a program, where only *open* and *lookAhead* are ever called on one object, then by looking at the property we can automatically note that this object can never violate it (by performing the first analysis on this object, instead of on the whole program), therefore both method calls can be silenced. This can be done for every object, producing a set of statements that can be disabled without affecting satisfaction of the property.

Flow-Sensitive Nop-Shadows Analysis. The first two analyses do not take into account any of the control-flow of the program, but they just consider which methods are invoked or not. This can be taken into account by considering the control-flow graph (CFG) of a program, which is a superset of its behaviour, and silence any statements that, if present or not, do not affect violation. Therefore, Bodden et. al. consider an over-approximation of a whole-program CFG and perform a synchronous composition of a property with such a CFG for each method (and considering the possible aliasing between objects), one can then determine which statements in the program (transitions in a CFG) never have an effect on violation (which Bodden et. al. call Nop shadows), meaning they can be silenced, reducing the amount of times the monitor is triggered at runtime.

As an example, consider the synchronous composition of the property and approximated CFG in Figure 2, and assume that all the object identifiers (s_i) associated with an event *may-alias* with each other. One can then note that after q_1, the synchronous composition is either in q_a or q_c; then taking $q_1 \xrightarrow{s_3.\mathit{open}} q_2$ will lead to q_b, and then $q_2 \xrightarrow{s_4.\mathit{close}} q_3 \xrightarrow{s_5.\mathit{open}} q_4$ will necessarily lead to q_b. Since, then, these two transitions do not affect the control-flow, they can be disabled such that they do not activate the monitor at runtime. Note the loops at state q_4 represent the flattened behaviour of a method called at that state, while those at q_1 and q_5, the behaviour outside the method.

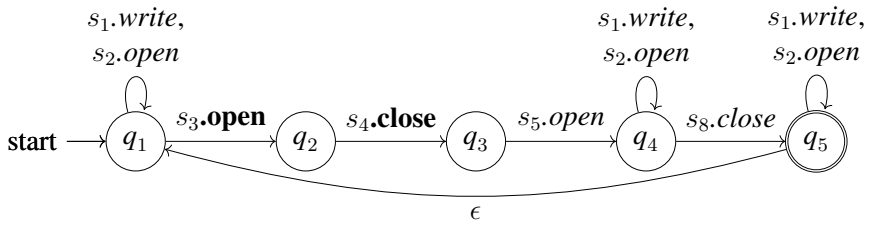

Figure 2: Example method CFG generalized to whole-method CFG.

3 Residual Control-Flow Analysis of DATEs

The properties considered by Clara are simple automata, however these can be extended to allow for more expressive properties. One such extension of automata is used in Larva [CPS09], which uses DATEs (Dynamic Automata with Timed Events). DATEs extend finite-state machine by enriching the transitions which trigger on *events*, with *conditional guards,* and *side-effect actions* which can affect monitoring state (e.g. by changing some variable local to the DATE). Note that in proper DATEs conditions can depend not just on the monitoring variables' state but also on the program state, however in this paper we limit the analysis to conditions on the monitoring state which is interesting by itself, since unfolding a DATE with such conditions can result in an infinite automaton (e.g. the non-regular language $a^n b^n$ can be represented with a finite DATE, but not with a finite-state machine). DATEs also include other extensions which we do not deal with in this paper, namely: (i) timers, in the same spirit as timed automata [AD94], but extended with stop-watches (e.g. event $c@5$ triggers when timer c reaches 5 seconds); and (ii) communication channels for communication between DATEs — with the actions taken when following a transition possibly involving sending of messages, and starting, stopping, or resetting of a timer. For full semantics of DATEs, refer to [CPS09, ACPS17]. Even without these extended features, properties in DATEs with conditions and actions on transitions are not, in general, amenable to the shadow effect analysis of Clara, since a sequence of Clara-detected NOP shadows that loop in the property can still have an effect on the monitoring state due to transitions' actions.

Consider the DATE shown in Figure 3. Transitions are labelled by a triple $e \mid c \mapsto a$ — when event e occurs and if condition c holds, the transition is taken, executing action a[4]. For instance, the top transition between states q_0 and q_1 triggers when a user is whitelisted and the monitoring variable *transferCount* is at least 3, and if taken resets this variable. Applying Clara to this property by ignoring the conditions and actions would result, for instance with the first analysis removing the *transfer* transition in state q_1 since upon a *transfer* the monitor would never change states. Clearly, taking this transition could have an

[4]We leave out the bar and arrow when the condition is true or the action is skip (the identity action).

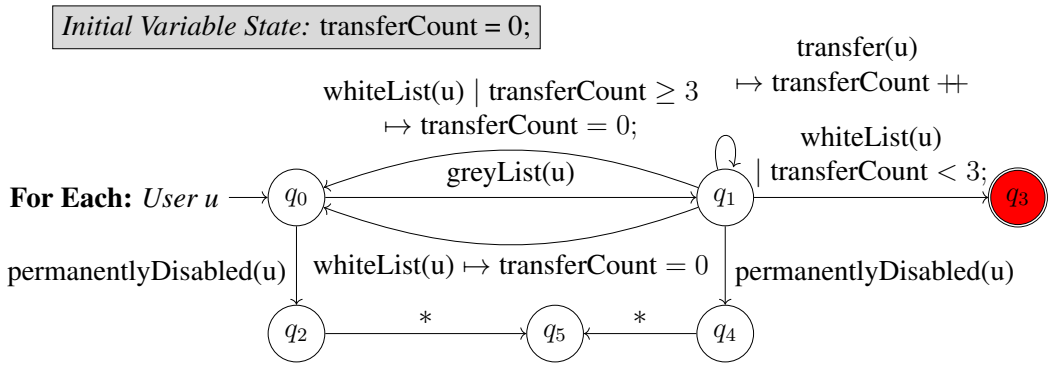

Figure 3: Example DATE specifying that once a user is greylisted they can only be whitelisted after performing three or more transfers.

effect on which future transitions are activated. For similar reasons, Clara's third analysis may disable transitions unsoundly.

3.1 Preliminaries

We start by identifying what we mean by a DATE in this paper, and continue exploring some notions and results we will need to present and prove our residual analysis correct.

Definition 13. *A DATE D is a tuple $\langle Q, \Sigma, \Theta, q_0, \theta_0, B, \delta \rangle$, where Q is the set of states, Σ is an alphabet of events, Θ is the type of monitoring variable states, $q_0 \in Q$ is the initial state, $\theta_0 : \Theta$ is the initial monitoring variable state, $B \subseteq Q$ is a set of bad states, and $\delta \subseteq Q \times \Sigma \times C \times A \times Q$ is a transition relation with conditions ($C = \Theta \to \mathbb{B}$) and actions ($A = \Theta \to \Theta$). We write $q \xrightarrow{e|c \mapsto a} q'$ for $(q, e, c, a, q') \in \delta$, skip for the identity action, and D for the type of DATEs.*

Property automata as defined in Section 2 can be seen as instances of DATEs with a transition $q \xrightarrow{e} q'$ being translated into $q \xrightarrow{e|true \mapsto skip} q'$.

Determinism of the transition relation of a DATE is typically desirable from a monitoring perspective for efficiency reasons. However, in the presence of actions, determinism is crucial since otherwise it would not be impossible to decide which actions to perform. To ensure determinism, events and conditions on transitions from a state must be mutually exclusive.

