Residual Runtime Verification via Reachability Analysis

Chukri Soueidi[0000−0002−6112−9946] and Yliès Falcone[0000−0002−0114−0641]
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
{chukri.a.soueidi,ylies.falcone}@inria.fr

Abstract. We leverage static verification to reduce monitoring overhead when runtime verifying a property. We present a sound and efficient analysis to statically find safe execution paths in the control flow at the intra-procedural level of programs. Such paths are guaranteed to preserve the monitored property and thus can be ignored at runtime. Our analysis guides an instrumentation tool to select program points that should be observed at runtime. The monitor is left to perform residual runtime verification for parts of the program that the analysis could not statically prove safe. Our approach does not depend on dataflow analysis, thus separating the task of residual analysis from static analysis; allowing for seamless integration with many RV frameworks and development pipelines. We implement our approach within BISM, which is a recent tool for bytecode-level instrumentation of Java programs. Our experiments on the DaCapo benchmark show a reduction in instrumentation points by a factor of 2.5 on average (reaching 9), and accordingly, a reduction in the number of runtime events by a factor of 1.8 on average (reaching 6).

Keywords: Residual Runtime Verification, Instrumentation, Parametric Monitoring, Control Flow, Runtime Overhead, Bad prefix.

1 Introduction

Runtime verification (RV) [6,19,27,17,18] is a formal method that allows checking whether a run of a system respects a specification. The specification usually formalizes a correctness property and is written in a suitable formalism based for instance on temporal logic or finite-state machines. There are still many challenges in runtime verification, see [28] for a survey. One of the most prominent and fundamental ones, hindering its wide applicability in application domains, is the runtime overhead introduced when augmenting a system with runtime verification. Overhead is caused by the instrumentation code inserted in the program to generate traces. In the case of online monitoring, overhead also comprises the evaluation of traces by the monitors.

RV can complement and has been used in combination with other formal static verification methods such as model checking [26], deductive verification [12] and static analysis [15,9,12,4], as well as informal dynamic methods such as testing [13] and debugging [23]. While a complete *a priori* verification is ideally desirable for verifying program correctness, proving the correctness of many properties is fundamentally undecidable statically. However, static verification often relies on conservative approximations that produce sound results while sacrificing completeness. Combining static and runtime verification for a more complete verification scheme seems natural. The combination is useful in the two complementary directions. For static verification, it improves

completeness by deferring verification of undecidable fragments for some properties until runtime. For RV, it reduces the overhead of monitoring by pruning parts of the program that can be statically analyzed. In this paper, we pursue the second direction.

We follow the terminology of [15] and refer to the introduced technique as *residual runtime verification*. Our work is directed to handle properties that can be expressed by finite-state automata, such as typestate [31] errors, supporting different formalisms and monitoring approaches that allow specifications with data. In such approaches, a parametric monitor receives a parametrized trace and spawns multiple monitors for different trace slices corresponding to sets of related objects [20]. We see our contributions as follows. We present a *sound and efficient* technique to statically find "safe" execution paths in the control flow at the intra-procedural level of programs. Such paths are guaranteed to preserve the monitored property and thus can be ignored at runtime. As a result, the monitor is left to perform verification for residual parts of the program that the analysis failed to prove safe statically. Our approach, at its core, does not depend on data-flow analysis nor on a static construction of the full call graph of the program, which might be difficult and expensive to produce in practice. Thus, we separate the problem of static analysis from the residual analysis, allowing for seamless integration with the RV workflow. Instead, we analyze the control-flow graphs of single methods and rely on over-approximations of the behavior of the program. We assume that the variables generating events within one method *may-alias* and our analysis reasons about all possible projections of traces. We also handle instructions that may allow references to escape from methods by including them in the analysis to guarantee soundness. We demonstrate the effectiveness of our analysis when we reduce the number of instrumentation points by a factor of 2.5 on average (reaching 9), and accordingly, the number of generated events at runtime by a factor of 1.8 on average (reaching 6). Our approach is fully implemented as an extension of the instrumentation tool BISM [30], and [29] where we presented the *control-flow graph automaton*, a model to abstract the program behavior at the intra-procedural level.

The rest of this paper is structured as follows. Section 2 motivates our approach with a running example. Section 3 reviews background notions. Section 4 defines the requirements for residual analysis. Section 5 describes our instantiation of residual analysis at the interprocedural level. Section 6 briefly overviews our implementation. Section 7 reports on our experiments. Section 8 reviews the related research focusing on residual analysis. Section 9 concludes and presents perspectives.

2 Motivating Example and Approach Overview

We first introduce our running example. We are interested in monitoring the **SafeIterator** property which specifies that "*A collection should not be updated when an iterator associated with it is created and being used*". Figure 1, shows a contrived Java method m along with its control-flow graph (CFG). It retrieves 2 lists (lines 3,4), updates them (lines 6,10,11), creates iterators (lines 7,15), and calls the "next" method on the iterator (line 14). We are interested in answering the following questions:

- **Q1**: Can we fully verify this program statically? If yes, then there is no need to instrument and runtime monitor it.

```
1  void m()
2  {
3     List l1 = ... ;
4     List l2 = ... ;

6     l1.add(..); // event u
7     Iterator it = l1.iterator();// event c

9     if(someflag) {
10       l2.add(..); // event u
11       l2.add(..); // event u
12    }

14    Object o = it.next(); // event n
15    it = l1.iterator(); // event c
16    l2.add(..); // event u
17 }
```

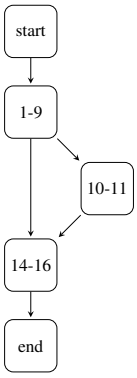


Fig. 1: A method using Iterators in Java, and its CFG.

– **Q2**: If not, can we statically verify some parts of it? If yes, how can we find them so that we only monitor the *residual* parts?

By manually inspecting the program and its control-flow graph we see that, at runtime, it may violate the property if the execution enters the `if` block, labeled (10-11) in the graph. More precisely, a violation can occur if both of the following conditions are met: **(1)** `someflag` evaluates to *true*; and **(2)** if the variables `l1` and `l2` alias each other i.e., they refer to the same object in memory. Let us consider that Condition **(1)** is only decidable at runtime. To generally decide Condition **(2)** statically, we need to perform pointer analysis on the program that checks all calling contexts `m` and return whether `l1` and `l2` alias. In practice, we may get one of the following results about our query: the two objects *must-alias*, *may-alias*, or *must-not-alias*. Moreover, pointer analysis often times out and never returns a result. However, to answer **Q1**, we need to get the result that `l1` and `l2` *must-not-alias*, i.e., they refer to different objects in memory. This is a sufficient condition to statically ensure that m will behave correctly at runtime regardless of the control flow since the update actions on Lines 10-11 are not on the list iterated by iterator `it`. To answer **Q2**, by observing Lines 15-16, we can see that, regardless of what happens at execution, these two instructions are safe and their execution does not need to be monitored. Also, in Line 6, the instruction is safe since it updates the list before the creation of the iterator.

