Control-Flow Residual Analysis for Symbolic Automata[*]

Shaun Azzopardi
shaun.azzopardi@um.edu.mt

Christian Colombo
christian.colombo@um.edu.mt

Gordon J. Pace
gordon.pace@um.edu.mt

Department of Computer Science, University of Malta, Msida, Malta

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 [8]). 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.

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. These 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 [17]. Traditionally, these two approaches have been seen as alternatives to each other, although their possible complementarity has started to be explored in recent years [3, 11, 13].

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 so that certain parts of the program are proved safe to not monitor. We call this *residual analysis*, with the pruned property called a *residual*. In language-theoretic terms, the language of a residual intersected with that of a program is equal to the language of the original property intersected with the program. This paper deals with the creation of such residuals in the presence of properties parametrised over different objects with different behaviour.

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

Adrian Francalanza, Gordon J. Pace (Eds.): Second International Workshop
on Pre- and Post-Deployment Verification Techniques (PrePost 2017)
EPTCS 254, 2017, pp. 29–43, doi:10.4204/EPTCS.254.3

In particular, Clara [11] is one such approach, acting on purely control-flow properties (without any consideration for data state) defined as automata with transitions triggered by method calls 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. 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 (e.g. one may want to talk about the value of a transaction after some sequence of events).

DATEs (Dynamic Automata with Events and Timers) [12] involve symbolic automata possibly running in parallel, with transitions triggered by either a method invocation or a timer event, and conditioned on a boolean expression on variables specific to the monitor and the program. Moreover, when triggered, a transition can also perform an action which may also affect the triggering of other transitions (e.g. increase some internal counter used by another transition's condition). In this paper, we do not handle timers and the dynamic creation of DATEs, and focus on extending Clara to automata with transitions guarded by conditions and actions as an initial step towards handling full DATEs. Even with this limitation, applying Clara directly to DATEs proves to be unsound, as we shall discuss informally, given transitions with side-effects may be removed (possibly changing the verdict of the reduced property).

Our contribution in this paper is two-fold: (i) extending the intuition behind Clara's first two analyses to produce both a residual DATE and a residual instrumentation of the program, and (ii) a novel analysis that uses a control-flow graph of a program to determine if any transitions in a DATE can (or can not) be reached by the program. In a case study, we show that each of these analyses can produce significant reduction in runtime overheads (down to 4% from an average of 97% of the original unmonitored run time), we also explain in which cases such results can be expected. Full proofs of the results presented in this paper can be found in [7].

In presenting our results, we first detail some formal preliminaries, using simple automata with events in Section 2 and use these to discuss Clara's analyses, while in Section 3 we present DATEs and define residuals over them, which we evaluate in Section 4. We discuss related work in Section 5, and conclude proposing future work in Section 6. Due to space restrictions, the proofs of the results presented in this paper are not included. However, they are available in a technical report available online [7].

2 Programs' Runtime Traces and Abstractions

In this section we look at safety properties over the control-flow of a program, written in the form of automata with transitions being triggered by the program[1]. We will build on this formalism in the rest of the paper.

Definition 2.1 (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 leave out π if it is clear from the context.

Finally, we will write $\pi \restriction \Sigma'$ to denote the property automaton identical to π except that the alphabet is restricted to Σ': $\Sigma_{\pi \restriction \Sigma'} \overset{\text{def}}{=} \Sigma'$ and $\delta_{\pi \restriction \Sigma'} \overset{\text{def}}{=} \{(q, e, q') \in \delta_{\pi} \mid e \in \Sigma'\}$); and $\pi \restriction \delta'$ as π with the transition relation restricted to δ': $\delta_{\pi \restriction \delta'} \overset{\text{def}}{=} \delta_{\pi} \cap \delta'$.

[1] It is worth noting, that although we will allow for branching on data values in our formalism, we do not trigger transitions over changes in data values, hence our characterisation of these properties as being over the control-flow of the program.

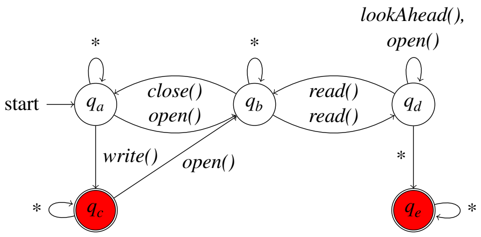

Figure 1: Property disallowing writing on a closed stream, and writing or closing while in the middle of an odd number of reads.

Consider that in monitoring we are concerned with points of interest during the execution of the program (e.g. when a method is called), which correspond to particular statements or regions in a program's source code. We use these corresponding program statements to trigger events at runtime to enable monitoring — and calling them *event generators*, since they generate property events. The problem of verification is then, given a property automaton and a program, to ensure that all program traces generated do not transition into a bad property state [16]. We call $t \in \Sigma^*$ 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 2.2 (Property satisfaction)**.** A ground trace $t \in \Sigma^*$ is said to satisfy property automaton π if no prefix of t leads to a bad state from the initial state: $t \vdash \pi \overset{\text{def}}{=} \forall t' \in \mathit{prefixes}(t) \cdot \nexists q_B \in B \cdot q_0 \overset{t'}{\Rightarrow} q_B$. 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[2]. 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.