Assumption 2. *Given a DATE D, the transition relation is deterministic over events and conditions:*

$$q \xrightarrow{e|c \mapsto a} q' \wedge q \xrightarrow{e|c' \mapsto a'} q'' \wedge (\exists \theta \cdot c(\theta) \wedge c'(\theta)) \implies c = c' \wedge a = a' \wedge q' = q''$$

Thanks to this assumption, we can use the transition function in an applicative manner.

Definition 14. *The concrete transition function of a DATE $\delta \in (Q \times \Theta) \times \Sigma \to Q \times \Theta$ is defined over the DATE and monitoring variable state:*

$$\delta((q,\theta),e) \stackrel{\text{def}}{=} \begin{cases} (q', a(\theta)) & \textit{if } q \xrightarrow{e|c \mapsto a} q' \wedge c(\theta) \\ (q,\theta) & \textit{otherwise} \end{cases}$$

We will write $\delta^* \in (Q \times \Theta) \times \Sigma^* \to Q \times \Theta$ to denote the transitive closure of δ.

Note how this definition makes the DATEs semantics implicitly total with regards to events and conditions, since if there is no transition for a certain event (or for a certain event where a condition is never true) we remain at the same state. We do not make DATEs explicitly total for conciseness.

Statically we assume that we cannot know the resolution actions applied to a certain state. Therefore to reason about transitioning statically we define an over-approximation static transition function for DATEs, which takes into account conditions which are syntactically equivalent to true or false, but all others are taken to act completely non-deterministically.

Definition 15. *Given states $q, q' \in Q$ and event $e \in \Sigma$, we say that q* potentially goes to *q'* with event *e, written $q \xrightarrow{e}_{approx} q'$, if such a transition might be possible:*

$$\begin{aligned} q \xrightarrow{e}_{approx} q' \stackrel{\text{def}}{=} \; & (q \xrightarrow{e|c \mapsto a} q' \wedge c \neq \textit{false}) \vee \\ & (q = q' \wedge \nexists q'' \cdot q'' \neq q \wedge q \xrightarrow{e|\textit{true} \mapsto a} q'') \end{aligned}$$

Given a DATE, the static transition function *$\Delta_D \in 2^Q \times \Sigma \to 2^Q$ is defined to be the function which, given a set of states and an event returns the set of states potentially reachable from any of the input states: $\Delta_D(S,e) \stackrel{\text{def}}{=} \{q' \mid \exists q \in S \cdot q \xrightarrow{e}_{approx} q'\}$. We will write $\Delta_D^* \in 2^Q \times \Sigma^* \to 2^Q$ to denote its transitive closure, also writing $Q \stackrel{t}{\Rightarrow}_D Q'$ to denote that $\Delta_D^*(Q,t) = Q'$. Finally, we will write $q \hookrightarrow_D q'$ to denote that q' is reachable from q with the over-approximated transition relation: $q \hookrightarrow_D q' \stackrel{\text{def}}{=} \exists t \cdot q' \in \Delta_D^*(\{q\},t)$.*

We prove that Δ is really an over-approximation of δ.

Theorem 3. *The approximate transition function is an over-approximation of the concrete transition function:*

$\forall \theta : \Theta, t : \Sigma^*, q \in Q \cdot \exists \theta' : \Theta \cdot \delta((q,\theta'),t) = (q',\theta) \implies q' \in \Delta(\{q\},t)$.

Proof.

Case: $\exists (q,e,c,a,q') \in \delta \cdot c(\theta)$

$\delta((q, \theta), e)$
(by defn. of δ)
$= (q', a(\theta))$

$\Delta(\{q\}, e)$
(since case premise implies $c \neq$ *false*, which means $q \xrightarrow{e}_{approx} q'$,
and then by defn. of Δ)
$\ni q'$

As needed.

Case: $\nexists (q, e, c, a, q') \in \delta \cdot c(\theta) \wedge q \neq q'$
$\delta((q, \theta), e)$
(by defn. of δ)
$= (q, \theta)$

$\Delta(\{q\}, e)$
(by case premise $q \rightarrow_{approx} q$, and by defn. of Δ^*)
$= \{q\}$

As needed.

□

Corollary 1. *The transitive closure of the approximate transition function is an over-approximation of the transitive closure of the concrete transition function:*
$\forall \theta : \Theta, t : \Sigma^*, q \in Q \cdot \exists \theta' : \Theta \cdot \delta^*((q, \theta'), t) = (q', \theta) \implies q' \in \Delta^*(\{q\}, t)$.

Proof. Follows by induction on t, using Thm. 3. □

As we did for property automata, we define what it means for a DATE to satisfy the different kinds of traces using this static approximation of transitioning at runtime, and relate it to the static transition function.

Definition 16. *A ground trace $t \in \Sigma^*$ is said to satisfy a DATE D (with $\Sigma \subseteq \Sigma_D$), if none of its prefixes applied to the transitive closure of D, starting from the initial state of D and the*

initial monitoring variable state, lead to a bad state of D.

$$t \vdash D \stackrel{\text{def}}{=} \forall t' \in \textit{prefixes}(t) \cdot \delta^*((q_0, \theta_0), t') = (q, \theta) \wedge q \notin B_D$$

We define the satisfaction of a runtime parametrized trace and a statically parametrized trace as before with respect to to a DATE and over this ground trace satisfaction operator. Similarly we define the equivalence relations between DATEs and programs using this satisfaction with respect to a DATE.

Theorem 4. *If the static transition function applied to any prefix of t from the start state of D does not contain a bad state then t satisfies D.*

$$\forall t \in \Sigma^* \cdot \forall t' \in \textit{prefixes}(t) \cdot \Delta(\{q_0\}, t) \cap B_D \neq \emptyset \implies t \vdash D$$

Proof. This follows by considering, for contradiction, that $t \nvdash D$, and using Cor. 1. □

Looking at Figure 3 we can see that not all states are useful. Specifically consider how being at q_2, q_45 and q_5 implies that a trace will not violate, but however note that transitioning between q_2 and q_5 (and between q_4 and q_5) is useless, and thus q_5 is not useful since it is only reachable from states from which we can already conclude satisfaction. Thus note how we can reduce a property by keeping only states that are reachable from the initial state, and that can reach a bad state or that are a transition away from such states.

Definition 17. *A state q in DATE D is said to* possibly lead to a violation in *D, written badAfter(q), if it is reachable from the initial state and a bad state is reachable from it.*

$$\textit{badAfter}(q) \stackrel{\text{def}}{=} q_0 \hookrightarrow q \wedge \exists q' \in B \cdot q \hookrightarrow q'$$

A state q in DATE D is said to be immediately satisfying in *D, written goodEntryPoint(q), if it cannot possibly lead to a violation and is one transition away from such a state that can possibly lead to a violation.*

$$\textit{goodEntryPoint}(q) \stackrel{\text{def}}{=} \neg \textit{badAfter}(q) \wedge \exists q' \cdot q' \xrightarrow{e|c \mapsto a} q \wedge \textit{badAfter}(q')$$

A state q in DATE D is said to be useful in *D, written useful(q), if it can possibly lead to a violation or is satisfying in D.*

$$\textit{useful}(q) \stackrel{\text{def}}{=} \textit{badAfter}(q) \vee \textit{goodEntryPoint}(q)$$

Given a DATE D, its states and transitions can be reduced to the reachable useful ones to obtain $\mathcal{R}(D)$ which contains only states in D which are useful, and all the transitions between states that satisfy badAfter and their transitions to states that satisfy goodEntryPoint(q).