Pointer analysis may not always conclude with a result, especially for Java programs. In addition to the inability to construct the full static call graph of the program, Java allows for dynamic class loading and reflection which often cause additional problems to pointer analysis. Our work relies on the idea that when statically analyzing cases such as the one of Condition (2), one can safely assume that such two variables *may-alias*, even without performing pointer-analysis. Also, we analyze methods separately and thus need to handle escaping references. Objects in the program that are relevant to the property may escape from the method to a subroutine or a return statement and

produce events there. As such, we handle all instructions that may allow references to escape, such as method calls, with special *escape events*.

Our over-approximation might miss some positive answers to **Q1**, therefore missing some optimization opportunities. However, based on our observation (such as the experiments in Sec. 7), cases where one needs to perform pointer analysis such as in Condition (2) are less frequent in many Java programs. As such, our approach mainly addresses **Q2** while it is also capable of answering **Q1** but, in certain cases, less effectively.

3 Background

We recall concepts related to monitoring in general and our verification approach in particular. We assume basic familiarity with automata theory such as the definitions of a finite-state machine, words, runs, and acceptance, and refer to [22] for more details.

3.1 Monitoring

Let Σ be a set of events, Σ^* and Σ^ω are the sets of all finite and infinite traces over Σ, respectively. A finite trace is a sequence of events, a word in Σ, that can be modeled by a function $t : [1, n] \to \Sigma$ for a trace of length n. We say that an event belongs to the trace, noted $e \in t$, when $e \in \mathrm{codom}(t)$. A property φ is a *language* over Σ which is a subset of Σ^*. Given a trace t in Σ^*, the set of prefixes of t, noted $pre(t)$, is defined as: $pre(t) = \{p \in \Sigma^* \mid \exists s \in \Sigma^* : t = ps\}$. The set of matching prefixes is the set of prefixes of a trace within a given language L.

Definition 1 (Matching prefixes [10]). *Given a language $L \subseteq \Sigma^*$ and a trace $t \in \Sigma^*$, the matching prefixes of t in L is given by: $match_L(t) = pre(t) \cap L$.*

Many monitoring techniques and approaches essentially rely on the detection of *bad and good prefixes*, which are intuitively the witnessing sequences allowing a monitor to conclude about monitoring the program based on the trace observed so far.

Definition 2 (Bad/Good prefixes [25]). *Given a language $L \subseteq \Sigma^*$ of finite traces over Σ (or of infinite traces, $L \subseteq \Sigma^\omega$). A finite trace $u \in \Sigma^*$ is a bad prefix for L, if $\forall w \in \Sigma^* : uw \notin L$ (or $\forall w \in \Sigma^\omega : uw \notin L$, if L is over infinite traces). Moreover, u is a good prefix for L, if $\forall w \in \Sigma^*$: $uw \in L$ (or $\forall w \in \Sigma^\omega : uw \in L$, if L is over infinite traces).*

The languages of bad and good prefixes are extension-closed; since every continuation of a bad or a good prefix for a language, L, by a finite word is also a bad (good) prefix for L. When monitoring at runtime, we are interested in reporting a violation/satisfaction of a property from a trace as early as possible. Matching a *bad* (alternatively *good*) prefix is sufficient to produce a final verdict since every continuation of the trace will produce the same result. For instance, the techniques in [7,16] synthesize a monitor (as a finite-state automaton) that recognizes the good and bad prefixes of the language denoted by a temporal-logic formula or by an automaton over infinite traces.

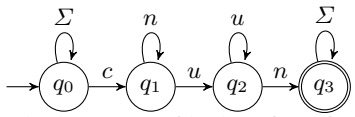


Fig. 2: Monitor recognizing the language of bad prefixes for the *SafeIterator* property.

Example 1 (SafeIterator monitor). Figure 2 shows the monitor that checks for the violation of the property from Sec. 2. Event c denotes a creation of an iterator associated with a list by calling `list.iterator()`, event u denotes an update on a list by calling `list.add(..)`, and event n denotes calling the next method `iterator.next()` on an iterator. The monitor recognizes the bad prefixes in the traces received from a running program. Note that the monitor reaches the accepting state when seeing the pattern $c.n^*.u^+.n$, as it suffices to conclude that the whole run violates the property.

Definition 3 (Property satisfaction). *We say that a trace $t \in \Sigma^*$, satisfies a property $\varphi \subseteq \Sigma^*$ denoted by $t \models \varphi$ iff $t \in \varphi$. Alternatively, for a safety property φ with its language of bad-prefixes L, $t \models \varphi$ iff $match_L(t) = \emptyset$. And, for a co-safety property φ' with its language of good-prefixes L', $t \models \varphi$ iff $match_{L'}(t) \neq \emptyset$.*

3.2 Parametric Monitoring

Monitoring is in practice performed on parametric monitors that receive events accompanied by runtime information about the objects producing them, allowing to monitor each related set of objects in the program separately. There is a myriad of different approaches to parametric monitoring that differ in the manner they interpret events with runtime information and project these to instances of monitors. See [11,5] for example approaches and [21,18] for overviews. Here, we sketch a simple and general approach to parametric monitoring that can be adapted to several existing approaches.

We denote the set of variables defined by a parametric monitor by X, and the set of values that these variables can take by V. These values usually correspond to objects in the memory of the execution environment of the program. A variable binding $\theta : X \rightharpoonup V$ maps monitor parameters to their values and $\mathcal{B}$ is the set of all possible bindings in a program. A parametric event $e\langle\theta\rangle$ is then a pair $(e, \theta) \in \Sigma \times \mathcal{B}$. We denote the set of all parametric events as $\Sigma\langle X\rangle$ and a parametric trace as a word in $\Sigma\langle X\rangle^*$.

Example 2 (Parametric traces). One trace that the program from Figure 1 may generate at runtime is the following: τ = $(u, [l \mapsto \mathrm{o}(l1)])$ $(c, [l \mapsto \mathrm{o}(l1), i \mapsto \mathrm{o}(it)])$ $(u, [l \mapsto \mathrm{o}(l2)])$ $(u, [l \mapsto \mathrm{o}(l2)])$ $(n, [l \mapsto \mathrm{o}(l1), i \mapsto \mathrm{o}(it)])$ $(c, [l \mapsto \mathrm{o}(l1), i \mapsto \mathrm{o}(it)])$ $(u, [l \mapsto \mathrm{o}(l2)])$. The event $(c, [l \mapsto \mathrm{o}(l1), i \mapsto \mathrm{o}(it)])$ denotes the creation of an iterator, event c, where the variable l, representing the associated list, is bound to the runtime object of $l1$ denoted by $\mathrm{o}(l1)$ and the variable i, representing the created iterator, is bound to the runtime object of it denoted by $\mathrm{o}(it)$.