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 [14] (or typestate automata [11]). 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[3] to which the event pertains. Note that we allow an alphabet to be parametrised by a set of identifiers α with: $\Sigma_\alpha \overset{\text{def}}{=} \alpha \times \Sigma$. Traces over such alphabets will be referred to as parametrised traces.

A trace contains events possibly related to different objects. To consider satisfaction of a property by one object we project the trace onto that object (assuming an equivalence relation between the objects).

Definition 2.3 (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 with identifiers equivalent to x, as follows [4]:

$$\begin{array}{lll}
\langle\rangle \downarrow x & \overset{\text{def}}{=} & \langle\rangle \\
((x',p) \colon rt) \downarrow x & \overset{\text{def}}{=} & \begin{cases} (x',p) \colon (rt \downarrow x) & \text{if } x \equiv x' \\ rt \downarrow x & \text{otherwise} \end{cases}
\end{array}$$

[2] Note that in this example there are outgoing transitions from bad states, although any trace that goes through a bad state is judged as violating. Clara's and our semantics allow this, since we may want to count the number of violations, or in the case of DATEs transitions may actually repair the violation (although we would still want to note it).

[3] 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.

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

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

At runtime, we have perfect knowledge of the equivalence relation between parametrised events. However, when using static analysis, this is not always possible. In the case of parametrisation by objects, several variable alias analyses exist, that can give partial information on whether two source code variables can point to the same runtime object (we use [10]). In such cases, we have three possible outcomes: (i) the two variables always refer to the same object; (ii) the two variables 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 (e.g. method calls) if their objects always (must) match, and a *may* relation $\equiv_{\mathsf{may}}$, which relates two event generators if their objects may match, with the former relation being a subset of the latter [5]. Thus, with such variables as static object identifiers, when statically we are given a trace of event-identifier couples, we can extract the possible runtime traces generated by these, through projection on certain identifiers. This is in contrast to parametrised traces where the behaviour of each object is perfectly known and thus only one ground trace is generated through projection.

Definition 2.4 (Static Parametrised Traces)**.** A static parametrised trace is a parametrised trace $st \in \Sigma_\alpha^*$, with two relations over α: (i) a must-alias equivalence relation $\equiv\ \in \alpha \leftrightarrow \alpha$, and (ii) a may-alias relation $\equiv_{\mathsf{may}} \in \alpha \leftrightarrow \alpha$, such that $\equiv\ \subseteq\ \equiv_{\mathsf{may}}$. We define the projection of a static parametrised trace st with respect to parameter x, written $st \Downarrow x$, as follows:

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

We overload this notation over sets of static parametrised traces: $ST \Downarrow x \overset{\text{def}}{=} \bigcup_{st \in ST} st \Downarrow x$; and over sets of identifiers: $ST \Downarrow X \overset{\text{def}}{=} \bigcup_{x \in X} ST \Downarrow x$.

Trace st is said to satisfy property automaton π , written $st \Vvdash \pi$, if for each identifier $x \in \alpha$ the projection of st onto x satisfies the property: $st \Vvdash \pi \overset{\text{def}}{=} \forall x \in \alpha \cdot st \Downarrow x \vdash \pi$.

It is worth noting that although we are only considering parametrisation over a single identifier (e.g. to an object), a property can be parametrized over multiple objects. The work presented here applies to that case, by abstracting tuples of identifiers into a single identifier, with point-wise must and may-alias relations.

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

Definition 2.5 (Programs)**.** For a program P over an alphabet Σ with a set of runtime objects Obj and static object identifiers $ObjId$, (i) we will write P_Σ^R to denote the set of parametrised traces over Obj i.e. $P_\Sigma^R \subseteq \Sigma_{Obj}^*$ with equivalence $\equiv$ over Obj; (ii) we will write P_Σ^S to denote the set of static parametrised traces over $ObjId$ i.e. $P_\Sigma^S \subseteq \Sigma_{ObjId}^*$ with relations $\equiv$ and $\equiv_{\mathsf{may}}$.

In what follows, we assume that the static parametrised trace generator is an over-approximation of the parametrised trace generator: $P_\Sigma^R \downarrow Obj \subseteq P_\Sigma^S \Downarrow ObjId$.

In our effort to reduce overheads by reducing a property, we will need to ensure that monitoring the program by the created residual is enough, i.e. both the original property and the residual give the same verdict for any given program trace, although their results may vary for other traces.

[5] Other analyses would be applicable in the case of parametrization by data instead of objects, and when this is not possible all identifiers can be related by the may relation soundly

Definition 2.6 (Equivalence)**.** 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: $\forall t \in T \cdot t \vdash \pi \iff t \vdash \pi'$. This is lifted to parametrised and static parametrised traces, and sets thereof.

It is straightforward to prove that equivalence up to traces is an equivalence relation and is preserved when reducing the set of traces, from which the next property holds.