We can then show that DATE reduced for reachability is equivalent to the original DATE with respect to an approximation of a program, by using the theorems and propositions presented in the previous sections, which we claim also apply to DATEs by simply using the DATE satisfaction operators. We prove these in Section A.

Lemma 1. *The concrete transition function for a DATE and that for its reachability-reduction are equivalent from states both reachable from the initial state and that can reach a bad state.*

$$\textit{badAfter}(q) \implies \forall e : \Sigma, \theta : \Theta \cdot \delta_D((q,\theta),e) = \delta_{\mathcal{R}(D)}((q,\theta),e)$$

Proof. This follows directly from the definition of $\mathcal{R}(D)$, since it keeps all transitions outgoing from *badAfter* states. □

Lemma 2. *The concrete transition function for a DATE applied to a state that is immediately satisfying leads to a state that is not itself satisfying.*

$$\begin{aligned}\neg\textit{badAfter}(q) \implies &\forall e : \Sigma, \theta : \Theta \cdot \delta_D((q',\theta),e) = (q',\theta') \wedge q' \notin B \\ &\wedge \forall e : \Sigma, \theta : \Theta \cdot \delta_{\mathcal{R}(D)}((q',\theta),e) = (q'',\theta'') \wedge q'' \notin B\end{aligned}$$

Proof. Note how $\neg$*badAfter*(q) implies that $q' \notin B$ since if $q' \in B$, given that $q \hookrightarrow q'$, it would satisfy *badAfter*(q). Since $\mathcal{R}(D)$ has less transitions than D the result holds. □

Theorem 5. *A DATE D is equivalent with respect to its reachability-reduction $\mathcal{R}(D)$ (with alphabet Σ), with respect to any set of traces.*

$$\forall T \subseteq \Sigma^* \cdot D \cong_T \mathcal{R}(D)$$

Proof.

$$\text{To Prove} : \forall T \subseteq \Sigma^* \cdot \forall t : T \cdot t \vdash D \iff t \vdash \mathcal{R}(D)$$

By induction on t :

Base Case : $t = \langle\rangle$

$\langle\rangle \vdash D$

(by defn. of $\vdash$)

$\iff q_0 \notin B_D$

(by defn. of $\mathcal{R}(D)$ and since either *useful*(q_0) or $q \notin Q_{\mathcal{R}(Q)}$)

$\iff q_0 \notin B_{\mathcal{R}(D)}$

(by defn. of $\vdash$)

$\langle\rangle \vdash \mathcal{R}(D)$

Inductive Hypothesis: Given $t \in T \cdot t \vdash \boldsymbol{D} \iff t \vdash \mathcal{R}(D)$

Inductive Case: For $t' = t +\!\!+ \langle e \rangle$

$(\implies)$

$t +\!\!+ \langle e \rangle \vdash \boldsymbol{D}$

(by defn. of $\vdash$)

$\iff t \vdash \boldsymbol{D} \wedge \delta_D^*((q_0, \theta_0), t +\!\!+ \langle e \rangle) = (q, \theta) \wedge q \notin B_D$

(by the ind. hyp and the defn. of δ^*)

$\iff t \vdash \mathcal{R}(D) \wedge \delta_D(\delta_D^*((q_0, \theta_0), t), e) = (q, \theta) \wedge q \notin B_D$

(setting $\delta_D^*((q_0, \theta_0), t) = (q', \theta')$)

$\iff t \vdash \mathcal{R}(D) \wedge \delta_D((q', \theta'), e) = (q, \theta) \wedge q \notin B_D$

Case 1: *badAfter*(q')

(by Lemma. 1)

$\implies \delta_D((q', \theta'), e) = \delta_{\mathcal{R}(D)}((q', \theta'), e)$

As needed.

Case 2: $\neg$*badAfter*(q')

(by Lemma. 2)

$\implies \delta_{\mathcal{R}(D)}((q', \theta'), e) = (q'', \theta'') \wedge q'' \notin B_{\mathcal{R}(D)}$

As needed.

$(\impliedby)$

The same argument can be applied by replacing $\mathcal{R}(D)$ by D and vice-versa.

□

3.2 Residual Analysis

We are concerned with producing *residuals* of a DATE, where a residual, in general, represents a subset of the possible violating behaviour of a property with respect to a program. Residuals can thus be reductions of a property that remain equivalent to the original property with respect to some behaviour.

In this section we start by presenting some results which we shall use when presenting our extension of Clara, results which allow us to reason about when the reduction of DATEs

with respect to some alphabet is equivalent to the original DATE, in the context of some program's behaviour. We also introduce the notion of union between two reductions of a DATE, to allow the behaviour of multiple objects to be reasoned about individually but unified for the monitor at runtime.

First we discuss when an alphabet-reduction of DATE is equivalent to the original DATE, since we shall be using this operator in our analyses.

Lemma 3. *Given a DATE and some alphabet Σ, with $\Sigma \subseteq \Sigma_D$, then for any state and event $e \in \Sigma$ the application of the concrete transition function of the DATE is equal to the application with the concrete transition function of the DATE reduced by the alphabet Σ.*

$$\forall e \in \Sigma, \theta : \Theta, q \in Q_{D\restriction\Sigma} \cdot \delta_{D\restriction\Sigma}((q,\theta),e) = \delta_D((q,\theta),e)$$

Proof. This follows from the definition of alphabet-reduction, since any state in $D \restriction \Sigma$ has the same outgoing Σ-transitions in both $D \restriction \Sigma$ and D. □

We can extend this to the transitive closure of both concrete functions and all traces using the alphabet the DATE is reduced by.

Lemma 4. *Given a DATE, and a set of traces over its alphabet, then given such a trace the application with the concrete transition function of the DATE is equal to the application with the concrete transition function of the DATE reduced by the alphabet of the set of traces.*

$$\forall T \subseteq \Sigma_D^*, t \in T \cdot \delta^*_{D\restriction\Sigma(T)}((q_0,\theta_0),t) = \delta^*_D((q_0,\theta_0),t)$$

Proof. This follows by Lemma. 3 and induction on t. □

Theorem 6. *A DATE D with alphabet Σ and its alphabet-reduction by some alphabet $\Sigma' \subseteq \Sigma$ are equivalent with respect to a set of ground traces $T \subseteq \Sigma^*$.*

$$D \restriction \Sigma(T) \cong_T D$$

Proof. This follows directly from Lemma. 4. □

Different objects in a program activate different monitors, as we discussed. These objects may have different behaviour, and specifically they may use different subsets of the DATE alphabet. We will thus consider the residuals of a DATE, with respect to different parts of a program. We want to then combine these residuals into a single DATE that we can monitor the program with soundly and completely. We define this union between DATEs that are component-wise subsets of another DATE, otherwise it cannot be assured that the union produces a DATE.

Definition 18. *Given two DATES that are component-wise subsets of another DATE, then their component-wise union is defined as the property with both of their transitions, states and bad states, and starting from the same initial state.*

$$D \sqcup D' \stackrel{\text{def}}{=} \{Q_D \cup Q_{D'}, \Sigma_D \cup \Sigma_{D'}, q_0, \theta_0, B_D \cup B_{D'}, \delta_D \cup \delta_{D'}\}$$

We shall be producing residuals of a DATE by using the alphabet-reduction, then the reachability-reduction, and then performing the union on all these residuals. The next theorem shows that this union preserves the behaviour of the single reduced DATEs.

