A Story of Parametric Trace Slicing, Garbage and Static Analysis

Giles Reger
University of Manchester, Manchester, UK
giles.reger@manchester.ac.uk

This paper presents a proposal (story) of how statically detecting unreachable objects (in Java) could be used to improve a particular runtime verification approach (for Java), namely parametric trace slicing. Monitoring algorithms for parametric trace slicing depend on garbage collection to (i) cleanup data-structures storing monitored objects, ensuring they do not become unmanageably large, and (ii) anticipate the violation of (non-safety) properties that cannot be satisfied as a monitored object can no longer appear later in the trace. The proposal is that both usages can be improved by making the unreachability of monitored objects explicit in the parametric property and statically introducing additional instrumentation points generating related events. The ideas presented in this paper are still exploratory and the intention is to integrate the described techniques into the MarQ monitoring tool for quantified event automata.

1 Introduction

This paper explores ideas for improving the performance of runtime verification based on parametric trace slicing by the static identification of unreachable objects. Runtime verification [8] is the task of checking whether a given property holds on a given execution of a computational system. Typically an execution is abstracted as a *trace* (e.g. a finite sequence of observations called events) and runtime verification becomes checking whether this trace is in the language defined by the property. Checking can be *online* whilst the monitored program is running, or *offline* after the program has run. In either case, the monitored system must be *instrumented* to produce events of interest, either to be immediately processed by a monitor or to be stored in a log file.

I am particularly concerned with *parametric* runtime verification for Java programs. The term parametric refers to the events being observed being annotated with parameters, such as for example `open`(`readme.txt`) or `insert`($idABC$, `12`). Within the context of monitoring Java programs I will assume that parametric events are generated by method calls with the event name being the method name and the parameters being a relevant subset of the method parameters and target object[1]. We can, for convenience, assume that generation of such events is achieved by the Aspect-Oriented Programming (AOP) tool AspectJ. The (parametric) properties being checked are then defined in terms of these parametric events. *Parametric trace slicing* is then a method for checking parametric properties that involves *slicing* a parametric trace up based on parameter values and then processing each slice in separation.

I will now use a running example (which we will use throughout the paper) to clarify the notion of parametric property, introduce the technique of parametric trace slicing, and discuss the main ideas of this paper related to the detection of unreachable objects. I keep the presentation informal here in the introduction and make things (a little) more formal later. Figure 1 presents a Java method that takes a

[1] This assumption is not a restriction of the technique, but convenient for this paper.

Adrian Francalanza, Gordon J. Pace (Eds.): Second International Workshop
on Pre- and Post-Deployment Verification Techniques (PrePost 2017)
EPTCS 254, 2017, pp. 1–14, doi:10.4204/EPTCS.254.1

```
public static void writeToFile(String fileName, Collection records){
  File file = new File(fileName);
  file.open();
  Iterator iterator = records.iterator();
  while(iterator.hasNext()){
    file.write(iterator.next());
  }
  // file.close();
  records.removeAll();
}
```

Figure 1: Example Java program used as running example.

file name and a collection of records and writes the records to a file with the given name. I will consider three properties that I might want to check on this method.

- *HasNext*: For every iterator object i we only call i.`next`() if a preceding call of i.`hasNext`() returned true with no intermediate calls to i.`next`() or i.`hasNext`().
- *UnsafeIter*: For every collection c and iterator object i created from c, the iterator i is not used (e.g. by calls to i.`next`()) after c has been updated.
- *OpenClose*: For every file object f, the file cannot be written to or closed if not opened, cannot be opened once already open, and must eventually be closed once opened.

The first two represent *safety* properties whilst the third is a form of *response* property as the opening of a file creates an obligation that it is eventually closed. Figure 2 gives state machines for each of the properties (their semantics with respect to parameters will become clear later) where I make the assumption that states are closed to failure (to avoid an unnecessary clutter of transitions).

Let us use the *HasNext* property to illustrate the idea around parametric trace slicing. As the program is executed various Iterator objects will be created and used. Observing this will produce a *trace* of such calls. For example we might observe the trace

`hasNextTrue`(*iter1*).`next`(*iter1*).`hasNextTrue`(*iter1*).`hasNextTrue`(*iter2*).`next`(*iter2*).`next`(*iter1*).

where calls to *iter2* are nested within calls to *iter1*. The idea of parametric trace slicing is to *slice* this parametric trace into two propositional traces containing event names only and check these using standard methods (e.g. using a finite state automaton). Here the trace slice for *iter1* is `hasNextTrue.next`. `hasNextTrue.next` and for *iter2* it is just `hasNextTrue.next` and both slices satisfy the property.

These properties play the following role in this paper:

- For the *HasNext* property we can observe that the `iterator` object does not escape the above `writeToFile` method. This means that we could insert a call to the monitor at the end of the method indicating that the monitor no longer needs to track this object. This can be helpful to the monitoring effort as it eagerly reduces the size objects being stored by the monitor.
- The *UnsafeIter* propery allows us to apply a similar argument within the context of another object (`records`) still being potentially live.