A parametric property $\Lambda X.\varphi$ (notation borrowed from [11]) is then defined over traces of parametric events such that $\Lambda X.\varphi \subseteq \Sigma\langle X\rangle^*$. To monitor each group of related objects separately, a parametric monitor slices a parametric trace according to the values bound to the monitor parameters carried within events. Slicing is achieved by projecting

a trace τ on all seen bindings using a projection function denoted by $\tau{\downarrow_\theta}$. We omit the formal details here for brevity. The projection results in a set of traces, we refer to as *projected traces*, where each trace contains non-parametric events that correspond to related objects in the program and is sent to a monitor that was spawned specifically for that slice.

Definition 4 (Projected traces).
Given a parametric trace τ in $\Sigma\langle X\rangle^*$, the set of projected traces is denoted by $Proj(\tau) \subseteq \Sigma^*$, and is defined as:

$$Proj(\tau) = \bigcup_{\theta \in \mathcal{B}} \tau{\downarrow_\theta}$$

Example 3 (Projected traces). Consider τ from Ex. 2. A parametric monitor will check at runtime the relation between $\mathrm{o}(l1)$ and $\mathrm{o}(l2)$[1] then accordingly produce $Proj(\tau) = \{ucuuncu\}$ if $\mathrm{o}(l1) = \mathrm{o}(l2)$ or $Proj(\tau) = \{ucnc, uuu\}$ if $\mathrm{o}(l1) \neq \mathrm{o}(l2)$.

Definition 5 (Parametric property satisfaction). *A parametric trace $\tau \in \Sigma\langle X\rangle^*$ satisfies a parametric property $\Lambda X.\varphi$ denoted by:*

$$\tau \models \Lambda X.\varphi \stackrel{\text{def}}{=} \forall t \in Proj(\tau) : t \models \varphi$$

3.3 Upward Closure

We recall the notions for subwords and their closures for regular languages. We refer to [24] for full details and borrow their definitions. For a word $x \in \Sigma^*$, the length of x is denoted by $|x|$, and for $1 \leq i \leq |x|$, let x_i denote the i-th letter of x. We denote the empty word by ϵ. A *subword* is obtained by removing certain letters from a word at arbitrary positions, and, a *superword* is obtained by inserting any number of letters into a word at arbitrary positions. We say that a word x is a subword of y, denoted by $x \sqsubseteq y$, equivalently y is a superword of x when there are positions $0 < p_1 < p_2 < \ldots p_l \leq |y|$ such that $x[i] = y[p_i]$ for all $1 \leq i \leq l = |x|$. For $\Sigma = \{a, b, c\}$, we have $\varepsilon \sqsubseteq ab \sqsubseteq acba$.

Definition 6 (Upward closure of a language). *For a language $L \subseteq \Sigma^*$, the upward closure of L, is denoted by $\uparrow L$ and defined as $\{x \in \Sigma^* \mid \exists y \in L : y \sqsubseteq x\}$.*

For any language $L \subseteq \Sigma^*$, we have $L \subseteq \uparrow L$. Moreover, a language L is *upward-closed* if $L = \uparrow L$. For a regular language L recognized by a non-deterministic finite-state automaton (NFA), we can obtain an NFA recognizing $\uparrow L$ by simply adding transitions without increasing the number of states. More precisely, given an automaton $A = (\Sigma, Q, \delta, Q_0, F)$ recognizing L, the NFA $A^\uparrow = (\Sigma, Q, \delta', Q_0, F)$ recognizing $\uparrow L$ is obtained by adding a self loop on every state $q \in Q$ and every letter $s \in \Sigma$ such that $\delta' = \delta \cup \{\langle q, s, q\rangle \mid q \in Q, s \in \Sigma\}$.

[1] This can checked with == in Java.

3.4 Programs, CFG and Instrumentation

Given a program P, let $Methods$ be the set of all its methods, $Instructions$ the set of all byte-code instructions, and $Instructions_m$ all instructions of a method m. The $CFG_m = \langle B_m, E_m \rangle$ of m is a directed graph, where B_m is the set of nodes such that each instruction is a node, and $E_m \subseteq B_m \times B_m$ are edges that connect nodes to their successors. The instruction in a node b is denoted by $b.instr$, and $b.entry$ (resp. $b.exit$) is a Boolean which holds if b is the entry node (resp. is an exit node) for the method. To monitor a program, we abstract its execution in a trace of events extracted at runtime. This is achieved by instrumentation, which can be modeled by the function $\mathrm{instrument} : Instructions^* \rightharpoonup \Sigma\langle X \rangle^*$.

4 Residual Analysis of Parametric Properties

We aim to verify a program P against some parametric property $\Lambda X.\varphi$. The behavior of a program is abstracted by the set of parametric event traces that it can produce at runtime. Let $[P] \subseteq \Sigma\langle X \rangle^*$ be such set for a program P. The verification problem can then be stated as checking if all the traces of the program satisfy the property:

$$P \models \Lambda X.\varphi \stackrel{\text{def}}{=} \forall \tau \in [P] : \tau \models \Lambda X.\varphi$$

Any verification technique aiming to verify the program should then be able to explore all the parametric traces that the program can generate. Moreover, the technique should also be able to explore for each parametric trace τ the set of projected traces $Proj(\tau)$ (see Sec. 3.2). However, statically, exploring the parametric traces requires full knowledge of the call graph of the program, whereas exploring projected traces requires knowledge of the aliasing relations between objects producing them. We know that obtaining such information is generally undecidable statically. Meanwhile, at runtime, this information is completely available. Yet, runtime verification incurs overhead on the execution of the program where this overhead is typically positively correlated with the size of traces. Our interest is then to statically verify parts of the program and leave a residual part for runtime verification.

Our proposed residual analysis *statically* identifies a set of instructions in the program, $\mathcal{S}_P$, that can be safely silenced/ignored at runtime from the monitor side without affecting verification. Ignoring an instruction means that there is no need to produce an event when it executes. As such, we aim to construct the *residual instrumentation* function $\mathrm{residual} : Instructions^* \to (\mathcal{S}_P \to \Sigma\langle X \rangle^*)$. Let us note $Runs \subseteq Instructions^*$ the set of all the possible runs of a program P. Instrumenting the program with $\mathrm{residual}$ should ideally produce shorter traces than $\mathrm{instrument}$, however, for both, we should get the same monitoring verdict. We can state the condition that should be met by the residual analysis as follows:

$$\begin{aligned} \forall r \in Runs : &|\mathrm{residual}(r)| \leq |\mathrm{instrument}(r)| \\ &\wedge\ \mathrm{residual}(r) \models \Lambda X.\varphi \iff \mathrm{instrument}(r) \models \Lambda X.\varphi \end{aligned}$$

To perform the residual analysis statically and produce the set $\mathcal{S}_P$, we can over-approximate the program behavior by constructing a set $[\widehat{P}] \supseteq [P]$. This allows us to

explore all the parametric traces that the program can produce but also traces that the program might never produce. A residual analysis should then check whether silencing some instructions does not affect the verification verdict of any trace in $[\widehat{P}]$, and safely assumes the same effect in $[P]$. Yet, given that $[\widehat{P}]$ is an over-approximation, the analysis may suffer from false positives, which are instructions that can indeed be silenced however the analysis found the opposite. In what follows, we consider a subset $[\widehat{P_m}] \subseteq [\widehat{P}]$ for our residual analysis, these are traces that are fully produced in single methods.