Proposition 2.1. 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 the static parametrised trace generator can be expressed in terms of its projection onto ground traces.

Proposition 2.2. $\pi \cong_{P^S_\Sigma} \pi' \iff \pi \cong_{P^S_\Sigma \Downarrow ObjId} \pi'$.

As we have discussed, some parts of the program may be proved safe to unmonitor, upon which we can silence these parts and transform the static parametrised trace generator by turning off certain identifier-event pairs.

Definition 2.7. Given a static parametrised trace st, and a set of identifier-event pairs $E \in 2^{ObjId \times \Sigma}$, the silencing of st by E, written, $silence(st, ips)$, is defined to be the original trace st except for elements in E:

$$\begin{array}{lcl} silence(\langle\rangle, E) & \overset{\text{def}}{=} & \langle\rangle \\ silence((x,e) \colon st, E) & \overset{\text{def}}{=} & \begin{cases} silence(st, E) & \text{if } (x,e) \in E \\ (x,e) \colon silence(st, E) & \text{otherwise} \end{cases} \end{array}$$

We can also define equivalence between a program and its transformation (see [7]), however given lack of space we focus solely on equivalence between residuals and the original property in this paper, but define silencing since Clara

Based on the notions presented, we discuss the analysis techniques used in Clara [11], where each of its analyses reduces the points in a program that activate the monitor at runtime. The only inputs Clara needs is the source code of the program (which is then analysed using Soot [19]) and a property automaton. 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 parametrised program $P = P^S_\Sigma$ and property π, the reduced program approximation obtained through Clara, $P' = Clara(P^S_\Sigma, \pi)$, is sound with respect to π: $P \cong_\pi P'$.
Clara uses three analysis techniques to reduce the program approximation [11]:
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 also 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 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 then produce, for each object, a set of such instrumentation points that can be disabled without affecting the result of monitoring.
Flow-Sensitive Nop-Shadows Analysis. The first two analyses do not take into account any of the control-flow of the program, they just consider which methods are invoked or not but not in which order.

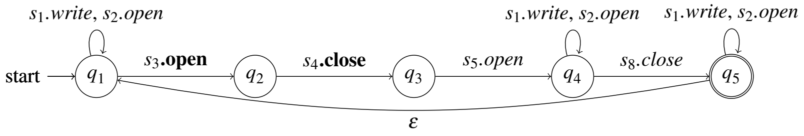

Figure 2: Example method CFG generalized to whole-method CFG.

This can be taken into account by considering a control-flow graph (CFG) of a program, which represents a superset of its event-triggering behaviour, and silence any statements that, if present or not, do not affect violation.

Through several over-approximations of a whole-program CFG and synchronous composition of these with the property, Bodden et al. identify sequences of instrumentation points that only ever transition from and to the same state (taking into account parametrisation of the property), with no bad states in between. Such points (which Bodden et al. call *nop shadows*) never have an effect on violation, 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 in Figure 1 and the approximated CFG in Figure 2, and assume that all the object identifiers (s_i) associated with an event always refer to the same event. One can then note that after q_1, the synchronous composition is either in q_a or q_c; then taking $q_1 \xrightarrow{s_3.open} q_2$ will lead to q_b, and then $q_2 \xrightarrow{s_4.close} q_3 \xrightarrow{s_5.open} q_4$ 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.

3 Control-Flow Residual Analysis of DATEs

The properties considered by Clara are automata with explicit state. However, some properties require a richer specification language — an extension to automata to deal with more expressive properties, are DATEs [12]. One way in which DATEs extend finite state automata is through the introduction of a symbolic state which can be checked and updated on transitions which trigger on *events*, with *conditional guards,* and perform *side-effect actions* affecting the symbolic monitoring state.[6]

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[7]. 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's first analysis to this property by ignoring the conditions and actions would result in 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 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 our residual analysis.

[6] DATEs also include other extensions which we do not deal with in this paper, such as timers and communication channels. For full semantics of DATEs, refer to [12, 2].

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

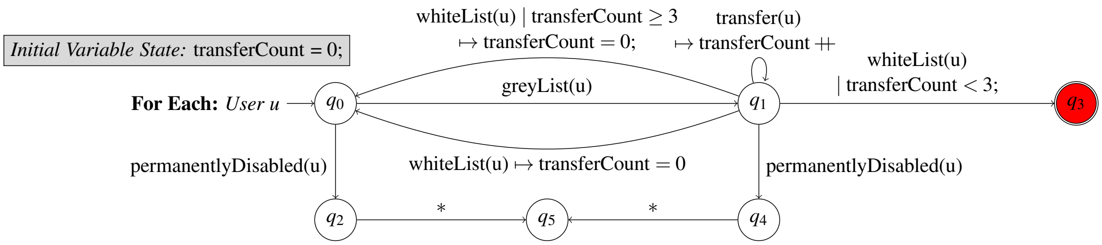

Figure 3: Example DATE specifying that once a user is greylisted they can only be whitelisted after performing three or more transfers.