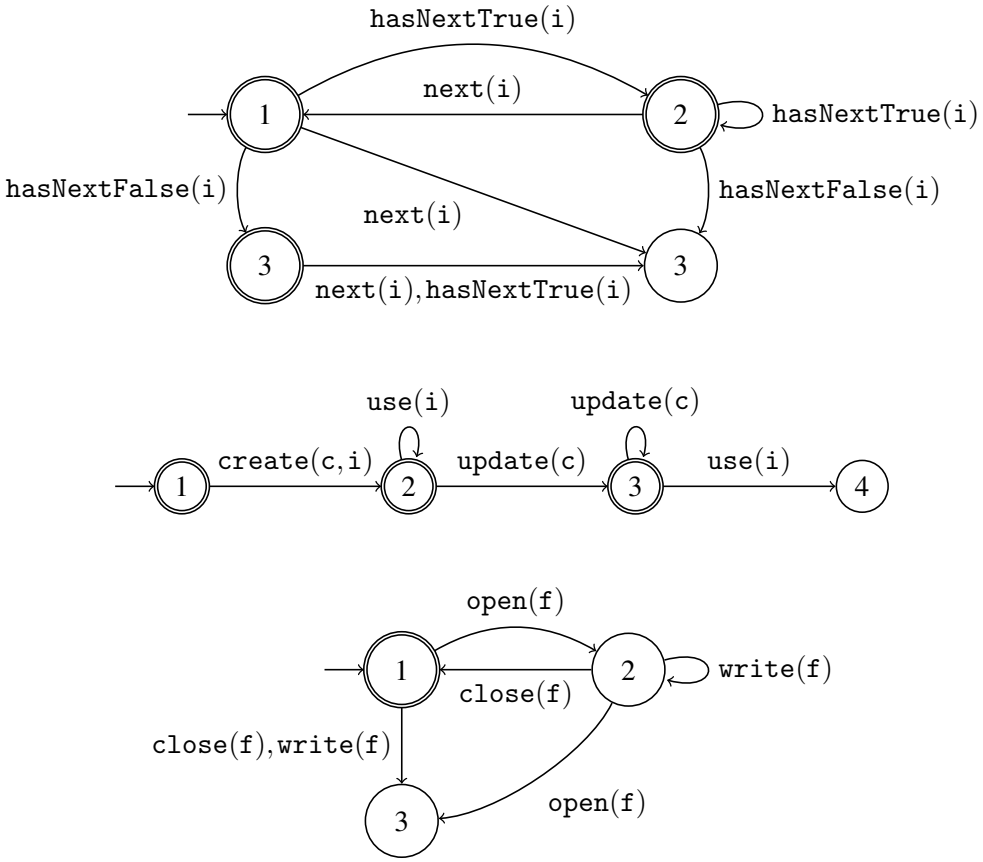

Figure 2: State machines for the three example properties.

- The *OpenClose* property allows us to discuss *anticipation*. Normally this property cannot be satisfied or violated on a finite prefix of a trace. However, it is common to consider the end of a program as indicating that no further events occur, which gives us a single infinite extension allowing us to decide satisfaction or violation. This approach therefore delays this decision to the end of the program, which has a number of issues. Now we can observe that the `file` object becomes unreachable at the end of this method. A good monitoring approach would detect a violation of this property when the associated object is garbage collected, as discussed below. However, eagerly detecting this unreachability and statically adding instrumentation alerting the monitor of this, allows the monitor to detect the violation as early as possible.

Contributions. Whilst this paper does not present an implementation of any of the ideas informally described above, it does present the background and ideas that form the basis for such an implementation. In Section 2 I review existing work on static detection of unreachable objects i.e. garbage, and in Section 3 I introduce parametric trace slicing and discuss the various ways in which information from this static analysis can help. I finish in Sections 4 and 5 by discussing the expected impact of these ideas and setting out my agenda for future work in this area.

2 Statically Detecting Unreachable Objects

Here I briefly review various approaches for statically determining the lifetime of an object i.e. points in the code where we can be certain that an object will become unreachable. The first part discusses general techniques (these standard techniques can be found in textbooks [13]) whilst the second part focusses on one particular approach.

2.1 The Intended Result

With respect to the overall idea being put forward by this paper, the actual static analysis techniques employed do not matter. What matters is the result of the analysis and how this can be used to optimise the runtime verification approach. Therefore, I will start by discussing what the result should be.

The goal is to perform a code transformation to insert instrumentation points into the program to record when an object will no longer be observed by the monitor. This should be sound i.e. such instrumentation points should only be added where it is certain that the object can no longer be observed. I introduce this at the (informal) level of instrumentation points as this is also the level at which I introduce the runtime verification approach later.

At this point I note that in general I assume the previous goal to mean that the object becomes unreachable in general. However, a stronger analysis would take a monitored property into account and attempt to identify objects whose use becomes irrelevant to that property. Whilst this might sound like something we want, it would most likely be prohibitively expensive.