Proposition 11. *Given two sets of traces $T_0, T_1 \subseteq \Sigma^*$, and a DATE D then the union of D's alphabet-reduction with respect to each of the sets of traces is equivalent to D with respect to the union of the sets of traces.*

$$\mathcal{R}(D \upharpoonright \Sigma(T_0)) \sqcup \mathcal{R}(D \upharpoonright \Sigma(T_1)) \cong_{T_0 \cup T_1} D$$

Proof.

$$\begin{aligned}
\text{Given } & D' = \mathcal{R}(D \upharpoonright \Sigma(T_0)) \sqcup \mathcal{R}(D \upharpoonright \Sigma(T_1)) \\
\text{To Prove: } & \forall t : T_0 \sqcup T_1 \cdot t \vdash D' \iff t \vdash D \\
\text{By induction on } & t\text{:} \\
\text{Base Case: } & \langle\rangle \vdash D' \iff \langle\rangle \vdash D \\
& \langle\rangle \vdash D' \\
& (\text{by defn. of union and } \delta^*) \\
& = (q_0, \theta_0) \\
& (\text{by defn. of } \delta^* \text{ and} \\
& \text{since } q_0 \text{ is the initial state of D}) \\
& \langle\rangle \vdash D
\end{aligned}$$

$$\begin{aligned}
\text{Inductive Hypothesis: } & t \vdash D' \iff t \vdash D \\
\text{Inductive Case: } & \text{For } t' = t +\!\!+ \langle e \rangle \textit{ and w.l.g. assume } t' \in T_0 \\
\text{Proof: } & t +\!\!+ \langle e \rangle \vdash D \\
& (\text{by Thm. 6, Thm. 5, and defn. of } \vdash) \\
\iff & t +\!\!+ \langle e \rangle \vdash \mathcal{R}(D \upharpoonright \Sigma(T_0)) \wedge t \vdash D \\
& \wedge \delta_D(\delta_D^*((q_0, \theta_0), t), e) = (q, \theta) \wedge q \notin B \\
& (\text{by ind. hyp.}) \\
\iff & t +\!\!+ \langle e \rangle \vdash \mathcal{R}(D \upharpoonright \Sigma(T_0)) \wedge t \vdash D'
\end{aligned}$$

$$\wedge\, \delta_D(\delta^*_D((q_0,\theta_0),t),e) = (q,\theta) \wedge q \notin B$$

Suppose, for contradiction, that: $\delta_{D'}(\delta^*_{D'}((q_0,\theta_0),t),e) = (q',\theta') \wedge q' \in B$

(since we know $t \vdash D'$)
$\delta^*_{D'}((q_0,\theta_0),t) = (q'',\theta'') \wedge q'' \notin B$
(by defn. of δ)
$\implies \exists q'' \xrightarrow{e|c\mapsto a} q' \cdot q' \in B$
(since one violates and the other does not,
and by definition of union)
$\implies \delta^*_{D'}((q_0,\theta_0),t) \neq \delta^*_D((q_0,\theta_0),t)$
(since D' and D start from the same state)
$\implies \exists t' +\!\!+ \langle e' \rangle \in \textit{prefixes}(t) \cdot$
$\delta^*_D((q_0,\theta_0),t') = \delta^*_{D'}((q_0,\theta_0),t' = (q''',\theta''')$
$\wedge\, \delta_D((q''',\theta'''),e') \neq \delta_{D'}((q''',\theta'''),e')$
(by defn. of δ
and since D' is a component-wise subset of D)
$\implies \exists d = (q''',e',c,a,q'''') \in \delta_D \wedge d \notin \delta_{D'}$
(by defn. of union)
$\implies d \notin \delta_{\mathcal{R}(D\restriction\Sigma(T_0))} \wedge d \notin \delta_{\mathcal{R}(D\restriction\Sigma(T_1))}$
(by defn. of the reachability reduction)
$\implies \neg \textit{badAfter}_{\mathcal{R}(D\restriction\Sigma(T_0))}(q''')$
$\wedge\, \neg \textit{badAfter}_{\mathcal{R}(D\restriction\Sigma(T_1))}(q''')$
(by defn. of *badAfter*)
$\implies \nexists q_b \in B \cdot q''' \hookrightarrow_{\mathcal{R}(D\restriction\Sigma(T_0))} q_B$
$\wedge\, \nexists q_b \in B \cdot q''' \hookrightarrow_{\mathcal{R}(D\restriction\Sigma(T_1))} q_B$
(by defn. of union)
$\implies \nexists q_b \in B \cdot q''' \hookrightarrow_{D'} q_B$
※

□

Given these properties of DATEs we can now move on to presenting our constructions of residuals given a program.

3.3 Residual Constructions

As opposed to Clara, where the analyses are plugged in after instrumentation and function to turn off certain instrumentation points, we want our analysis to be more general. Thus here we formally describe analyses which both prune the property by removing transitions and states that are irrelevant for the program's violation, and silence statements that the analysis concludes will not affect violation.

We present three residuals, one without the events used by the program, another taking into account whether events can occur on the same object, and the last one removing from the DATE transitions that can never be used by a trace in the program. Theoretically, using the last analysis is enough, since each analysis is finer than the other, however practically one may want to use the other analyses since they are cheaper to compute. We prove that each of these is equivalent to the original DATE with respect to the given program. The last two analyses also present us with the opportunity to identify statements in the program that can be silenced safely, as we shall see.

3.3.1 Pruning non-occurring events

Recall that Clara's first analysis computed the set of symbols that need to be monitored, which did not include: (i) events that appeared only on transitions looping in the same state, (ii) symbols that do not appear in the program, and (iii) symbols only outgoing from states from which a bad state is not reachable in the property reduced by the previous types of symbols. We now consider these in the case of DATEs.

We cannot remove transitions such as in (i) since such idempotent transitions may perform actions which effect the triggering of other transitions (consider the looping transition on state q_3 in Figure 3). Those of type (ii) can be removed safely, since if a certain symbol does not appear in the program then clearly transitions tagged by such symbols in the property will never be taken and can be removed. While our reachability-reduction takes care of events of type (iii)[5].

We thus define $\mathit{residual}_0$ over a property D, with respect to a program P^S_Σ.

$$\mathit{residual}_0^P(D) \stackrel{\text{def}}{=} \mathcal{R}(D \restriction \Sigma(P^S_\Sigma))$$

We can then show easily that this is equivalent to the original DATE with respect to the used program.

[5]Note that this would not be always safe if we were considering multiple DATEs executing at the same time

Theorem 7. *The residual$_0$ of a DATE D is equivalent to D, with respect to program approximation P^S_Σ (with $\Sigma \subseteq \Sigma_{residual_0(D)}$).*

$$residual_0^P(D) \cong_{P^S_\Sigma} D$$

Proof. This clearly follow from Thm. 5, Thm. 6, and Prop. 7. □

3.3.2 Pruning transitions never usable on the same object

Like Clara, we can generalise the first analysis to consider events occurring on objects. Given an object, if the traces corresponding to it do not use the full alphabet of the DATE, then we can create a residual for the object's traces, which is enough to monitor that object soundly and completely. Consider Figure 3, if we have an object that we know is never greylisted but performs transfers then we can keep only the transitions triggered upon a transfer.

$$residual_1^P(D, oid) \stackrel{\text{def}}{=} \mathcal{R}(D \upharpoonright \Sigma(P^S_\Sigma \Downarrow oid))$$

Proposition 12. *The residual$_1$ with respect to an object identifier oid is equivalent to the original property D with respect to the possible traces of oid in the program P^S_Σ.*

$$residual_1^P(D, oid) \cong_{P^S_\Sigma \Downarrow oid} D$$

Proof. By Thm. 5 and Thm. 6. □

For runtime monitoring we now have two choices: (1) create a different monitor for each object identifier, corresponding to the associated residual, such that the monitor only instruments the statements associated with the identifier; (2) perform the union on all the residual DATEs and instrument the program as usual (i.e. by simply matching the DATE events with statements). The former would require new instrumentation techniques that employ static aliasing knowledge, while with the latter one could use existing techniques. We show that given the second choice, the resulting DATE is still equivalent to the original one, with respect to the program.