Definition 3.1. 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 denote the type of DATEs by $\mathbb{D}$.

Property automata as defined in Section 2 can be seen as instances of DATEs with a transition $q \xrightarrow{e} q'$ being translated into $q \xrightarrow{e|true \mapsto \text{skip}} q'$. In our experience with DATEs, the number of states used is typically rather small, and while the symbolic state can be unbounded (e.g. includes lists), in practice such usage is rare and is usually delegated to a database.

We assume determinism of the transitions: from each state, transitions with the same events have mutually exclusive conditions, with regards to any monitoring variable state. Thanks to this assumption, we can use the transition function in an applicative manner.

Definition 3.2. 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:

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

We will write δ^* to denote the transitive closure of δ.

This definition makes the DATEs semantics implicitly total with regards to events and conditions, since if there is no transition for a certain event we remain at the same state.

Transitioning thus depends on the symbolic state, which we could approximate statically through some kind of code or predicate analysis. However here we shall take the maximal over-approximation, by always considering that a transition's condition may both be true or false. More precise approximations can be relatively more expensive to compute than than this, which is why we made the decision to focus solely on control-flow analysis, leaving the use of predicate analysis for future work. This means that we must consider transitioning into multiple states, which we cater for in the following definition.

Definition 3.3. 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 a transition with event e and a condition that can be satisfied, or q' is q and there is no outgoing transition out of it that must be taken on e being triggered:

$$q \xrightarrow{e}_{approx} q' \stackrel{\text{def}}{=} (q \xrightarrow{e|c \mapsto a} q' \wedge c \neq \mathit{false}) \vee (q = q' \wedge \nexists q'' \cdot q'' \neq q \wedge q \xrightarrow{e|true \mapsto a} q'')$$

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 use $\Delta_D^* \in 2^Q \times \Sigma^* \rightarrow 2^Q$ to denote its transitive closure . Finally, we use $q \hookrightarrow_D q'$ to denote that q' is reachable from q with the static transition relation: $q \hookrightarrow_D q' \overset{\text{def}}{=} \exists t \cdot q' \in \Delta_D^*(\{q\},t)$.

We can prove that given a state and a ground trace, then the state reached by δ^* is also possibly reached (with Δ^*) from that state, with that trace, i.e. Δ^* over-approximates δ^*.

Theorem 3.1. $\forall \theta : \Theta, t : \Sigma^*, q \in Q \cdot \exists \theta' : \Theta \cdot \delta^*((q,\theta'),t) = (q',\theta) \implies q' \in \Delta^*(\{q\},t)$.

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

We define the satisfaction of a parametrised trace and a statically parametrised trace as before with respect to a DATE and over this ground trace satisfaction operator, and similarly for the equivalence relations between DATEs.

Base on the previous theorem, we can show satisfaction of a trace by considering whether any of its prefixes possibly lead to a bad state.

Theorem 3.2. 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: $(0\forall t \in \Sigma^* \cdot \forall t' \in \mathit{prefixes}(t) \cdot \Delta(\{q_0\},t) \cap B_D \neq \emptyset) \implies t \vdash D$.

Moving on to creating residuals, from Figure 3 we can see that not all states are useful: consider how being at q_2, q_4 and q_5 implies that one can no longer violate the property. q_5 can however only be reached through q_2 or q_4 where the monitor verdict is clear, and thus q_5 is useless. Thus we should keep states q_2 and q_4 (since before reaching them we do not know the verdict of the trace), but we do not need q_5 (since satisfaction is clear). We then classify a state as useful if: (1) it is reachable from the initial state, and it can reach a bad state (q_1), or (2) it is reachable by one step from a state of the first kind (q_2 and q_4).

Definition 3.5. A state q in DATE D is said to *possibly lead to a violation in D*, written $\mathit{badAfter}(q)$, if it is reachable from the initial state and a bad state is reachable from it: $\mathit{badAfter}(q) \overset{\text{def}}{=} q_0 \hookrightarrow q \wedge \exists q' \in B \cdot q \hookrightarrow q'$.

A state q in DATE D is said to be *an entry-point to a satisfied region in D*, written $\mathit{goodEntryPoint}(q)$, if it cannot possibly lead to a violation and is one transition away from such a state that can possibly lead to a violation: $\mathit{goodEntryPoint}(q) \overset{\text{def}}{=} \neg \mathit{badAfter}(q) \wedge \exists q' \cdot q' \xrightarrow{e}_{\mathit{approx}} q \wedge \mathit{badAfter}(q')$.

A state q in DATE D is said to be *useful in D*, written $\mathit{useful}(q)$, if it can possibly lead to a violation or is an entry-point to a satisfied region in D: $\mathit{useful}(q) \overset{\text{def}}{=} \mathit{badAfter}(q) \vee \mathit{goodEntryPoint}(q)$. Given a DATE D, it can be reduced to the reachable useful states to obtain $\mathscr{R}(D)$ which contains only states in D which are *useful*, and the transitions between them: $\delta_{\mathscr{R}(D)} \overset{\text{def}}{=} \{(q,e,c,a,q') \in \delta \mid \mathit{useful}(q) \wedge \mathit{useful}(q')\}$.