The next two sections will review some existing static analysis approaches and is not meant to be a contribution, rather a review to establish that the above goal can be met.

2.2 Escape Analysis

Escape analysis [2, 6] is used to determine if an object escapes a method[2] usually with the goal of swapping heap allocation for stack allocation as a compiler optimisation aimed at reducing memory overhead (and garbage collector load).

This analysis is typically *flow-insensitive*, meaning that the order of statements in a method is ignored, and *intraprocedural*, meaning that the analysis operates on a single method at a time. So let's take a look at the method of Figure 1. We can identify (via calls to `new` and the `Iterator()` factory method[3]) two allocated objects `file` and `iterator` and extract the lines referring to these objects[4].

```
File file = new File(fileName);
file.open();
file.write(iterator.next());
```

```
Iterator iterator = records.iterator();
while(iterator.hasNext())
file.write(iterator.next());
```

It is straightforward to observe that none of these lines cause either object to escape the method. To check this programatically it is necessary to be able to identify precisely which statements cause an object to escape. There are three such kinds of statement:

[2] It is also applied to check if an object escapes a thread with the goal or removing synchronisation information, but this usage is not of current interest here. Perhaps it would be interesting to explore this later for automatically 'desynchronising' monitoring activities.

[3] Note that this requires us to explicitly identify this method as a factory method. Techniques for this are discussed later.

[4] Here I will present these ideas at the source-code level but operations could equally (and perhaps more effectively) be applied at the bytecode level.

1. Returning the object from the method
2. Passing the object to another method (including constructors)
3. Creating a reference to this object from another object (which also escapes)

Flow-insensitive analysis as described above provides an over-approximation of the escaping objects for two reasons. Firstly, it is based on the assumption that all lines in the code are reachable. Detecting unreachable parts of the code is called *dead code analysis*. One could imagine combining these analyses but I suspect that unreachable code is not that common. Secondly, consider the following code where the `file` variable is reassigned. The usages of the two files will be conflated in a flow-insensitive analysis, meaning that some non-escaping objects may not be identified.

```
...
File file = new File(fileName);
anotherMethod(file);
file = new File(anotherFileName);
...
```

A simple solution to this issue is to first place the code in *static single assignment* (SSA) form which requires every variable to be assigned to exactly once and provides *flow sensitivity* for local variables. This is a standard transformation and perhaps assuming that it will be applied is quite reasonable.

The next (standard) challenge is that of *aliasing* i.e. ensuring that all versions of a single object are tracked. For example in the following code the fact that `file2` does not escape tells us that `file1` does not escape as both variables *point to* the same object.

```
...
File file1 = new File(fileName);
File file2 = file;
anotherMethod(file2);
...
```

Various *pointer analysis* methods exist to determine sets of variables that necessarily or possibly point to the same heap-allocated objects. Once such sets have been computed (discussed further below) then the previous analysis is lifted to sets of aliasing variables.

Whilst this approach may identify some useful short-lived objects it has some drawbacks. Firstly, intraprocedural pointer analysis is likely to be prohibitively imprecise. Secondly, flow insensitivity is adequate for detecting escaping objects but for identifying unreachable objects early it is too conservative. Lastly, the common usage of *factory methods* and *pure methods* (e.g. not leaking the input parameter) is likely to significantly reduce the effectiveness of simple escape analysis for our goal. The next section introduces some existing ideas to handle these issues.

2.3 Free-Me Analysis

I now introduce some ideas from the *free-me analysis* introduced by Guyer et al. [9] in 2006 for identifying opportunities for early object reclamation. I am sure other equally interesting work exists (please feel free to draw my attention to it) but a discussion of this work suffices to convince us that our above goal is achievable.

Pointer Analysis. The free-me analysis maintains a points_to set for each variable (and a transitively-closed version). These sets are then modified in a flow-insensitive manner using the following rules:

```
File write(String value, boolean ret){
  File file = newTempFile(); // factory
      method
  file.write(value);
  if(ret){ return file; }
  return null;
}
```

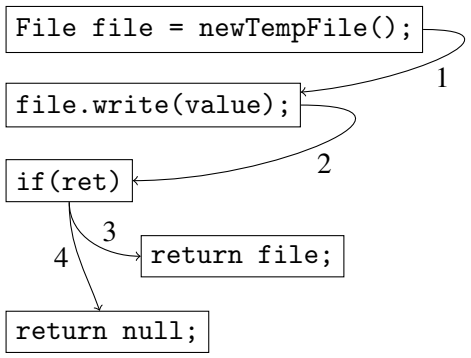

Figure 3: A small example with control flow graph.