5 Residual Analysis via Intraprocedural Reachability Analysis

We demonstrate our instantiation of the residual analysis at the intraprocedural level using reachability analysis. Recall that we avoid dataflow and pointer analysis, as such, we do not have a static call graph for the program nor the variable aliasing relation. In Sec. 5.1, we capture the behavior of a method by using its control-flow graph to construct a representative model that allows us to explore the parametric traces a method can generate. In Sec. 5.2, we deal with the over-approximations by extending the bad-prefix automaton to handle different projections that might be produced by a parametric trace. In Sec. 5.3, we then present the reachability analysis algorithm that finds *safe* and *violating* paths in the control-flow graph; by cutting the behavior in a model-based checking approach. Finally, in Sec. 5.4, we discuss the soundness of our analysis.

5.1 Capturing the Behavior

Our analysis treats methods separately, however, we need to be careful. If a method receives as an argument an object which is a type that is capable of producing events in the alphabet of the property, then we cannot assume any previous behavior. As such, we exclude such methods from the analysis. For the same reason, we exclude all methods that operate on static instances of the types involved.

For each method m, we map two types of instructions to events and discard all other instructions as they are irrelevant to our analysis. We keep instructions that produce events in Σ, given by the property specification. We also keep instructions that may allow any object reference to escape from the context of method m; we introduce the new *escape* event ($\#$) for such instructions. Escape events are assignments to class fields, method calls that pass objects by references, in addition, to return statements that return objects [14]. However, our analysis allows the user to specify a safe list of instructions, denoted by the set $SafeList$, defined over the compile type information such as method names, package and type names, and opcodes. For instance, calling `System.out.print(l1.toString())` is a safe instruction. All instructions that are escape events and are not in $SafeList$ are added to the set Esc_m.

Given the alphabet of a property Σ and the control-flow graph of a method m, $CFG_m = \langle B_m, E_m \rangle$, we replace each block b in B_m with b' and map its instruction to an event, and construct the CFG Automaton as follows.

$$b'.instr \;=\; b.instr.\text{map}\left(i \mapsto \begin{cases} i \text{ if } i \in \Sigma, \\ \# \text{ else if } i \in Esc_m \\ \epsilon \text{ otherwise} \end{cases}\right)$$

Definition 7 (CFG Automaton). *Given the mapped CFG_m, the CFG Automaton is the non-deterministic finite-state automaton $\mathcal{A}^c_m = (\Sigma \cup \{\#\}, Q, \delta, q_0, F)$ constructed as follows:*

$$Q = \{q_b \mid b \in B_m\} \qquad q_0 = \{q_b \mid b \in B_m \ \wedge \ b.entry = true\ \}$$
$$F = Q \qquad \delta = \{\langle q_b, s, q_{b'}\rangle \mid \langle b, b'\rangle \in E_m \ \wedge \ b.\text{instr} = s\}$$

Each node in the control-flow graph is now represented as a state in the CFG Automaton. We make all states accepting states and merge states connected with ϵ transitions. Now, by traversing the CFG Automaton, we can explore the paths that method m can take at runtime and thus the parametric traces it can produce.

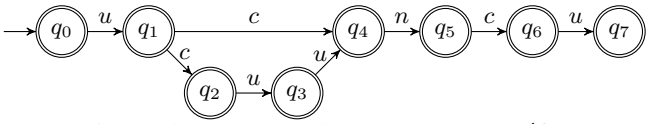


Fig. 3: The constructed CFG Automaton $\mathcal{A}^c_m$

Example 4 (CFG Automaton). Figure 3, shows the CFG Automaton constructed from the method in Sec. 2. Each state corresponds to an instruction that we are interested in the program. Two traces can be explored from the automaton in Figure 3, $t_1 = ucuuncu$, which corresponds to the parametric trace τ from Ex. 2, and $t_2 = ucncu$.

5.2 Extending the Automaton of Bad Prefixes

We now describe how our analysis handles the over-approximations and extends the bad prefix automaton.

Handling Variables May-alias Recall from Sec. 3.2, that a parametric trace τ in $\Sigma\langle X\rangle^*$ at runtime is projected into $Proj(\tau)$ to possibly multiple traces in Σ^*, depending on the aliasing relationship between the objects carried in the events. At runtime, this aliasing relationship is available for the parametric monitor to do the projection. However, statically for our residual analysis this information is not available. Our central idea in this paper is to avoid performing data-flow analysis and assume that the objects producing events in a method may-alias. For two events, our analysis should then consider the case when the objects bound to them *must-alias* and the case when they *must-not-alias*. In the former case, both events will be projected into the same trace, and in the latter, they will be projected into different traces.

Example 5 (Projected traces approximation with may-alias). Consider the trace t_1 which can be explored with the CFG Automaton from Ex. 4. At runtime, if the program takes such a control flow path, it emits a parametric trace that produces either of the projected traces from Ex. 3 depending on whether $l1$ and $l2$ alias. Since we avoid producing the aliasing relation statically and assume that $l1$ and $l2$ may alias, we should then consider in our residual analysis the disjunction of both cases. Thus the traces

$pt_1 = \{ucuuncu, ucnc, uuu\}$ should be checked by our residual analysis. As for t_2 from Ex. 4, by the same reasoning, the traces to be checked are $pt_2 = \{ucncu, ucnc, u\}$. Hence for method m, the set of traces that should be checked is $pt_1 \cup pt_2$.

The CFG Automaton allows us to explore the different paths that the program can take at runtime, however, its traces are too coarse. They may be *polluted* with events that do not correspond to the same trace at runtime. We notice from above that this is equivalent to generating and considering all the subwords of a trace, where the real trace can be any subword of a trace that can be explored with the automaton. Thus, to safely handle the different projections, we use the upward closure, from Sec. 3.3, of the language of bad prefixes L. By using the upward closure $\uparrow L$, we can recognize a bad prefix in a full trace or any subword of it since $L \subseteq \uparrow L$, allowing us to find bad prefixes in all possible projected traces. However, we restrict the closure by removing the Σ self-loops from the initial and final states as we want to find the shortest paths that match a bad prefix.