Theorem 8. *The union of each residual of a DATE D, associated with each object identifier, is equivalent to D with respect to the static approximation P^S_Σ.*

$$\bigsqcup_{oid \in ObjId} residual_1^P(D, oid) \cong_{P^S_\Sigma} D \tag{1}$$

Proof. By Prop. 13 and Prop. 12 clearly the DATEs are equivalent with respect to the ground traces generated by the static approximation of the program, and using Prop. 7 the theorem follows. □

Moreover, consider again an object that only performs transfers, with respect to it Figure 3 would be reduced to only the initial state, and thus we can say statically that the object satisfies the property. In this manner we no longer need to monitor this particular object when it performs a transfer, and thus any transfers associated solely with it can be silenced. We identify such instrumentation points with no effect on the monitor verdict, when given a set of statically parametrized traces $ST : \mathbb{\Sigma}^*_{ObjId}$, and an instrumentation point function $\mathcal{I} \in ObjId \to InstrPnt$, by identifying the instrumentation points associated with an object identifier whose associated event does not occur in the identifier's residual.

$$noEffect(ST, \mathcal{I}, D) \stackrel{\text{def}}{=} \{\mathcal{I}(oid) \mid \forall st \in ST \cdot (\exists n : \mathbb{N} \cdot st(n) = (oid, e)) \implies \nexists (q, e, c, a, q') \in \delta_{residual_1^P(D,oid)}\}$$

Theorem 9. *In general, given a statically parametrized program, and its reduction through silencing instrumentation points in noEffect are equivalent with respect to the second residual.*

$$silence(P^S_\Sigma, noEffect(P^S_\Sigma, \mathcal{I}, D)) \equiv_{\bigsqcup_{oid \in ObjId} residual_1^P(D,oid)} P^S_\Sigma$$

Proof. Given that statically parametrized events are only disabled if they never trigger any transition in their respective residual (see definition of *noEffect*), removing them does not affect the control-flow with respect to the DATE.
We shall consider only one static trace, and one instruction pointer being silenced, such that only one (oid, e) pair is removed, the result follows by considering every other pair being silenced, and by generalising it for sets of static traces.

Consider $st \in \mathbb{\Sigma}^*_{ObjId}$ which contains (oid, e) at some index, and $\mathcal{I}(oid) \in noEffect(ST, \mathcal{I}, D)$. And consider $st \Downarrow oid$ and recall that the $\equiv$ relation is reflexive, such that $oid \equiv oid$, which implies:

$$\forall t \in st \Downarrow oid, \exists t', t'' \cdot t = t' +\!\!+ \langle e_{oid} \rangle +\!\!+ t''$$

with the event e_{oid} being created by the projection when considering (oid, e) in the trace. If we silence this from st, resulting in st', by definition of silencing the generated traces correspond to the previous ones without the generated e:

$$\forall t \in st' \Downarrow oid', \exists t', t'' \cdot t = t' +\!\!+ t'' \wedge t' +\!\!+ \langle e_{oid} \rangle +\!\!+ t'' \in st \Downarrow oid$$

We will write D' for $\bigsqcup_{oid \in ObjId} residual_1^P(D, oid)$. We shall consider transitioning over D' to show the result.

To Prove: $t' +\!\!+ \langle e_{oid} \rangle +\!\!+ t'' \vdash D' \iff t' +\!\!+ t'' \vdash D'$

Proof: Consider $\delta^*_{D'}((q_0, \theta_0), t') = (q, \theta)$

Now consider $\delta_{D'}((q, \theta), e_{oid}) = (q', \theta')$

Case 1: $\nexists(q, e_{oid}, c, a, q') \in \delta_{D'} \implies q = q'$

From which the result follows.

Case 2: $\exists d = (q, e_{oid}, c, a, q') \in \delta_{D'}$

By theorems 6 and 12 we can use interchangeably $D \restriction (\Sigma(P^S_\Sigma \Downarrow \mathit{oid}))$ or D'

$\exists d = (q, e_{oid}, c, a, q') \in \delta_{D'}$

(by definition of e_{oid}: $d \notin \delta_{\mathit{residual}^P_1(D,oid)}$)

$\iff \neg \mathit{badAfter}_{D\restriction(\Sigma(P^S_\Sigma \Downarrow oid))}(q')$

(by definition of *badAfter* and reachability-reduction)

$\implies \nexists q_b \in B \cdot q' \hookrightarrow q_b$

(since after reaching q, $t' +\!\!+ \langle e_{oid}\rangle t''$ can no longer violate)

$\implies t \nvdash D' \iff t' \nvdash D'$

(Taking the contrapositive.)

$\implies t \vdash D' \iff t' \vdash D$

From which the result follows.

□

Consider also $st \Downarrow \mathit{oid}'$, where $oid \equiv_{\mathit{may}} \mathit{oid}'$ (note if they are not related by any of the two relations the result follows immediately), of which we can characterise its generated ground traces as follows:

$$\forall t \in st \Downarrow \mathit{oid}', \exists t', t'' \cdot \quad t = t' +\!\!+ \langle e_{oid}\rangle +\!\!+ t'' \wedge t' +\!\!+ t'' \in st \Downarrow \mathit{oid}' \\ \oplus \quad t \neq t' +\!\!+ \langle e_{oid}\rangle +\!\!+ t''$$

Given the definition of silencing then we can come up with a correspondence between ground traces generated by *st* and *st*′:

$$\forall t \in st' \Downarrow \mathit{oid}', \exists t', t'' \cdot \quad t \in st \Downarrow \mathit{oid}' \\ \oplus \quad t = t' +\!\!+ t'' \wedge t' +\!\!+ \langle e_{oid}\rangle +\!\!+ t'' \in st \Downarrow \mathit{oid}'$$

The result clearly follows from this, since the ground traces generated by *st*′ are a subset of those generated by *st*′, by the two previous statements. □

3.3.3 Removing unusable transitions

For the previous two analyses we have assumed a static approximation of the program, but just considered the used alphabet, ignoring the way flow of the events, which makes

these analyses relatively cheap to compute. This third analysis, however, makes use of the control-flow of the program, by considering the possible traces of a program.

Consider Figure 3, and the trace *whitelist*; *greylist*; *transfer*^{n}. Given the previous two analyses, only the transitions between states q_0, q_1 and q_3 would remain. However, clearly the *whitelist* transitions from q_1 can never be activated, since the trace only performs *whitelist* before the user is greylisted. Thus we can remove these *whitelist* transitions.