Assignment	$v_1 = v_2$	$\mathsf{points\_to}(v_1) \cup = \mathsf{points\_to}(v_2)$
Field Access	$v_1 = v_2.f$	$\mathsf{points\_to}(v_1) \cup = \mathsf{points\_to\_transitive\_closure}(v_2)$
Field Assignment	$v_1.f = v_2$	$\forall n \in \mathsf{points\_to}(v_1) : \mathsf{points\_to}(n) \cup = \mathsf{points\_to}(v_2)$
Static Field Access	$v_1 = Cls.f$	$\mathsf{points\_to}(v_1) \cup = \mathsf{points\_to}(g)$
Static Field Assignment	$Cls.f = v_1$	$\mathsf{points\_to}(g) \cup = \mathsf{points\_to}(v_1)$

A special fresh variable g is used to represent all globally reachable objects and is used for global variables, static field accesses and in method summaries (discussed below). The rule for field assignment is somewhat non-standard but its conservatism allows for simple method summaries.

Method Summaries. A method summary is a set of pairs (p,q) where q is an input parameter and p is either an input parameter or the special labels *global* or *return*. The semantics of (p,q) is that after the method is called the object pointed to by q is reachable from p. In the above pointer analysis this allows points_to sets to be appropriately updated after method calls (for virtual methods all matching method summaries are applied). The analysis also identifies factory methods as those whose return object is always a newly allocated object and includes this information in the summary. In the pointer analysis factory methods are replaced by new allocation sites.

Liveness Analysis. Consider the method in Figure 3 where the `file` object possibly escapes the method but there is one path through the method where it does not. Ideally we would add an instrumentation call just before the final line to indicate that the current version of the `file` object will become unreachable. But the escape analysis described above will not achieve this.

The idea is to apply traditional *liveness analysis* [11] to identify at each program point which variables contain values that may be needed later. This is applied as a backward analysis on the control flow graph where liveness is introduced at points where a variable is used and propagated backwards with unions being taken at merge points. This is applied at the granularity level of single statements (rather than basic blocks) as the usage is different from that of the traditional analysis.

This information is combined with pointer analysis to give *reachability* for the objects associated with each allocation point at each edge in the control flow graph. However, the flow-insensitivity of the pointer-analysis provides an over-approximation of reachability if, for example, an object eventually becomes globally reachable. The analysis introduces a concept of *potential liveness* restricting the inherited liveness status from global variables to after the associated assignment. Additionally, objects originating from factory methods are not marked as globally reachable due to the above approach of replacing such calls by allocations.

The described approach allows us to identify that edge 4 in the small control flow graph of Figure 3 is unreachable for the object pointed to by `file`. Further details are beyond this review and I invite the reader to refer to the original text *free-me analysis*.

Inserting Calls. The above analysis will identify, for each allocation site, a set of program points (edges in the control flow graph) where the object becomes unreachable. Instrumentation calls can be inserted at each site but some points will dominate others and the earliest of these should be selected. It is important to ensure that an object is not flagged as unreachable multiple times; although this is far less important in our setting than in the setting of the original work. The trick used by the original approach is to use a temporary variable that is set to `null` after it is freed and a freeing method that detects and ignores values set to `null`.

3 Parametric Trace Slicing, Monitoring, and Garbage

Here we introduce parametric trace slicing and its relation to monitoring whilst also considering how the garbage information identifiable by the techniques in the previous section can be used to optimise monitoring. The monitoring approach presented here is based on the original work of Roşu and Chen [5], which was later adapted by myself and others [1] in the definition of quantified event automata (QEA). Whilst QEA will feature at the end of this story, most of the ideas here are directly related to the original work around JavaMOP [12].

3.1 The Underlying Approach

We assume disjoint sets of event names Σ, variables X and values D. A valuation θ is a map (partial function with finite domain) from variables to values. The submap operator $\sqsubseteq$ and least-upper-bound operator $\sqcup$ on maps are defined as usual. We write $\mathsf{dom}(\theta)$ for the domain of θ.

A *parametric event signature* is a pair consisting of an event name e and a list of variables $x_1,\ldots,x_n$ written $e(x_1,\ldots,x_n)$. A *parametric event alphabet* is a finite set of parametric event signatures with at most one signature per event name. For a given parametric event alphabet A, a *parametric event* is a pair consisting of an event name e and a valuation θ, written $e(\theta)$, such that there exists $e(x_1,\ldots,x_n) \in A$ such that $\{x_1,\ldots,x_n\} = \mathsf{dom}(\theta)$. A *parametric trace* is a *finite* sequence of parametric events and a *propositional trace* is a finite sequence of event names. Let ε be the empty sequence. A *parametric property* for a parametric event alphabet A is a predicate over A^* (parametric traces over A). A *propositional property* is a predicate over Σ^*.

We assume an instrumentation method that extracts events from programs. There is an implicit assumption that an event name corresponds to a method name and the associated variables correspond to a subset of method parameters and return value. This is not necessary but standard, particularly in the early work on parametric trace slicing. We note that assuming that a method call can only correspond to a single event in the system limits expressiveness, which is dealt with in the QEA work [1] but this solution complicates matters and we ignore it here.

For the purposes of this paper, a *parametric specification* is a finite state automaton with a parametric event alphabet. Let the *propositional abstraction* of this automaton be the one given by preserving event names only i.e. with alphabet Σ.

Next we will define how a parametric specification describes a parametric property. The idea is to use a slicing operator to transform the trace-checking problem for this parametric property into one of

checking a propositional property on a set of propositional traces. The slicing operator is defined as follows.