Handling Escape Events In Sec. 5.1, when constructing the CFG automaton, we introduced the escape $\#$ events. Since our analysis analyzes each method separately, we are oblivious to what might be happening in $\#$-transitions. We have to assume that they might produce events untracked by the method under analysis. To handle them safely, we add a $\#$-transition in the bad prefixes automaton from each state to all of its reachable states. Intuitively, this means when $\#$ event is encountered in a path, we assume that the path is not safe anymore and that it might match a bad prefix.

Extending the Bad Prefixes Automaton We proceed to show how we extend the automaton of bad prefixes to handle the multiple projected traces and the escape events.

Definition 8 (Extended automaton of bad prefixes). *Given the language of bad prefixes $\mathcal{L}(bad_\varphi)$ recognized by automaton $\mathcal{A}^{bad_\varphi} = (\Sigma, Q, \delta, Q_0, F)$ with its extended transition function $\hat{\delta}$. The extended automaton of bad prefixes is defined as $\mathcal{A}^{\uparrow bad_\varphi} = (\Sigma \cup \{\#\}, Q, \delta', Q_0, F)$ where:*

$$\begin{aligned}
\delta' = \quad & \delta \setminus \{\, \langle q, s, q\rangle \mid s \in \Sigma \wedge (q \in F \vee q \in Q_0) \,\} && (1)\\
\cup \quad & \{\, \langle q, s, q\rangle \mid s \in \Sigma \cup \{\#\} \wedge q \in Q \wedge q \notin Q_0 \wedge q \notin F\} && (2)\\
\cup \quad & \{\, \langle q, \#, q'\rangle \mid q, q' \in Q \wedge \exists w \in \Sigma^* : \hat{\delta}(q, w) = q' \wedge q' \notin F\} && (3)
\end{aligned}$$

The extended automaton has the same states. We remove the self-loops from initial and final states, as we want to find the shortest paths that match a bad prefix (1). We add the upward closure by adding Σ and $\#$ self-loops on all other states (2). We add $\#$-transitions from each state to the reachable states from it (3).

Example 6 ($\mathcal{A}^{\uparrow bad_\varphi}$ for the SafeIterator property). Figure 4, shows a construction of automaton $\mathcal{A}^{\uparrow bad_\varphi}$. Recall the pattern $c.n^*.u^+.n$ from Ex. 1, the new automaton will now recognize such a pattern while also handling the two over-approximations above.

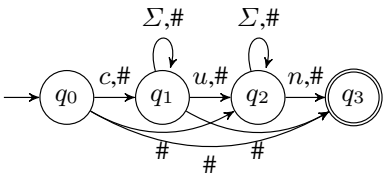


Fig. 4: The constructed automaton $\mathcal{A}^{\uparrow bad_\varphi}$.

5.3 Cutting the Behavior

We now proceed to describe how we find violating paths in the method. The idea is to traverse the constructed CFG automaton $\mathcal{A}^c_m$ state by state and check whether there is a path, starting from the visited state, that makes the extended bad prefixes automaton reach a final state. We limit the discussion here to matching bad prefixes, nevertheless, the same analysis works for matching good prefixes. However, when finding paths that match good prefixes, these will be the safe paths.

Given an automaton, $\mathcal{A}$, $\mathcal{A}(q)$ denotes $\mathcal{A}$ where q is set to be the initial state. Recall that, given a finite state machine $\mathcal{A}(q)$ with its extended transition function $\hat{\delta}$ [22], a state q is coreachable if there exists a word $s \in \Sigma^*$ such that $\hat{\delta}(q, s) \in F$. State q is reachable if there exists a word $s \in \Sigma^*$ such that $\hat{\delta}(q_0, s) = q$ and q_0 is an initial state.

Algorithm 1: Marking violating and safe paths

```
1  Given A^c_m = (Σ ∪ {#}, Q, δ, q0, F), A^{↑bad_φ}
2  V_m = ∅ // represents all states in a violating path
3  S_m = Q // represents all states in a safe path
4  work := q0 // represents a worklist stack
5  visited = ∅
6  while work not empty do
7      q = work.pop()
8      if q ∉ visited then
9          visited = visited ∪ q
10         if q ∉ V_m then
11             Â = A^c_m(q) × A^{↑bad_φ}
12             if L(Â) ≠ ∅ then
13                 V_m = V_m ∪ {{q' | (q', −) ∈ coreachable(Â)} ∩ reachable(A^c_m(q))}
14         foreach q'' in {q'' | ⟨q, s, q''⟩ ∈ δ} do
15             work.push(q'')
16         end
17 end
18 S_m = S_m \ V_m
```

Algorithm 1, shows how to mark all states in $\mathcal{A}^c_m$ as either safe (in $\mathcal{S}_m$) or violating (in $\mathcal{V}_m$). The algorithm implements a depth-first search starting from the initial node of $\mathcal{A}^c_m$. We maintain a *work* stack and *visited* set, in lines (4,5,7,9,15), to hold automaton states to be visited and states that were already visited, respectively. For each state q we visit, we set q as the initial node and find the intersection with the $\mathcal{A}^{\uparrow bad_\varphi}$, line (11).

If the intersection is not empty (line 12), we find the set of all co-reachable states in the intersection automaton. Each state in the intersection automaton $\hat{\mathcal{A}}$ corresponds to a state in $\mathcal{A}^c_m$ and $\mathcal{A}^{\uparrow bad_\varphi}$. For each coreachable state in $\hat{\mathcal{A}}$, we add its corresponding state in $\mathcal{A}^c_m$ to the set $\mathcal{V}_m$, (line 13). We do not revisit states that are already in $\mathcal{V}_m$ (line 10) since paths leading to a final state in $\mathcal{A}^{\uparrow bad_\varphi}$ are already explored by the intersection.

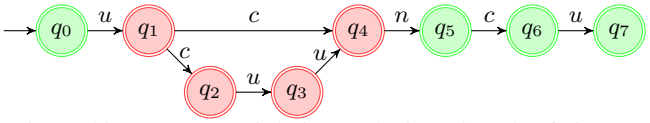


Fig. 5: Marking property violating paths in red, and safe in green.

Example 7 (Property violating states). Figure 5, shows the CFG Automaton constructed from the program from Figure 1, where states marked in red exist in a property-violating path. The red states in the automaton are the states that we need to instrument, and the green states are hidden from instrumentation. We can see that instead of instrumenting at 8 different locations, we only have to instrument at 4 locations.

For our residual analysis, for each method m we analyze, we add the instructions corresponding to the states in $\mathcal{S}_m$ to the set $\mathcal{S}_P$. As for the other states in $\mathcal{V}_m$, their corresponding instructions will be instrumented for runtime monitoring.

5.4 Scope and Soundness of the Analysis

We first argue that our analysis only affects the traces that are fully produced in one method. Recall from Sec. 5.1, that the nodes of CFG automaton $\mathcal{A}^c_m$ correspond to instructions in method m that produce events. We use the notation $\mathrm{ev}(q)$ to denote the corresponding event from an automaton state, and $\mathrm{events}(t)$ to denote all events from a trace t. If some trace t contains events produced by instructions outside of m, then no instruction in m that produced events in t was marked safe.