We can then show that a 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.

Theorem 3.3. A DATE D is equivalent to its reachability-reduced counterpart $\mathscr{R}(D)$ (with alphabet Σ), with respect to any set of traces: $\forall T \subseteq \Sigma^* \cdot D \cong_T \mathscr{R}(D)$.

3.2 Residual Analysis

We are concerned with producing *residuals* of a DATE [6] with respect to some known information about the program — the part of the property which cannot be proved from what we know about the program. Recall that previously we informally characterized a residual D' of a DATE D, given a static program P^S_Σ over that same alphabet, by one property: the intersection of the program intersected with that of property and separately with the residual are equal: where $L(D)$ is the set of bad traces accepted by D, $L(D') \cap (P^S_\Sigma \Downarrow \mathit{ObjId}) = L(D) \cap (P^S_\Sigma \Downarrow \mathit{ObjId})$. This is equivalent to our notion of equivalence: $D' \cong_{P^S_\Sigma} D$. Note that the reachability-reduction just defined is a residual for all programs.

In this section, we start by presenting a number of definitions and results which we shall use to extend the analysis used in Clara to create DATE residuals.

One of the ways we shall be creating residuals is by restricting the alphabet of a DATE, the result of which is equivalent to the original DATE, with respect to some kind of traces.

Theorem 3.4. A DATE D with alphabet Σ and its alphabet-restriction by some alphabet $\Sigma' \subseteq \Sigma$ are equivalent with respect to a set of ground traces $T \subseteq \Sigma^*$: $D \upharpoonright \Sigma(T) \cong_T D$

Since we are looking at parametrised traces, different objects in a program activate different instances of the monitor. These objects may have different behaviour and thus they may use different subsets of the DATE alphabet. We will thus, in the next section, consider the residuals of a DATE with respect to different objects, however each of these residuals individually is not enough to monitor the whole program soundly with. We define a union operator over DATEs to allow this.

To ensure that the combination of DATEs (D and D') remains a DATE we require that they start from the same initial state and monitoring state, and that a DATE exists that captures both of their behaviour: $\exists D'' : \mathbb{D} \cdot Q_D \cup Q_{D'} \subseteq Q_{D''} \wedge \Sigma_D \cup \Sigma_{D'} \subseteq \Sigma_{D''} \wedge B_D \cup B_{D'} \subseteq B_{D''} \wedge \delta_D \cup \delta_{D'} \subseteq \delta_{D''}$, i.e. D and D' are said to be *component-wise subsets* of D''.

Definition 3.6. Given two DATES that are component-wise subsets of another DATE[8], 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: $D \sqcup D' \overset{\text{def}}{=} \{Q_D \cup Q_{D'}, \Sigma_D \cup \Sigma_{D'}, q_0, \theta_0, B_D \cup B_{D'}, \delta_D \cup \delta_{D'}\}$

We shall be producing residuals of a DATE by using the alphabet-restriction, 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.

Theorem 3.5. Given sets of traces $T_0, T_1 \subseteq \Sigma^*$, and DATE D, the union of D's alphabet-restriction with respect to each of the sets of traces is equivalent to D with respect to the union of the sets of traces: $\mathscr{R}(D \upharpoonright \Sigma(T_0)) \sqcup \mathscr{R}(D \upharpoonright \Sigma(T_1)) \cong_{T_0 \cup T_1} D$.

We can now move on to presenting our constructions of residuals.

3.3 Residual Constructions

We start by formally describing 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

[8] This ensures that the union, as defined here, is in turn merely the bigger DATE without some states and/or transitions, which is still a DATE, since determinism is preserved.

is enough, since each analysis is finer than the other, however in practice one may want to first 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 provide the opportunity of identifying statements in the program that can be silenced safely.

3.3.1 Absent event pruning

Recall that Clara's first analysis computes the symbols that should be monitored, thus excluding: (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 affect 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 already takes care of events of type (iii)[9]. We thus define *residual*$_0$ over a property D, with respect to a program P^S_Σ: $residual_0(D) \stackrel{\mathsf{def}}{=} \mathscr{R}(D \restriction \Sigma(P^S_\Sigma))$.

Based on Thm. 3.3, Thm. 3.4, and Prop. 2.2, we can show that this is equivalent to the original DATE with respect to the used program.

Theorem 3.6. The *residual*$_0$ of a DATE D is equivalent to D, with respect to program approximation P^S_Σ (with $\Sigma \subseteq \Sigma_{residual_0(D)}$): $residual_0(D) \cong_{P^S_\Sigma} D$

3.3.2 Object-specific absent event pruning

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 that is enough to monitor just that object soundly. 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. Based on the results given in Thm. 3.3 and Thm. 3.4, we define this as: $residual_1(D, objId) \stackrel{\mathsf{def}}{=} \mathscr{R}(D \restriction \Sigma(P^S_\Sigma \Downarrow objId))$.