Definition 1 (Parametric Trace Slicing) *Given valuation θ and parametric trace τ let $\tau\downarrow_\theta$ be the propositional trace-slice for θ defined as follows.*

$$\begin{array}{lcll} \varepsilon\downarrow_\theta & = & \varepsilon & \\ e(\theta')\tau\downarrow_\theta & = & e(\tau\downarrow_\theta) & \text{if } \theta'\sqsubseteq\theta \\ & & (\tau\downarrow_\theta) & \text{otherwise} \end{array}$$

This preserves those event names where the valuation is relevant to (included in) θ.

Example 1 *Given a valuation $\theta=[c\mapsto A, i\mapsto B]$ and parametric trace*

$$\tau = \texttt{use(D) create(A,B) create(A,C) use(B) use(C) update(A) use(B)}$$

the corresponding trace slice for θ is

$$\tau\downarrow_\theta = \texttt{create use update use}$$

The possible valuations used in slicing are dependent on the parametric trace being checked i.e. they are built from (parts of) those valuations observed at runtime. We define such valuations as follows.

Definition 2 (Induced Valuations) *Given a trace τ the valuation θ is* induced *by τ if (i) there is an event $e(\theta')\in\tau$ such that $\theta'\sqsubseteq\theta$, and (ii) for every $(x\mapsto v)\in\theta$ there exists an event $e(\theta')\in\tau$ such that $(x\mapsto v)\in\theta'$.*

Example 2 *For the trace τ from Example 1 we have the induced valuations*

$$[c\mapsto A, i\mapsto B] \quad [c\mapsto A, i\mapsto C] \quad [c\mapsto A, i\mapsto D]$$

although the last valuation can be identified as redundant *using techniques not described here.*

By construction, if a valuation θ is not induced by τ then $\tau\downarrow_\theta=\varepsilon$. If the initial state of a parametric specification is accepting then this means that restricting to induced valuations does not alter the next definition but we do not enforce this restriction.

Finally, we can define the parametric traces accepted by a parametric specification as follows.

Definition 3 (Trace Acceptance) *A parametric specification Γ accepts a parametric trace τ if for every valuation θ induced by τ we have*

$$\tau\downarrow_\theta\in\mathscr{L}(P)$$

where $\mathscr{L}(P)$ is the language of the propositional abstraction of Γ.

Example 3 *Given the valuation θ and trace τ from Example 1 and the parametric specification for the* UnsafeIter *proeprty given in Figure 2. We can see that the slice $\tau\downarrow_\theta$ given in Example 1 is not in the language of the propositional abstraction as it reaches state 4, which is non-final. Therefore, the trace is not accepted as there is at least one induced valuation where the given property does not hold.*

We have now defined the semantics of our parametric specifications. Note that Figure 2 already defined three parametric specifications.

To check a trace against a parametric property then requires three steps. The first step is to construct the set of valuations, which first requires extracting values from the trace. This set is likely to be very large and theoretically exponential in the length of the trace, although usually much smaller in practice. The second step is to slice the trace to produce a slice per valuation. The final step is to check each slice against the underlying property. Clearly separating monitoring into three steps like this is not practical. The next question is how we efficiently monitor such specifications without separating monitoring into three separate steps.

Algorithm 1 An incremental algorithm for performing parametric trace slicing.

```
1: Let Lookup be a map from valuations to states initial mapping the empty valuation to the initial state
2: for event e(θ) ∈ τ do
3:     for θ′ in dom(Lookup) from biggest to smallest do
4:         if θ is consistent with θ′ then
5:             if θ′ ⊑ θ then
6:                 Update Lookup(θ′) using e
7:             else if θ ⊔ θ′ is not in dom(Lookup) then
8:                 Add θ ⊔ θ′ to Lookup using Lookup(θ′) updated using e
9: If an entry in Lookup is in a non accepting state then Fail otherwise Accept
```

3.2 Monitoring Algorithms for Parametric Trace Slicing

The above semantics is non-incremental due to the need to compute induced valuations before performing slicing. A number of algorithms exist for incremental monitoring [12, 15]. One of the most simple of these approaches is captured in Algorithm 1. Here trace slices are represented by the state reached in the (propositional abstraction of the) parametric specification. The idea is to, for each incoming event, search through existing valuations and (i) update the information for the associated trace slice if the event is relevant, and (ii) add new valuations if they do not exist. Iterating through existing valuations from biggest to smallest is necessary to ensure that the valuation storing the most information about a trace slice is used when adding a new valuation (this idea is called maximality in other work).

I note two things. Firstly, that this algorithm is heavily dependent on the size of Lookup, which in the given algorithm never shrinks. And secondly, the algorithm is inefficient as each step is linear in the number of stored valuations, and in the worse case the number of valuations can be exponential in the length of the trace seen so far e.g. $v^{|\tau|}$ where v is the number of variables used in the parametric specification. Although this assumes heavy reuse of values in the trace and typically the number of valuations does not grow exponentially, but may still grow super-linearly.