Proposition 1 (Scope of the analysis). *Given a parametric trace τ in $[P]$:*

$$\begin{aligned}&\forall t \in \mathit{Proj}(\tau), \forall m \in \mathit{Methods}: \\ &\quad (\ \exists i \in \mathit{Instructions} \backslash \mathit{Instructions}_m : \mathrm{instrument}(i) \in t\ ) \\ &\qquad \implies \{\ \mathrm{ev}(q) \mid q \in \mathcal{S}_m\} \cap \mathrm{events}(t) = \emptyset\end{aligned}$$

Proof. Assume that there exists some trace t that has events produced outside of m i.e. $\exists i \in \mathit{Instructions} \backslash \mathit{Instructions}_m : \mathrm{instrument}(i) \in t$ is true. Such traces can be split into two types. Traces that contained events before the execution of m at runtime (1), and traces that start from m but have some events that are produced outside of m at runtime (2). We will show that for both types of traces, the analysis would result in $\mathcal{S}_m = \emptyset$. Since we exclude from the analysis any method that receives a parameter of a type that generates events. Traces from (1), will not be affected by analysis. For that to happen, method m should receive the objects generating the events. Therefore, m will be excluded from the analysis resulting in $\mathcal{S}_m = \emptyset$. As for (2), any escape of an object, which might produce events outside m, is captured by the $\#$ transitions.

From the construction of the bad-prefixes automaton and Algorithm 1, such transitions will result in reaching a final state from any state in the CFG automaton, resulting in $\mathcal{S}_m = \emptyset$. Hence for both types of traces we have $\mathcal{S}_m = \emptyset$, therefore $\{\mathrm{ev}(q) \mid q \in \mathcal{S}_m\} \cap \mathrm{events}(t) = \emptyset$ holds, and the proposition holds. □

Proposition 1 in fact depends on the specification of the $SafeList$ from Sec. 5.1. If some method was added by the user that is not safe, i.e. allows references to escape, then the proposition will not hold. From the above, we also see that our analysis only affects instructions that produce events only in traces that are collected fully in the method itself since otherwise $\mathcal{S}_m = \emptyset$. For soundness, we need to guarantee that at any run of the program, an event that we marked safe in our residual analysis does not have any effect on deciding the violation/satisfaction of the property for any projected trace at runtime. As we showed that the analysis only affects projected traces that are fully produced in one method, we only reason about single methods when discussing soundness.

Theorem 1 (Soundness of the analysis). *Given a language $L \subseteq \Sigma^*$ and $\mathcal{S}_m$ resulting from the analysis on method m, the analysis is sound iff*

$$\begin{aligned}&\forall a_1 \cdots a_i \cdots a_n \in \Sigma^+, \forall i \in \mathbb{N} : \\ &\quad match_L(a_1 \cdots a_i \cdots a_n) \neq match_L(a_1 \cdots a_{i-1} a_{i+1} \cdots a_n) \\ &\qquad \implies a_i \notin \{\, \mathrm{ev}(q) \mid q \in \mathcal{S}_m \}\end{aligned}$$

The condition states that given a projected trace at runtime, if we remove an event a_i from it and get a different match from the new trace i.e. $match_L(a_1 \cdots a_i \cdots a_n) \neq match_L(a_1 \cdots a_{i-1} a_{i+1} \cdots a_n)$, then our analysis must have not statically marked a_i as safe ($a_i \notin \{\, \mathrm{ev}(q) \mid q \in \mathcal{S}_m\}$).

Proof. The proof follows from the definition of Algorithm 1. Assume that when our analysis removes a_i, then $match_L(a_1 \cdots a_i \cdots a_n) \neq match_L(a_1 \cdots a_{i-1} a_{i+1} \cdots a_n)$. This means that a_i is in an extension of $a_1 \cdots a_{i-1}$ that leads to a final state in the monitor of the bad-prefixes of L, or else $match_L(a_1 \cdots a_i \cdots a_n) = match_L(a_1 \cdots a_{i-1} a_{i+1} \cdots a_n)$. However, if a_i is in such a path, then it will be added to $\mathcal{V}_m$ as per Line 7 of Algorithm 1 since the algorithm finds any path from a state in the CFG that reaches the final state of the automaton of bad-prefixes. Then a_i is not in $\mathcal{S}_m$ and $a_i \notin \{\, \mathrm{ev}(q) \mid q \in \mathcal{S}_m\}$ holds. □

6 Implementation

We implement our work as a plugin to the BISM [30] Java byte-code instrumentation tool. Instrumentation directives in BISM are given with *transformers*, which resemble aspects of aspect-oriented programming. BISM provides a mechanism to compose multiple transformers. Transformers, in composition, are capable of controlling the visibility of instructions. For each property, we then apply two transformers to the program: the static analyzer, which performs the residual analysis with a single pass over the code of methods and hides safe instructions, and the second one to instrument the residual part for runtime monitoring. We extend BISM with a module that enables performing

the residual analysis and provides automata operations. The module is used to generate the CFG automata of methods, extend the automaton for bad prefixes, and detect the property-violating execution paths.

7 Evaluation

We report on our evaluation of the effectiveness of our approach[2].

Experimental setup. We compare the instrumentation overhead with our residual analysis, denoted by **RRV**, and without the analysis, denoted by **RV**. We instrument with BISM the programs, in the DaCapo suite [8], for the monitoring of the classical `SafeListIterator` (P1), `SafeMapIterator` (P2), and `SafeHasNext` (P3) properties. (P2) is similar to (P1) from Sec. 2 but is concerned with Java maps. (P3) specifies that a program does not call the `next` method before calling the `hasNext` method of an iterator. We include as *escape* events ($\#$) all assignments to class fields, all method calls that pass objects by references, and in addition, return statements that return objects [14]. We include in the $SafeList$ all calls to methods of Java classes. Other than the method calls relevant to the property and captured by instrumentation, these calls do not produce events. We note that *fop* is the only single-threaded benchmark, however, we can use the multi-threaded benchmarks as we checked that for the properties all the events in the projected traces are being produced within the same thread. We consider 100 runs and then calculate the mean and the standard deviation[3].

Evaluation metrics. We consider the number of affected instructions, methods, and classes by our residual analysis (RRV) and without it (RV). We also consider the improvement factor. We are also interested in evaluating the runtime overhead, that is, the performance degradation caused by instrumentation for monitoring. For runtime, we measure the execution time of the instrumented program. For used memory, we measure the used heap and non-heap memory after a forced garbage collection.