Proposition 3.1. The *residual*$_1$, for an identifier *objId*, is equivalent to the original DATE D with respect to the traces of *objId* in the program P^S_Σ: $residual_1(D, objId) \cong_{P^S_\Sigma \Downarrow objId} D$.

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. Using Prop. 3.1, Prop. 3.5 and Prop. 2.2, we can show that the resulting DATE in the second choice is still equivalent to the original one, with respect to the program.

Theorem 3.7. Given the residual for each identifier, then their union is equivalent to the original DATE, with the program approximation: $\bigsqcup_{objId \in ObjId} residual_1(D, objId) \cong_{P^S_\Sigma} D$.

Consider again an object that only performs transfers. With respect to this, Figure 3 would be reduced to just the initial state using the second residual, and thus we can statically conclude that the object satisfies the property. In this manner, we no longer need to monitor this particular object when it performs

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

a transfer, and thus any transfers associated solely with it can be silenced. Using an object-specific residual, we can then reduce the traces in the static program by removing identifier-events pairs, where the event does not appear in the identifier's residual: $\mathit{noEffect}((\mathit{objId},e),D) \overset{\text{def}}{=} e \notin \Sigma_{\mathit{residual}_i(D,\mathit{objId})}$.

A static program without such pairs will remain equivalent to the original static program, with respect to that residual (i.e. corresponding traces in the respective programs will maintain the same verdict with respect to the residual)[10].

3.3.3 Unusable transition pruning

In the previous two analyses we have ignored the flow of events, in fact we consider only the alphabet of the program and DATE. Our novel third analysis, however, makes use of the control-flow, by considering the possible traces of a program.

Consider Figure 3, and the trace $\mathit{whitelist};\mathit{greylist};\mathit{transfer}^n$ as the program (over a single object). Given the previous two analyses, only transitions between states q_0, q_1 and q_3 would remain. However, 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 transitions. We define what it means for a trace to use a transition.

Definition 3.7. A trace is said to use a transition from q to q' with an event e, in a DATE D, if the transitions condition is not false, and a strict prefix of the trace can potentially go to q and the head of the remaining suffix is e: $\mathit{uses}(t, q \xrightarrow{e|c\mapsto a} q') \overset{\text{def}}{=} c \neq \mathit{false} \wedge \exists t' +\!\!+ \langle e \rangle \in \mathit{prefixes}(t) \cdot q \in \Delta^*_D(\{q_0\}, t')$

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 generalise this notion by removing transitions that cannot be used by any of the traces: $\mathit{residual}^P_2(D, P^S_\Sigma) \overset{\text{def}}{=} \mathscr{R}(D \upharpoonright \{d \in \delta \mid \exists t : P^S_\Sigma \Downarrow \mathit{ObjId} \cdot \mathit{uses}(t,d)\})$.[11]

Theorem 3.8. The $\mathit{residual}_2$ is equivalent to the original DATE: $\mathit{residual}_2(D, P^S_\Sigma) \cong_{P^S_\Sigma} D$

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

$$\begin{aligned}
\mathit{residual}_2(D, \langle\rangle) &\overset{\text{def}}{=} D \\
\mathit{residual}_2(D, P^S_\Sigma : Ps) &\overset{\text{def}}{=} \mathit{residual}_2(\mathit{residual}_2(D, P^S_\Sigma), Ps)
\end{aligned}$$

Theorem 3.9. Given a set of static program over-approximations, then applying the third residual construction, consecutively, returns one which is equivalent to the original with respect to the intersection of projection into ground traces of each approximation: $\mathit{residual}_2(D, Ps) \cong_{\bigcap_{0<i<\mathit{length}(Ps)} Ps(i) \Downarrow \mathit{ObjId}_i} D$[12]

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, the static transition of a DATE ranges

[10] See [7] for a full formalisation of what this means, and a proof for that statement, which we have omitted from here, given lack of space.

[11] Note that we can also define this for singular objects, $\mathit{residual}^P_2(D, P^S_\Sigma, \mathit{objId}) \overset{\text{def}}{=} \mathscr{R}(D \upharpoonright \{d \in \delta \mid \exists t : P^S_\Sigma \Downarrow \mathit{objId} \cdot \mathit{uses}(t,d)\})$ and perform the same *noEffect* analysis as with the previous residual.

[12] This is significant because the ground traces generated by static programs are always a subset of that generated by the runtime program, ensuring that the intersection of the statically generated traces here is a subset of those at runtime.

over a set of states, considering both possibilities of a transition being taken or 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

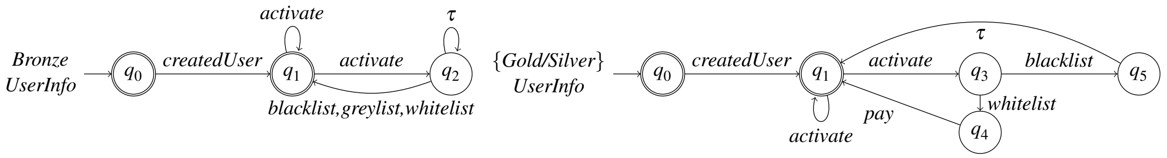

Figure 4: CFG lifted with respect to aliasing of each sub-type of UserInfo, with respect to the property in Figure 5.