Efficient monitoring approaches relying on complex indexing data structures have been introduced (see [12, 15]) but these remain super-linearly related to the number of valuations as they employ redundancy to ensure efficient indexing. Whilst this is often a suitable trade-off, there is still a cost associated with maintaining such structures and access operations remain somewhat proportional to the size of the structures. The message here is that *the number of valuations being tracked directly impacts efficiency*. We Therefore, reducing this number (for example by detecting that objects in a valuation become unreachable and relevant events can no longer be observed) will have a positive effect.

3.3 Anticipation

The reader may have noticed a problem with Algorithm 1. Even though the trace is processed incrementally the result of whether the full trace is accepted only comes at the end. It is possible to report on the acceptance of the current prefix (i.e. what would happen if the trace ended here) but this does not necessarily relate to the final verdict.

What we want to do is *anticipate* the final verdict as soon as possible. For the *HasNext* and *UnsafeIter* properties given in Figure 2 this is relatively straightforward as they are safety-properties. As soon as any entry in Lookup enters a non-accepting state there is no way for the trace to be accepted and final violation can be reported straight away. For the *OpenClose* property the safety element can be detected

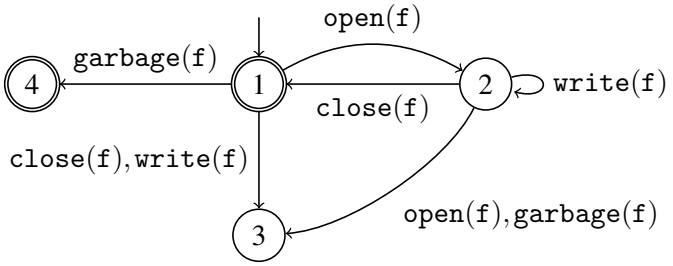

Figure 4: Adding `garbage` events to the *OpenClose* automaton..

early but the eventually closing a file part cannot (as it stands).

What is the formal idea here? If the current trace cannot be *extended* to an accepting trace then it can be marked as non-accepting. We can capture the notion of possible extensions by *reachability* in the finite state automaton. In the *HasNext* finite state automata failure is captured by the explicit failure state 3 where, whereas in the *UnsafeIter* automata there is (in addition to the explicit state 4) an implicit failure state where, by construction, no accepting states are reachable. In the *OpenClose* automata state 3 has no reachable accepting states so any entries in Lookup entering this state can report a final violation.

This is the first point where detection of unreachable objects can help. If, for the *OpenClose* property we can detect that a file object becomes unreachable when the associated automaton is in the non-accepting state 2 then we know that this cannot be remedied later. Instead of adding special code to the monitoring approach to handle this case[5] we can simply modify the automaton for the property and add a `garbage` event (produced by the techniques discussed earlier) to the alphabet of the property. Figures 4 and 5 show how this can be done for the *OpenClose* and *UnsafeIter* properties respectively. Importantly, the addition of garbage events can be an automatic transformation applied to automata.

An observation here is that for the *UnsafeIter* property if c becomes garbage at state 3 it is still necessary to continue monitoring i as a violation can still occur, however if this occurs earlier then the property cannot be violated any longer. Furthermore, detecting that i becomes garbage means that the property can never be violated as the only paths to a non-accepting state involve the i object. Therefore, the valuation can be removed if i becomes garbage (which is the case in our example from Figure 1) but not necessarily if c becomes garbage.

3.4 Garbage-Aware Indexing

Indexing structures employed by the monitoring algorithms mentioned above are typically *garbage-aware* in the sense that they employ *weak references* so that once all objects pointed to by a valuation become unreachable that entry in the data structure can be removed. The initial implementation of this idea in JavaMOP was incorrect as it did not allow for the anticipation of failure as described above (which led to missing some property violations), this was fixed later [10].

The usage of weak references is important as it prevents the monitor *leaking* memory and it keeps data structures small. However, doing this via garbage collection has a disadvantage as objects are not collected until space is needed. Therefore, the second usage of static garbage information would be to eagerly reduce the size of these data structures by directly reporting this garbage.

[5] Although I note that it is standard to have special monitoring code, along with the usage of *weak references*, to handle such cases, as discussed later.

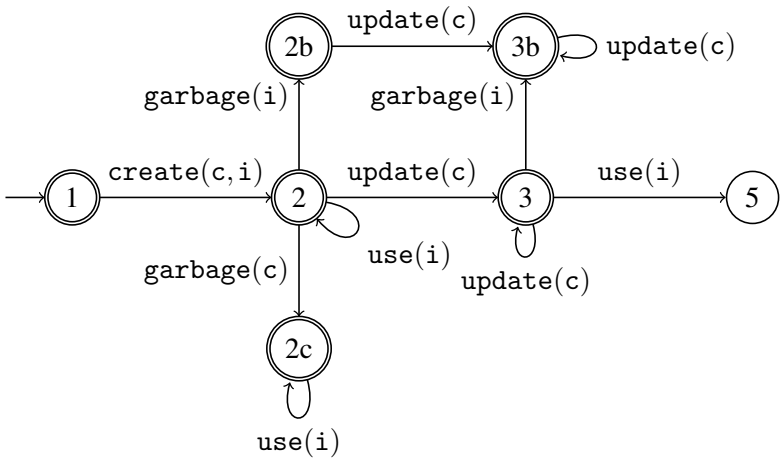

Figure 5: Adding `garbage` events to the UnsafeIter property.