In general, given a trace, this does not always trigger all the transitions in a DATE, in fact we can prune away the unused transitions and the residual DATE will remain equivalent to the original one with respect to that trace. We can generalize this notion by, when given a set of traces, removing transitions that cannot be used by any of the traces. We use the static transition function to simulate the concrete one, since statically we do not have access to the monitoring variable state.

$$\textit{residual}_2^P(\boldsymbol{D}, P_\Sigma^S) \stackrel{\text{def}}{=} \mathcal{R}(\boldsymbol{D} \restriction \{q \xrightarrow{e|c \mapsto a} q' \mid \exists t : P_\Sigma^S \Downarrow \textit{ObjId}, t' +\!\!+ \langle e' \rangle \in \textit{prefixes}(t) \cdot q \in \Delta_D^*(\{q_0\}, t') \wedge e' = e\})$$

Theorem 10.

$$\textit{residual}_2^P(\boldsymbol{D}, P_\Sigma^S) \cong_{P_\Sigma^S} \boldsymbol{D}$$

Proof.

To Prove: $\forall t \in P_\Sigma^S \Downarrow \textit{ObjId} \cdot t \vdash R \iff t \vdash \boldsymbol{D}$
($\textit{with } R = \textit{residual}_2^P(\boldsymbol{D}, P_\Sigma^S)$)

By induction on t:

Base Case: $t = \langle\rangle$
Follows immediately since both DATEs start from the same initial state

Inductive Hypothesis: $t \in P_\Sigma^S \Downarrow \textit{ObjId} \cdot t \vdash \textit{residual}_2^P(\boldsymbol{D}, P_\Sigma^S) \iff t \vdash \boldsymbol{D}$

Inductive Case: $t' = t +\!\!+ \langle e \rangle$

Proof: $t +\!\!+ \langle e \rangle$
(by defn. of $\vdash$)
$\iff t \vdash D \wedge \delta_D(\delta_D^*((q_0, \theta_0, t), e) = (q, \theta) \wedge q \notin B$
(by ind. hyp)
$\iff t \vdash R \wedge \delta_D(\delta_D^*((q_0, \theta_0, t), e) = (q, \theta) \wedge q \notin B$

Suppose, for contradiction, that: $\delta_R(\delta_R^*((q_0, \theta_0, t), e) = (q', \theta') \wedge q' \in B$

$$
\begin{aligned}
&\text{(since } R \text{ and } D \text{ start from the same initial state)} \\
&\exists t' +\!\!+ \langle e' \rangle \in \textit{prefixes}(t) \\
&\cdot\ \delta_D^*((q_0, \theta_0, t) = \delta_R^*((q_0, \theta_0, t) = (q'', \theta'') \\
&\wedge\ \delta_D^*((q_0, \theta_0, t) \neq \delta_R^*((q_0, \theta_0, t) \\
&\text{(since R is a component-wise subset of D)} \\
\implies &\exists d = (q'', e', c, a, q''') \in D \wedge d \notin R \\
&\text{(by defn. of R)} \\
\implies &\neg \textit{badAfter}(q'') \\
&\quad \vee \nexists t'' \in P_\Sigma^S \Downarrow \textit{ObjId}, t''' +\!\!+ \langle e' \rangle \in \textit{prefixes(t)} \\
&\qquad \cdot\ q'' \in \Delta_D^*(\{q_0\}, t'') \wedge e'' = e \\
&\quad \lightning \\
&\quad \text{since } \neg \textit{badAfter}(q'') \text{ implies that } q'' \text{ cannot lead} \\
&\quad \text{to a bad state, while } t' +\!\!+ \langle e \rangle \text{ would be} \\
&\quad \text{a candidate for the latter statement}
\end{aligned}
$$

□

Previously we discussed informally that multiple CFGs that approximate the program's behaviour can be created, meaning we may have multiple static over-approximations of a program, we thus can apply the *residual*$_2$ on each of these successively, creating a possibly finer residual than we can with one over-approximation.

$$
\begin{aligned}
\textit{residual}_2^P(D, \langle\rangle) &\stackrel{\text{def}}{=} D \\
\textit{residual}_2^P(D, P_\Sigma^S : Ps) &\stackrel{\text{def}}{=} \textit{residual}_2^P(\textit{residual}_2^P(D, P_\Sigma^S), Ps)
\end{aligned}
$$

Theorem 11. *Given a set of static program over-approximations, then applying the third residual construction on these, consecutively, returns a DATE that is equivalent to the original DATE, with respect to the intersection of projection into ground traces of each approximation.*

$$\textit{residual}_2^P(D, Ps) \cong_{\bigcap_{0<i<\textit{length}(Ps)} Ps(i)\Downarrow \textit{ObjId}_i} D$$

Proof. This follows from Thm. 10, Prop. 7, and Prop. 6. □

Note how Thm. 11 implies that the residual and DATE are equivalent to the program's behaviour at runtime, since by Assump. 1 each over-approximation's projection contains the runtime program..

Clara's third analysis silenced statements in the program that together do not have any effect on the flow with respect to the property, however given conditions on transitions in DATEs we do not necessarily know if a transition will be activated or not, hence why the static transition of a DATE ranges over a set of states, so as to consider the possibility that a transition is both triggered and not. Using the same principle, we can apply Clara's third analysis to method invocations that only trigger DATE transitions without conditions (or rather with the *true* condition), and that do not have actions with side-effects. However we do not detail this here given space constraints.

4 Case Study

Our optimisation techniques have been applied to a simple financial transaction system with users connecting to a proxy in order to enable communication with a transaction server inspired by the industrial systems we have previously used runtime verification on [ACPV16]. The proxy and the transaction server can be two different services (the client and provider), possibly provided by different providers, with the property specifying the behaviour that the transaction server expects out of the proxy.

In order to see how our approach scales with increased system load and monitoring overhead, the system was evaluated with different numbers of users behaving in a controlled random manner, thus allowing for repeatability of the experiment.

The system was verified with respect to a specification constraining payment patterns based on the status of the user, e.g. a blacklisted user can only perform a payment if it has not exceeded a certain risk threshold. The risk level of a user is calculated by the monitor by checking that the companies the users deal with in general have transacted with users in good standing (i.e. that are not currently blacklisted or greylisted), with the number of such users in bad standing having an effect on the risk level of the user in question[6]. This part of the specification is shown in Figure 4.

The memory used and execution time of the unmonitored program was compared to that of the monitored one, and that of the system with monitoring optimised with the three analyses applied cumulatively. The experiment was run for different numbers of users, with three sample executions used to normalize differences between measurements.

Given that the monitors used in the experiment do not have state to keep track of, no measurable memory overheads were to be found in any of the experiment runs. However, as expected, monitoring induced substantial processing overheads, and the results can be seen in Figure 5. The results show that our optimisation techniques had a substantial impact,

[6]Note that caching such a calculation does not aid the monitor performance since the risk level changes with each transaction — also those not involving the user in question.