These analysis techniques presented in this paper have been implemented in a tool[13]. It is worth noting that due to the differences in the analysis techniques, our tool does not build directly on Clara, although it uses the Soot [19] tool for Java program analysis. The tool was evaluated them on 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 [8]. 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 static analysis was performed three times: (1) with the original property to produce the program monitored by the first residual; (2) with the first residual to produce the program monitored by the second residual property, with some method calls silenced according to the instrumentation point analysis previously defined; and (3) with the second residual to produce the program monitored by the third residual. This is a one-time pre-deployment cost, and it was found to take just a few seconds.

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[14]. This part of the specification is shown in Figure 5, while Figure 4 illustrates the property-relevant behaviour of each possible user object, representing the system under analysis.

The memory used and execution time of the unmonitored program was compared to that of the monitored one, and to that of optimised monitors with the three analyses applied cumulatively. The experiment was run for different numbers of users, with three sample executions used to normalise differences between measurements. Given that the monitors used in the experiment do not have state to keep track

[13] The tool can be downloaded from https://github.com/shaunazzopardi/clarva.

[14] 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.

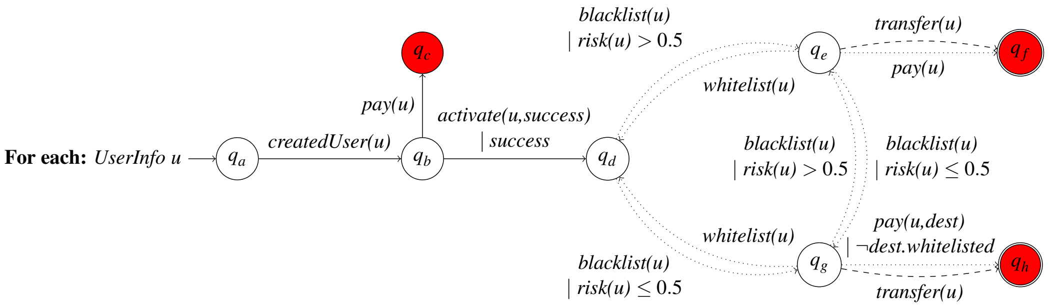

Figure 5: Property, with dashed transitions removed by the first analysis, and dotted by the third.

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

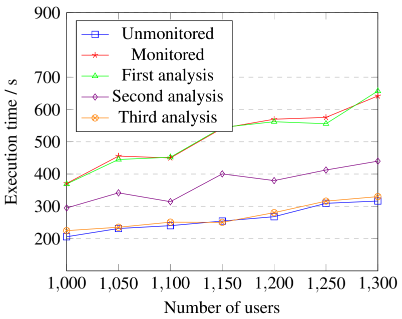

Figure 6: Table and plot of the experiment results.

of, no measurable memory overheads were found any of the experiment runs. However, as expected, monitoring induced considerable processing overheads (see Figure 6), which were substantially reduced using our optimisations from an average of 97% to just 4%.

The first analysis took roughly the same time as the monitored time, since it only removed transitions that could never be taken, resulting in the monitoring engine only bypassing a conditional check for the never-activated event (since *transfer* does not appear in Figure 4). The second analysis did not reduce the property itself, but turned off monitoring of all statements in the program associated only with bronze users, avoiding monitors being created at runtime which are never violated (consider that in Figure 4 bronze users cannot effect payments). The third analysis identified that blacklisted users are never allowed to affect a payment by the application (i.e. the program never makes it to q_f and q_h in Figure 5, consider the parallel composition of the CFGs and the property) 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_b and checked every incoming event against the only remaining outgoing transitions to q_c and q_d). This resulted in an insignificant level of overheads, given the monitoring engine only had to check against two transitions (while in state q_c), without any expensive conditions to check and no tight-looping.

Similar to the results from [11], the gains arise 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 of the system. In our client-provider scenario: (i) the client does not make use of all the functions the provider allows (at least not for every possible object); and (ii) the client is coded in such a way that allows reasoning about its control-flow e.g. blacklisting a user directly by setting a flag. 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 for API clients or clients accessing the server.

5 Related Work

Our work builds directly on the results of Bodden et al. [11], but there are many other instances of the use of static analysis in order to optimise dynamic analysis. In [6], we previously presented a high-level theory of residuals, and a model-based approach to combining static and dynamic analysis, and gave informal examples of how residuals of DATEs could be computed. In this paper we present formally this intuition.

Dwyer et al. [13] 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 affect violation. Jin et al. [14] 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. Other approaches try to make runtime overheads more predictable and manageable, but lose certainty of the verdict. [18] is an example of *event sampling*, where not all events generated by a program are processed by the program, where this approach uses a statistical model to approximate the gaps in the execution trace. [9] extends this approach to reduce the memory and time overheads incurred by statistical calculations performed at runtime, by estimating them statically.