However, I note that this may have limited impact as the garbage issues are larger for longer-living objects whilst the static analysis techniques proposed earlier are more suitable for short-lived objects. Nevertheless, in situations where many short-lived objects are being monitored at the same time I see this as potentially having a positive impact.

3.5 Monitoring Representative Objects

This idea is the least mature but also has the potential for having the most impact on monitoring overhead. My observation is that when we perform static analysis we consider all objects produced by a single allocation point together. In runtime verification we do not do this; every created object is monitored separately. This makes sense for long-lived objects with many possible paths through the program but for short-lived objects it is a large waste of monitoring time.

For example, in the method given in Figure 1 we should only monitor `file` and `iterator` once although this becomes less clear in the presence of the loop. If this method is called frequently then only monitoring the method once will have a massive impact on overhead.

The proposal is to detect objects that do not escape a method and add flags to indicate whether that object has been checked on all paths through the method. The reason that escape analysis is a key factor here is that we want to be able track all paths of a single object and doing this interprocedurally would be prohibitive. For our example method it is straightforward as there is a single path. For methods with complex control flow this may be prohibitively complicated. In essence, the proposal is to monitor an *abstraction* of observed concrete objects and whilst the idea here is to use an allocation-site abstraction, this could be further augmented for greater precision.

A related approach was previously proposed by Dwyer et al. in 2010 [7]. The idea there was to optimise loops to reduce monitoring overhead by detecting repeated iterations that did not differ in their effect, unrolling them, and only monitoring the first iteration. It appears that Dwyer et al.'s idea is related to this one and but has not been fully explored.

3.6 Offline Monitoring

A small but sometimes important point is that in implementations of parametric trace slicing the notion of equality is typically referential. This is reasonable for online monitoring as at any one time only one object can occupy a particular memory address. However, for offline monitoring where events are recorded in a log file and then read in later for checking there needs to be a way of unambiguously recording object identities. Simply printing the identity hash code is insufficient as there is the possibly of a clash with an object created later. Whilst this sounds improbable, I have experienced this in practice.

The solution is to either add allocation or garbage information to the log file. Allocation information is not always appropriate as we may only be monitoring a subset of a certain type of object that is used in a particular way and logging all allocations may increase the size of the trace by orders of magnitude.

Currently, I use a trick that creates objects with a custom `finalizer` weakly reachable by a monitored object such that collection of this object records unreachability of the monitored object. However this relies on finalizers, which are not guaranteed to be run, and requires additional objects. Therefore, static detection and reporting of unreachability is preferable (although sadly not a general solution).

4 Discussion or Will it Work?

The previous section proposed a number of approaches for using static information about unreachable objects to optimise the runtime verification effort. Here I discuss what impact I might expect these to have in general. Before doing so it is worth mentioning that tools such as JavaMOP and MarQ depend heavily on garbage information at runtime and without ideas such as garbage-aware indexing they would struggle with a real-world trace of any significant length. However, I note that whilst the impact of garbage has been studied [10], the previous view of representing garbage within the property being monitored is new.

Risks and Limitations. The main limitation of this approach is the scope of applicability. The static analyses discussed earlier are only applicable to objects that are are short lived and (relatively) local to a single method. Arguably, in such cases it could be relatively easy to apply static techniques to completely verify a property. Indeed, the more interesting properties for runtime verification are those involving objects and events spread out over time and the codebase. Due to realistic language features (such as reflection and dynamic loading) it is very difficult to make the described static analyses sound inter-procedurally i.e. many objects will be conservatively marked as reachable rendering the approach ineffective for non-local properties. Whilst more modern approaches [16, 17] may provide better precision, this will remain a limitation of the approach (however, see the other side of this below).

A related point to the above is that for such short-lived objects it is highly likely that modern generational garbage collection will collect objects quickly, providing the desired quick anticipation of liveness violations. The current use of weak references generally appears adequate and there is a risk that any advantage provided by static information could be represent a small increment at best.

Perceived Strengths. A counter to the above point about short-lived objects is that this approach can make use of partial information (where an object is unreachable in some paths only) in situations where intraprocedural static checking would not apply. Additionally, for properties involving multiple objects, this approach may only identify that one object becomes unreachable, which would not be usable information statically, but could have significant impact dynamically.

I expect the idea about monitoring representative objects to be the most fruitful in the long-term. Whilst the other ideas may decrease overhead per monitored object, this approach has the potential to remove the need to monitor a large number of objects at all.