Results. In Table 1, we report the results. The table demonstrates the effectiveness of the residual analysis as it reduces the number of instrumentation points by a factor of 2.5 on average (reaching 9.19), and accordingly, a reduction in the number of generated events at runtime by a factor of 1.8 on average (reaching 6.13). We notice that the reduction of instrumentation points does not always result in a reduction of runtime events for instance with *avrora* with (P1), where we find methods that produce most of the events that we could not prove safe statically. We also notice that most of our missed optimizations are due to escape $\#$ events (see Sec. 5.1). The more diverse operations between events, the more missed optimization. However, the $SafeList$ can be improved with the help of escape analysis to include more instructions that we can guarantee are safe for our analysis and accordingly reduce the number of escape events. We leave that

[2] Implementation details and experiments can be found at https://gitlab.inria.fr/monitoring/residual-runtime-verification-with-bism.

[3] We use Java JDK 8u251 with 16 GB maximum heap size on an Intel Core i9-9980HK (2.4 GHz. 16 GB RAM). We use the DaCapo version 9.12-bach.

Table 1: For each program (Bench), and property (P1), (P2), and (P3), we report # of relevant classes, methods, and instructions (Rel) producing events, number proved safe statically by our technique (Nop), # of events produced at runtime (RV) and after our analysis (RRV), improvement factor for # of instructions instrumented and events produced (Imp). K = 10^3, M = 10^6.

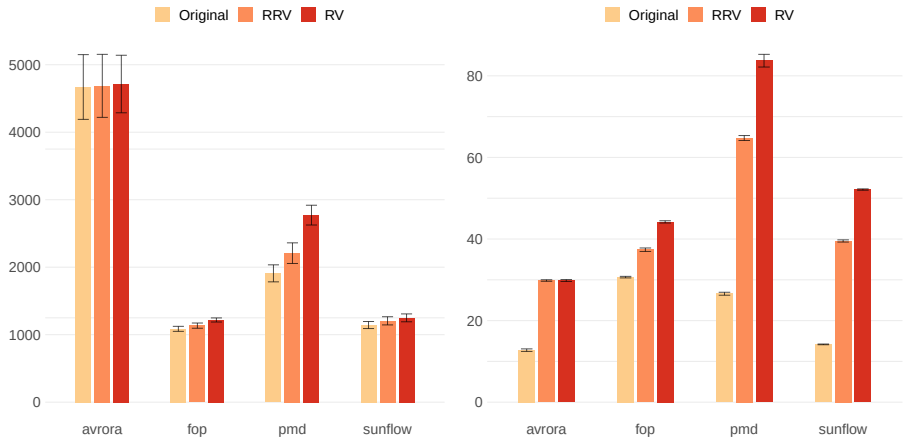
Bench	Property	# Classes		# Methods		# Instructions			# Events		
		Rel	Nop	Rel	Nop	Rel	Nop	Imp	RV	RRV	Imp
avrora	P1	41	14	99	56	165	86	2.09	1.36M	1.36M	1.00
fop	P1	123	33	275	103	700	210	1.43	729K	490K	1.49
sunflow	P1	11	2	35	15	50	15	1.43	2.55M	1.27M	2.00
pmd	P1	86	27	200	95	420	146	1.53	4.77M	778K	6.13
avrora	P2	41	19	111	78	160	117	3.72	353K	246K	1.43
fop	P2	100	28	206	85	2.9K	2.6K	9.19	545K	351K	1.55
sunflow	P2	11	6	32	24	40	26	2.86	2.55M	1.27M	2.00
pmd	P2	81	27	168	70	392	211	2.17	3.01M	2.6M	1.16
avrora	P3	32	11	76	33	160	79	1.98	1.5M	1.29M	1.16
fop	P3	70	7	145	31	376	67	1.22	1.07M	882K	1.21
sunflow	P3	8	2	12	3	29	3	1.12	3.93M	2.65M	1.48
pmd	P3	65	21	126	48	343	115	1.50	5.64M	5.23M	1.08

for future work as we envision adding plugins to incorporate static analysis. We also note that many of the events generated under classical instrumentation (RV) are irrelevant; they occur in methods that do not produce enough events to reach a final state in the monitor. As such, our analysis is effective in removing those from instrumentation. Figures 6 report the execution time and the memory usage for the benchmarks with all three properties combined. The figures show that **RRV** results in better performance in all benchmarks than classical instrumentation **RV**.

8 Related Work

Many research approaches combine static and runtime verification. We focus here on some influential and most recent tools devised for verifying general behavioral parametric properties in sequential programs via residual analysis.

CLARA [10,9] handles properties that can be expressed by finite-state automata by partially evaluating the runtime monitors at compile time and reducing the instrumentation points. It performs three-staged phases of analysis with increasing precision. The more precise phase uses a demand-driven pointer analysis and handles intra-procedural analysis. The first two phases of its analysis can be easily applied within our framework. However, unfortunately, CLARA is no longer maintained and so is its underlying instrumentation tool the abc compiler [3]. In [33], the authors present two optimizations for [10]. One optimization identifies changeless configurations during the backward analysis; the other one uses local object information to refine the forward analysis and backward analysis of the *nop-shadow* analysis. CLARVA [4] extends CLARA [10] to handle properties expressed by DATEs (Dynamic Automata with



(a) Execution time (ms). (b) Used memory (MB).

Fig. 6: Evaluation for the three properties.

Events and Timers [2]) where events are guarded by runtime conditions and timers. Similar to our approach, it transforms Java code into an automaton-based model and allows for the incorporation of control-flow analyses. CLARVA is capable of reducing the instrumentation points as well as reasoning about and pruning the property itself. However, the analysis relies on constructing the callgraph of the full program and on pointer analysis using Soot [32]. Our approach is still capable of producing optimizations with a single pass on the program and without any dependence on static analysis, separating the limitations of static analysis from the residual analysis.

STARVOORS [12] combines deductive theorem proving with control-flow reachability analysis allowing to target control and data-oriented properties. The formalism used for property specification is ppDATE (an extension of DATE) where the automaton states are extended with pre/post-conditions (*Hoare triples*). The property is reduced by pruning the transitions based on solving the triples with Java theorem prover KEY [1]. STARVOORS is capable of handling control and data-oriented properties, however, it focuses on pruning the property and does not reduce the instrumentation points.

In [26,34], the authors present Predictive Semantics for runtime monitoring at the intra-procedural level. In this setup, the program is analyzed, using the control flow graph (CFG) and program dependence graph (PDG), to find predictive words. Predictive words are events that will occur in sequence in a control-flow path. Then, the monitor at runtime will either receive a single event or a predictive word. This approach does not reduce the instrumentation points, hence does not reduce the overhead of instrumentation, however, it emits predictive words which may produce faster verdicts.

9 Conclusion and Perspectives

We introduce an analysis supporting *residual runtime verification* for parametric properties that can be expressed by finite-state automata. Our approach over-approximates

the behavior of the program and analyzes its methods separately relying only on their control-flow graphs to statically identify safe regions. We have demonstrated the effectiveness of our approach in monitoring the bad prefixes of a property, however, our approach can also be used with good prefixes (when monitoring co-safety properties for instance). Our approach is capable of producing overhead optimizations without any dependence on a specific type of static analysis, separating the task of static analysis from the residual analysis and allowing for seamless integration with many RV frameworks. It is fully implemented and integrated within the BISM instrumentation tool, which is the state-of-the-art instrumentation tool for Java programs. We also demonstrated the significant performance benefits at runtime.