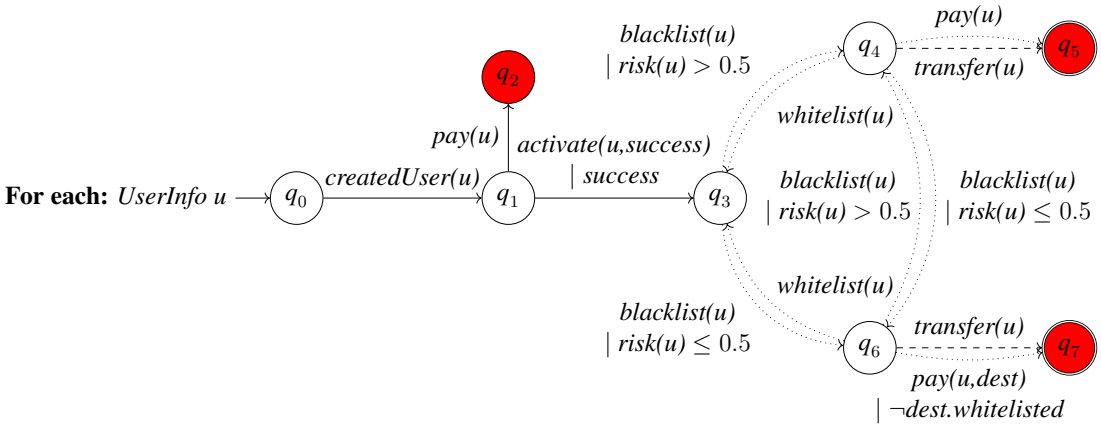

Figure 4: Property, with dashed transitions removed by the first analysis, and dotted by the second.

reducing overheads from an average of 97% to just 4%.

The first analysis took roughly the same time as the monitored time, mostly due to the fact that it only removed transitions that would never have been taken, resulting in the monitoring engine only bypassing a single conditional check for the never-activated event. The second analysis did not reduce the property itself, but turned off monitoring for all the statements in the program associated only with bronze users (which would have resulted in monitors being created at runtime for these users, but never change state). The third analysis used the program with instrumentation reduced by the second analysis and identified that blacklisted users are never allowed to affect a payment by the application (i.e. the program never makes it to q_5 and q_7 in Figure 4) and thus simply made sure that non-activated users do not affect payments (i.e. once a user was created, the monitor transitioned to q_1 and checked every incoming event against the only remaining outgoing transitions to q_2 and q_3). This resulted in an insignificant level of overheads, given the monitoring engine only had to check against two transitions (while in state q_2), without any expensive conditions to check and no tight-looping.

Similar to the results from [BLH12], the gains shown in the use case arise primarily since the system does not necessarily use all the events appearing in the property, and some of the correctness logic is encoded directly in the control-flow (as opposed to the data-flow) of the system. In our case, with a client-provider scenario we have: (i) the client does not make use of all the functions the provider allows (at least not for every possible user object in question); and (ii) the client program is coded in such a way that allows reasoning about its control-flow and not directly through data-flow e.g. blacklisting a user directly by setting a flag in the user's object. In practice, we envisage that this approach is applicable, for instance, when encoding properties over APIs or constraining server-access, allowing for monitoring overhead reduction in systems using the API or clients accessing the server.

No. of Users	Unmonitored	Monitored	After 1st	After 2nd	After 3rd
1000s	206s	371s	369s	295s	225s
1050s	231s	456s	445s	342s	235s
1100s	24s	450s	452s	314s	251s
1150s	254s	542s	544	400s	260s
1200s	268s	570s	562s	380s	280s
1250s	309s	576s	556s	413s	316s
1300s	316s	642s	657s	440s	330s
Average Overheads	0%	97.08%	95.94%	41.96%	4.13%

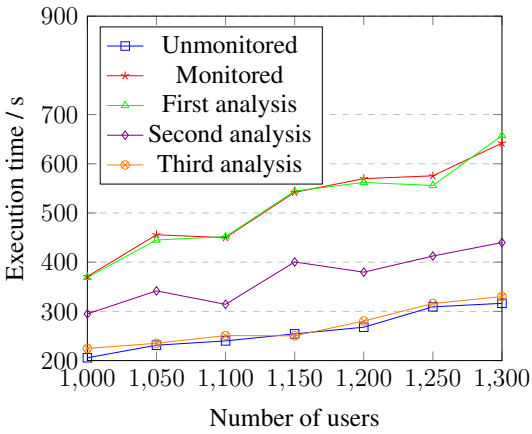

Figure 5: Table and plot of the experiment results.

5 Related Work

Our work builds directly on the results of Bodden et. al. [BLH12], but there are many other instances of the use of static analysis in order to optimise dynamic analysis. In [ACP16], we previously presented a theory of residuals, and model-based approach to combining static and dynamic analysis, under which this work falls except that here the model used (the CFG) is assumed to be a sound representation of the program.

Dwyer et. al. [DP07] take a different approach from ours or Clara's, wherein they identify safe regions in a program, i.e. sequences of statements that cannot violate a property, and if they are deterministic with respect to a property (if the monitor enters the region at a state q then it always exists at the same state q'). The effect of the region on the monitor is then replaced by a new unique event e, and the property augmented by a transition from q to q' with e. Note, that this summarises the effect of some instrumentation into one, wherein we simply remove instrumentation that does not effect violation.

Jin et. al. [JMGR11] also investigate parametric properties, and investigate optimisations which can be made to the implementation of monitoring logics at runtime, namely more efficient garbage collection of monitors associated with an object that has been garbage collected. It is worth noting how our second analysis may prevent some of this behaviour by detecting statically that an object may never violate and instead prevent the creation of its monitor.

6 Conclusions and Future Work

Through this work, we have extended static optimisation for the monitoring of parametric properties, in order to deal with automata which include a symbolic state in the form of DATEs. Thus, we can reduce both the property by using information about the system, and the program instrumentation by using information about the property. This static analysis depends purely on the control-flow of the program, and an aliasing relationship between relevant objects of the program, which we illustrated through an appropriate case study emulating a payment transaction system. These residual analyses have been implemented in a tool which we are currently preparing to make publicly available. We are also in the process of combining these results with those of StaRVOOrS [APS12], where in contrast to our approach, StaRVOOrS, static analysis is used to reduce the data-flow aspect of the specification (in the form of pre- and post-conditions), leaving the control-flow aspect (in the form of DATEs) for dynamic analysis. Our work is complementary to this approach, and in fact we are currently investigating how to optimise properties using both control and data flow static analysis.

References

[ACP16] Shaun Azzopardi, Christian Colombo, and Gordon Pace. *A Model-Based Approach to Combining Static and Dynamic Verification Techniques*, pages 416–430. Springer International Publishing, Cham, 2016.

[ACP17] Shaun Azzopardi, Christian Colombo, and Gordon Pace. Clarva - residual control-flow static analysis for finite-state properties with conditions and actions. In *The Computer Science Annual Workshop, CSAW'16*, 2017.

[ACPS15] Wolfgang Ahrendt, Jes¡C3¿¡BA¿s Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. A specification language for static and runtime verification of data and control properties. In *FM'15*, volume 9109. 2015.

[ACPS17] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Formal Methods in System Design*, pages 1–66, 2017.

[ACPV16] Shaun Azzopardi, Christian Colombo, Gordon J. Pace, and Brian Vella. *Compliance Checking in the Open Payments Ecosystem*, pages 337–343. Springer International Publishing, Cham, 2016.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[APS12] Wolfgang Ahrendt, Gordon Pace, and Gerardo Schneider. A Unified Approach for Static and Runtime Verification: Framework and Applications. In *ISOLA'12*, LNCS 7609. 2012.

[AY01] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, May 2001.

[BLH08] Eric Bodden, Patrick Lam, and Laurie Hendren. Object representatives: A uniform abstraction for pointer information. In *Proceedings of the 2008 International Conference on Visions of Computer Science: BCS International Academic Conference*, VoCS'08, pages 391–405, Swindon, UK, 2008. BCS Learning & Development Ltd.