6 Conclusions and Future Work

Through this work, we have extended a static optimisation for the monitoring of parametric properties, to deal with automata with symbolic state. We reduce both the property using the system, and the system instrumentation by using the property. This static analysis is based on the control-flow of the program, and an aliasing relationship between relevant objects of the program, which we illustrated through a 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, this tool can be used to pre-process DATEs before they are used to monitor a program. We are in the process of combining these results with those of StaRVOOrS [3], where in contrast to our approach, StaRVOOrS, static analysis is used to reduce the data-flow aspect of the specification (using 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

[1] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace & Gerardo Schneider (2015): *A Specification Language for Static and Runtime Verification of Data and Control Properties*. In: *FM'15*, 9109, doi:10.1007/978-3-319-19249-9_8.

[2] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace & Gerardo Schneider (2017): *Verifying data- and control-oriented properties combining static and runtime verification: theory and tools*. Formal Methods in System Design, pp. 1–66, doi:10.1007/s10703-017-0274-y.

[3] Wolfgang Ahrendt, Gordon Pace & Gerardo Schneider (2012): *A Unified Approach for Static and Runtime Verification: Framework and Applications.* In: *ISOLA'12*, LNCS 7609, doi:10.1007/978-3-642-34026-0_24.

[4] Rajeev Alur & David L. Dill (1994): *A Theory of Timed Automata. Theor. Comput. Sci.* 126(2), pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[5] Rajeev Alur & Mihalis Yannakakis (2001): *Model Checking of Hierarchical State Machines. ACM Trans. Program. Lang. Syst.* 23(3), pp. 273–303, doi:10.1145/503502.503503.

[6] Shaun Azzopardi, Christian Colombo & Gordon Pace (2016): *A Model-Based Approach to Combining Static and Dynamic Verification Techniques*, pp. 416–430. Springer International Publishing, Cham, doi:10.1007/978-3-319-47166-2_29.

[7] Shaun Azzopardi, Christian Colombo & Gordon Pace (2017): *Control-flow analysis for Symbolic Automata.* Technical Report CS-2017-01, Department of Computer Science, University of Malta. Available at http://www.um.edu.mt/ict/cs/research/technical_reports.

[8] Shaun Azzopardi, Christian Colombo, Gordon J. Pace & Brian Vella (2016): *Compliance Checking in the Open Payments Ecosystem*, pp. 337–343. Springer International Publishing, Cham, doi:10.1007/978-3-319-41591-8_23.

[9] Ezio Bartocci, Radu Grosu, Atul Karmarkar, Scott A. Smolka, Scott D. Stoller, Erez Zadok & Justin Seyster (2013): *Adaptive Runtime Verification*, pp. 168–182. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-35632-2_18.

[10] Eric Bodden, Patrick Lam & Laurie Hendren (2008): *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, BCS Learning & Development Ltd., Swindon, UK, pp. 391–405. Available at http://dl.acm.org/citation.cfm?id=2227536.2227569.

[11] Eric Bodden, Patrick Lam & Laurie Hendren (2012): *Partially Evaluating Finite-State Runtime Monitors Ahead of Time. ACM Trans. Program. Lang. Syst.* 34(2), pp. 7:1–7:52, doi:10.1145/2220365.2220366.

[12] Christian Colombo, Gordon J. Pace & Gerardo Schneider (2009): *Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties*, pp. 135–149. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-03240-0_13.

[13] Matthew B. Dwyer & Rahul Purandare (2007): *Residual Dynamic Typestate Analysis Exploiting Static Analysis: Results to Reformulate and Reduce the Cost of Dynamic Analysis.* In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, ACM, New York, NY, USA, pp. 124–133, doi:10.1145/1321631.1321651.

[14] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith & Grigore Rosu (2011): *Garbage Collection for Monitoring Parametric Properties. SIGPLAN Not.* 46(6), pp. 415–424, doi:10.1145/1993316.1993547.

[15] Akash Lal, Nicholas Kidd, Thomas Reps & Tayssir Touili (2007): *Abstract Error Projection*, pp. 200–217. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-540-74061-2_13.

[16] Martin Leucker & Christian Schallhart (2009): *A brief account of runtime verification. The Journal of Logic and Algebraic Programming* 78(5), pp. 293 – 303, doi:10.1016/j.jlap.2008.08.004.

[17] Rahul Purandare, Matthew B. Dwyer & Sebastian Elbaum (2012): *Monitoring Finite State Properties: Algorithmic Approaches and Their Relative Strengths*, pp. 381–395. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-29860-8_31.

[18] Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka & Erez Zadok (2012): *Runtime Verification with State Estimation*, pp. 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-29860-8_15.

[19] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam & Vijay Sundaresan (2010): *Soot: A Java Bytecode Optimization Framework.* In: *CASCON First Decade High Impact Papers*, CASCON '10, IBM Corp., Riverton, NJ, USA, pp. 214–224, doi:10.1145/1925805.1925818.