Our work lays the foundation for a residual analysis framework that, at its core, does not depend on any specific static analysis technique. Nevertheless, we plan to extend it with plugins that allow the user to easily incorporate static analysis results aiming to reduce over-approximations and increase precision. The user might opt to include static call graph construction and escape analysis for a more precise approximation of parametric traces, also data-flow pointer analysis for better approximations of projected traces. We plan to provide a language that easily integrates the results of such analysis into our residual analysis mainly via refining the safe list of instructions. We also plan to extend our approach to analyze concurrent programs and handle thread-escaping references.

References

1. Ahrendt, W., Beckert, B., Hähnle, R., Rümmer, P., Schmitt, P.: Verifying object-oriented programs with key: A tutorial. vol. 4709, pp. 70–101 (01 2006). https://doi.org/10.1007/978-3-540-74792-5_4
2. Ahrendt, W., Pace, G., Schneider, G.: A unified approach for static and runtime verification: Framework and applications. vol. 7609, pp. 312–326 (10 2012). https://doi.org/10.1007/978-3-642-34026-0_24
3. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Abc: An extensible aspectj compiler. In: Proceedings of the 4th International Conference on Aspect-Oriented Software Development. p. 87–98. AOSD '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1052898.1052906
4. Azzopardi., S., Colombo., C., Pace., G.: Clarva: Model-based residual verification of java programs. In: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - MODELSWARD,. pp. 352–359. INSTICC, SciTePress (2020). https://doi.org/10.5220/0008966603520359
5. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: Towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7436, pp. 68–84. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_9
6. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Lectures on Runtime Verification - Introductory and Advanced Topics, pp. 1–33 (2018). https://doi.org/10.1007/978-3-319-75632-5_1
7. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011). https://doi.org/10.1145/2000799.2000800

8. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: Java benchmarking development and analysis. SIGPLAN Not. **41**(10), 169–190 (oct 2006). https://doi.org/10.1145/1167515.1167488, https://doi.org/10.1145/1167515.1167488
9. Bodden, E., Lam, P., Hendren, L.: Clara: A framework for partially evaluating finite-state runtime monitors ahead of time pp. 183–197 (01 2010). https://doi.org/10.1007/978-3-642-16612-9_15
10. Bodden, E., Lam, P., Hendren, L.J.: Partially evaluating finite-state runtime monitors ahead of time. ACM Trans. Program. Lang. Syst. **34**(2), 7:1–7:52 (2012). https://doi.org/10.1145/2220365.2220366
11. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Kowalewski, S., Philippou, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 246–261. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_23
12. Chimento, J., Ahrendt, W., Pace, G., Schneider, G.: Starvoors: A tool for combined static and runtime verification of java (09 2015). https://doi.org/10.1007/978-3-319-23820-3_21
13. Chimento, J.M., Ahrendt, W., Schneider, G.: Testing meets static and runtime verification. In: Proceedings of the 6th Conference on Formal Methods in Software Engineering. p. 30–39. FormaliSE '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3193992.3194000
14. Choi, J.D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for java. SIGPLAN Not. **34**(10), 1–19 (oct 1999). https://doi.org/10.1145/320385.320386
15. Dwyer, M.B., Purandare, R.: Residual dynamic typestate analysis exploiting static analysis p. 124 (2007)
16. Falcone, Y., Fernandez, J., Mounier, L.: What can you verify and enforce at runtime? Int. J. Softw. Tools Technol. Transf. **14**(3), 349–382 (2012). https://doi.org/10.1007/s10009-011-0196-8
17. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press (2013). https://doi.org/10.3233/978-1-61499-207-3-141
18. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. Int. J. Softw. Tools Technol. Transf. **23**(2), 255–284 (2021). https://doi.org/10.1007/s10009-021-00609-z
19. Havelund, K., Goldberg, A.: Verify your runs. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. Lecture Notes in Computer Science, vol. 4171, pp. 374–383. Springer (2005). https://doi.org/10.1007/978-3-540-69149-5_40
20. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring Events that Carry Data, pp. 61–102. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_3
21. Havelund, K., Reger, G., Thoma, D., Zalinescu, E.: Monitoring events that carry data. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 61–102. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_3
22. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., USA (2006)

23. Jakse, R., Falcone, Y., Méhaut, J., Pouget, K.: Interactive runtime verification - when interactive debugging meets runtime verification. In: 28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017. pp. 182–193. IEEE Computer Society (2017). https://doi.org/10.1109/ISSRE.2017.19
24. Karandikar, P., Niewerth, M., Schnoebelen, P.: On the state complexity of closures and interiors of regular languages with subwords and superwords. Theoretical Computer Science **610**, 91–107 (2016). https://doi.org/10.1016/j.tcs.2015.09.028
25. Kupferman, O., Y. Vardi, M.: Model checking of safety properties. Form. Methods Syst. Des. **19**(3), 291–314 (Oct 2001). https://doi.org/10.1023/A:1011254632723
26. Leucker, M.: Sliding between model checking and runtime verification. In: Qadeer, S., Tasiran, S. (eds.) Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7687, pp. 82–87. Springer (2012). https://doi.org/10.1007/978-3-642-35632-2_10
27. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5), 293–303 (May 2009). https://doi.org/10.1016/j.jlap.2008.08.004
28. Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstic, S., Lourenço, J.M., Nickovic, D., Pace, G.J., Rufino, J., Signoles, J., Traytel, D., Weiss, A.: A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. **54**(3), 279–335 (2019). https://doi.org/10.1007/s10703-019-00337-w
29. Soueidi, C., Falcone, Y.: Capturing program models with bism. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. p. 1857–1861. SAC '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3477314.3507239
30. Soueidi, C., Kassem, A., Falcone, Y.: BISM: Bytecode-Level Instrumentation for Software Monitoring, https://gitlab.inria.fr/monitoring/bism-tool
31. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability **12**(1), 157–171 (Jan 1986). https://doi.org/10.1109/TSE.1986.6312929, https://doi.org/10.1109/TSE.1986.6312929
32. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. p. 13. CASCON '99, IBM Press (1999)
33. Wang, C., Chen, Z., Mao, X.: Optimizing nop-shadows typestate analysis by filtering interferential configurations. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **8174 LNCS**, 269–284 (2013)
34. Zhang, X., Leucker, M., Dong, W.: Runtime verification with predictive semantics. In: Proceedings of the 4th International Conference on NASA Formal Methods. p. 418–432. NFM'12, Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_37, https://doi.org/10.1007/978-3-642-28891-3_37