5 Conclusion

This paper has explored the idea of statically identifying unreachable objects and then using this information to optimise runtime verification using parametric trace slicing by

- anticipating failures of non-safety properties sooner than otherwise possible,
- keeping indexing data structures small,
- reducing the number of objects being monitored (in restricted circumstances), and
- dealing with a known issue with offline monitoring in this setting.

None of these ideas have been implemented yet but the intention is to incorporate them into the MarQ [15] tool. I note that supporting the additional features of QEA beyond those captured by the parametric specifications described in this paper may require some extra work. For example, QEA also allow objects to be captured by so-called *free variables* that are not involved in slicing and then for transitions to be guarded by predicates on these variables. This dramatically increases the complexity of the reachability question as it must consider the satisfiability of these guards.

As a final comment, this work began in [14] by exploring how the typestate analysis techniques employed by Clara [4, 3] could be applied to QEA. The idea was to utilise pointer analysis information to statically check non-safety properties. However, I then noticed that the same information could be used to optimise the runtime activity in different ways. This work looking at typestate analysis for QEA is also ongoing.

Acknowledgements. I would like to thank the reviewers for their helpful comments that helped improve the text and also provide further ideas to explore.

References

[1] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger & David E. Rydeheard (2012): *Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors*. In: *FM*, pp. 68–84. Available at `http://dx.doi.org/10.1007/978-3-642-32759-9_9`.

[2] Bruno Blanchet (1998): *Escape Analysis: Correctness Proof, Implementation and Experimental Results*. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, ACM, New York, NY, USA, pp. 25–37, doi:`10.1145/268946.268949`.

[3] Eric Bodden & Laurie J. Hendren (2012): *The Clara framework for hybrid typestate analysis*. *STTT* 14(3), pp. 307–326, doi:`10.1007/s10009-010-0183-5`. Available at `https://doi.org/10.1007/s10009-010-0183-5`.

[4] Eric Bodden, Patrick Lam & Laurie Hendren (2010): *Clara: A Framework for Partially Evaluating Finite-state Runtime Monitors Ahead of Time*. In: *Proceedings of the First International Conference on Runtime Verification*, RV'10, Springer-Verlag, Berlin, Heidelberg, pp. 183–197, doi:`10.1007/978-3-642-16612-9_15`.

[5] Feng Chen & Grigore Roşu (2009): *Parametric Trace Slicing and Monitoring*. In: *TACAS '09*, Berlin, Heidelberg, pp. 246–261, doi:`10.1007/978-3-642-00768-2_23`.

[6] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar & Sam Midkiff (1999): *Escape Analysis for Java*. *SIGPLAN Not.* 34(10), pp. 1–19, doi:10.1145/320385.320386.

[7] Matthew B. Dwyer, Rahul Purandare & Suzette Person (2010): *Runtime Verification in Context: Can Optimizing Error Detection Improve Fault Diagnosis?*, pp. 36–50. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-16612-9_4.

[8] Y. Falcone, K. Havelund & G. Reger (2013): *A Tutorial on Runtime Verification*. In Manfred Broy & Doron Peled, editors: *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems*, IOS Press, doi:10.3233/978-1-61499-207-3-141.

[9] Samuel Z. Guyer, Kathryn S. McKinley & Daniel Frampton (2006): *Free-Me: A Static Analysis for Automatic Individual Object Reclamation*. *SIGPLAN Not.* 41(6), pp. 364–375, doi:10.1145/1133255.1134024.

[10] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith & Grigore Roşu (2011): *Garbage Collection for Monitoring Parametric Properties*. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'11)*, ACM, pp. 415–424, doi:10.1145/1993498.1993547.

[11] Uday Khedker, Amitabha Sanyal & Bageshri Karkare (2009): *Data Flow Analysis: Theory and Practice*, 1st edition. CRC Press, Inc., Boca Raton, FL, USA, doi:10.1201/9780849332517.

[12] Patrick Meredith, Dongyun Jin, Dennis Griffith, Feng Chen & Grigore Roşu (2011): *An overview of the MOP runtime verification framework*. *J Software Tools for Technology Transfer*, pp. 1–41. Available at http://dx.doi.org/10.1007/s10009-011-0198-6.

[13] Flemming Nielson, Hanne R. Nielson & Chris Hankin (2010): *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

[14] Giles Reger (2016): *Considering Typestate Verification for Quantified Event Automata*. In: *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*, doi:10.1007/978-3-319-23820-3_21.

[15] Giles Reger, Helena Cuenca Cruz & David Rydeheard (2015): *MarQ: monitoring at runtime with QEA*. In: *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, doi:10.1007/978-3-662-46681-0_55.

[16] Yannis Smaragdakis & George Balatsouras (2015): *Pointer Analysis*. *Found. Trends Program. Lang.* 2(1), pp. 1–69, doi:10.1561/2500000014.

[17] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali & Eric Bodden (2016): *Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java*. In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pp. 22:1–22:26, doi:10.4230/LIPIcs.ECOOP.2016.22. Available at https://doi.org/10.4230/LIPIcs.ECOOP.2016.22.