[BLH12] Eric Bodden, Patrick Lam, and Laurie Hendren. Partially evaluating finite-state runtime monitors ahead of time. *ACM Trans. Program. Lang. Syst.*, 34(2):7:1–7:52, June 2012.

[CPS09] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. *Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties*, pages 135–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[DP07] Matthew B. Dwyer and Rahul Purandare. 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, 2007. ACM.

[JMGR11] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Rosu. Garbage collection for monitoring parametric properties. *SIGPLAN Not.*, 46(6):415–424, June 2011.

[LKRT07] Akash Lal, Nicholas Kidd, Thomas Reps, and Tayssir Touili. *Abstract Error Projection*, pages 200–217. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.

[PDE12] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. *Monitoring Finite State Properties: Algorithmic Approaches and Their Relative Strengths*, pages 381–395. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[VRCG+10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 214–224, Riverton, NJ, USA, 2010. IBM Corp.

A DATE Propositions

In this appendix we rehash pertinent parts of Section 2 for DATEs.

Definition 19. *A ground trace $t \in \Sigma^*$ is said to satisfy a DATE D (with $\Sigma \subseteq \Sigma_D$), if none of its prefixes applied to the transitive closure of D, starting from the initial state of D and the initial monitoring variable state, lead to a bad state of D.*

$$t \vdash D \stackrel{\text{def}}{=} \forall t' \in \mathit{prefixes}(t) \cdot \delta^*((q_0, \theta_0), t') = (q, \theta) \wedge q \notin B_D$$

We define the satisfaction of a runtime parametrized trace and a statically parametrized trace as before with respect to to a DATE and over this ground trace satisfaction operator.

$$\mathit{rt} \vdash D \stackrel{\text{def}}{=} \forall o \in \mathit{Obj}, t \in \mathit{rt} \Downarrow o, t' \in \mathit{prefixes}(t) \cdot t \vdash D$$
$$\mathit{st} \Vdash D \stackrel{\text{def}}{=} \forall \mathit{oid} \in \mathit{ObjId}, t \in \mathit{st} \Downarrow \mathit{oid}, t' \in \mathit{prefixes}(t) \cdot t \vdash D$$

And for sets of these.

$$\mathit{RT} \vdash D \stackrel{\text{def}}{=} \forall \mathit{rt} \in \mathit{RT} \cdot \mathit{rt} \Vdash D$$
$$\mathit{ST} \Vdash D \stackrel{\text{def}}{=} \forall \mathit{st} \in \mathit{ST} \cdot \mathit{st} \Vdash D$$

Using these definitions we can start proving the propositions we need for DATEs.

Proposition 13. *The runtime parametrised traces of a program P satisfy a typestate property if and only if the ground traces satisfy it:*

$$P^R_\Sigma \Vdash D \iff P^R_\Sigma \downarrow \mathit{ObjId} \vdash D$$

Proof. Follows immediately from Definitions 17, and 3. □

Proposition 14. *The static parametrised traces of a program P satisfy a typestate property if and only if the ground traces satisfy it:*

$$P^S_\Sigma \Vdash D \iff P^S_\Sigma \Downarrow \mathit{ObjId} \vdash D$$

Proof. Follows immediately from Definitions 17, and 9. □

Recall the assumption about correctness of statically parametrised programs: $P^R_\Sigma \downarrow \mathit{Obj} \subseteq P^S_\Sigma \Downarrow \mathit{ObjId}$.

This allows us to show that satisfaction of the static program implies that of the runtime program

Proposition 15. *Given program P and DATE D, if its static over-approximation of a program, P^S_Σ satisfies DATE D, then the runtime behaviour of the program, P^R_Σ also satisfies the property:*

$$P^S_\Sigma \Vdash\!\vdash D \implies P^R_\Sigma \Vdash D$$

Proof.

$$\begin{aligned}
& P^S_\Sigma \Vdash\!\vdash D \\
& \quad \text{(by Proposition 15)} \\
\implies & P^S_\Sigma \Downarrow \mathit{ObjId} \vdash D \\
& \quad \text{(by assumption that } P^R_\Sigma \downarrow \mathit{Obj} \subseteq P^S_\Sigma \Downarrow \mathit{ObjId}\text{)} \\
\implies & P^R_\Sigma \downarrow \mathit{Obj} \vdash D \\
& \quad \text{(by Proposition 14)} \\
\implies & P^R_\Sigma \Vdash D
\end{aligned}$$

□

We now overload the equivalence relations between programs and properties for DATEs.

Similarly we define the equivalence relations between DATEs and programs using this satisfaction with respect to a DATE.

Definition 20 (Equivalence)**.** *Two sets of ground traces, $T, T' \subseteq \Sigma^*$ are said to be* equivalent with respect to a DATE *D, written $T \cong_D T'$, if the verdict given with respect to one set of traces matches that of the other:*

$$T \cong_D T' \stackrel{\text{def}}{=} T \vdash D \iff T' \vdash D$$

This notion is lifted to parametrised and statically parametrised traces, and sets thereof, requiring equivalence up to object identifiers.

$$\mathit{RT} \cong_D \mathit{RT}' \stackrel{\text{def}}{=} \forall o \in \mathit{Obj} \cdot \mathit{RT} \downarrow o \cong_D \mathit{RT}' \downarrow o$$

$$\mathit{ST} \cong_D \mathit{ST}' \stackrel{\text{def}}{=} \forall \mathit{oid} \in \mathit{ObjId} \cdot \mathit{ST} \Downarrow \mathit{oid} \cong_D \mathit{ST}' \Downarrow \mathit{oid}$$

Two DATEs D and D' are said to be equivalent with respect to a set of ground traces T, written $D \cong_T D'$, if every trace in T is judged the same by either property:

$$D \cong_T D' \stackrel{\text{def}}{=} \forall t \in T \cdot t \vdash D \iff t \vdash D'$$

This notion is lifted to parametrised and statically parametrised traces, and sets thereof, requiring equivalence up to object identifiers.

$$D \cong_{RT} D' \stackrel{\text{def}}{=} \forall o \in \mathit{Obj} \cdot D \cong_{RT\downarrow o} D'$$
$$D \cong_{ST} D' \stackrel{\text{def}}{=} \forall \mathit{oid} \in \mathit{ObjId} \cdot D \cong_{ST\Downarrow \mathit{oid}} D'$$

It is straightforward to prove that equivalence up to traces is an equivalence relation.

Proposition 16. *$\cong_D$, $\cong_T$,$\cong_{RT}$, and $\cong_{ST}$ are all equivalence relations.*

Proof. This follows from the definition over ground traces using $\Longleftrightarrow$, Definition 18. □

Equivalence of traces is then preserved when reducing the set of traces.

Proposition 17. $\forall T' \subseteq T \cdot D \cong_T D' \implies D \cong_{T'} D'$

Proof. This follows from the definition of $\cong_T$ and $\vdash$, definitions 17 and 18. □

Given this, we can easily conclude the following proposition.

Proposition 18. *If $D \cong_T D'$ and $D' \cong_{T'} D''$, then $D \cong_{T \cap T'} D''$.*

We can also show that equivalence with respect to an approximation of a program can be expressed in terms of its projection onto ground traces.

Proposition 19. $D \cong_{P^S_\Sigma} D' \iff D \cong_{P^S_\Sigma \Downarrow \mathit{ObjId}} D'$.

Proof. This follows directly from Proposition 15. □