A Story of Parametric Trace Slicing, Garbage and Static Analysis

Giles Reger

School of Computer Science, University of Manchester, UK

PrePost 2017

Context

Helped develop the Quantified Event Automata (QEA) language and associated MarQ runtime monitoring tool

Have started thinking about typestate-analysis for QEA, wrote about it at ISoLA 2016

This idea grew out of that and I thank Adrian for encouraging me to write the idea down

Introduction

In this talk I will outline some ideas around how we can relate the ideas of

- Garbage collection at runtime
- Static identification of object unreachability

to improve the performance of runtime monitoring based on parametric trace slicing

Note that we are explicitly exclusively in the realms of monitoring `Java` programs using a monitor that shares the same JVM.

These ideas haven't yet been implemented but the intention is to realise them in the MarQ runtime monitoring tool for QEA

The Idea

At a high level:

- Parametric trace slicing is a runtime monitoring approach that tracks the behaviour of groups of objects
- By detecting when some of those objects become garbage we can
  - Optimise the monitoring algorithm
  - Potentially detect violations of co-safety properties
- But there can be a delay before something is recognised as garbage
- The idea is to statically identify points where an object will become unreachable to insert explicit garbage events

Now I will introduce parametric trace slicing and how it can be improved by garbage detection and then discuss how static analysis can play a part

Overview

1 Parametric Trace Slicing

2 Online Monitoring and Garbage

3 Static Analysis

4 What's Next?

Parametric Trace Slicing

Used first in `tracematches` but named and extended to total matching in the JavaMOP work. Later adopted by the QEA language (and others)

A solution for parametric runtime monitoring concerned with events that carry parameters

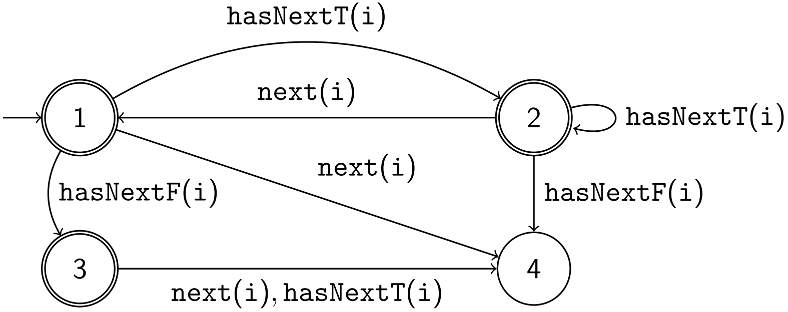
The philosophy behind the approach is to slice a trace based on the values of parameters and to consider each slice separately

I will introduce the idea by example

HasNext Example

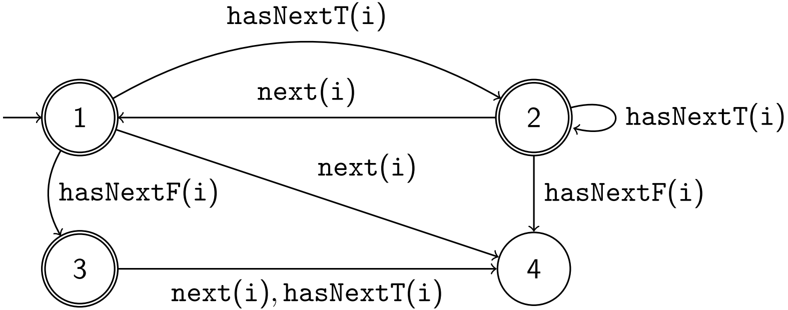
HasNext

For every iterator object *i* (instance of `java.util.Iterator`) we only call *i*.`next()` if a preceding call of *i*.`hasNext()` returned true with no intermediate calls to *i*.`next()` or *i*.`hasNext()`.



HasNext Example

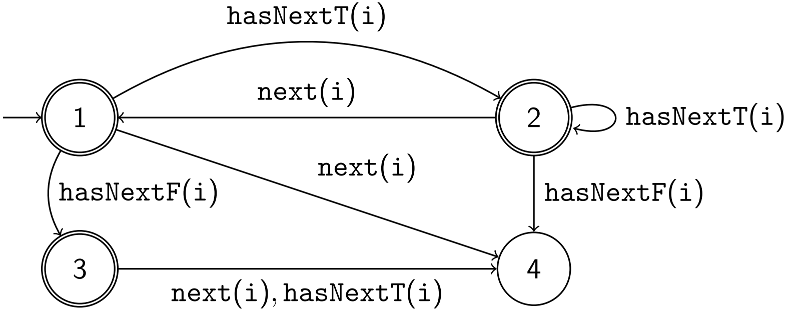
hasNextT($i1$) next($i1$) hasNextT($i1$) hasNextF($i2$) next($i2$) next($i1$)



HasNext Example

$\texttt{hasNextT}(i1)\ \texttt{next}(i1)\ \texttt{hasNextT}(i1)\ \texttt{hasNextF}(i2)\ \texttt{next}(i2)\ \texttt{next}(i1)$

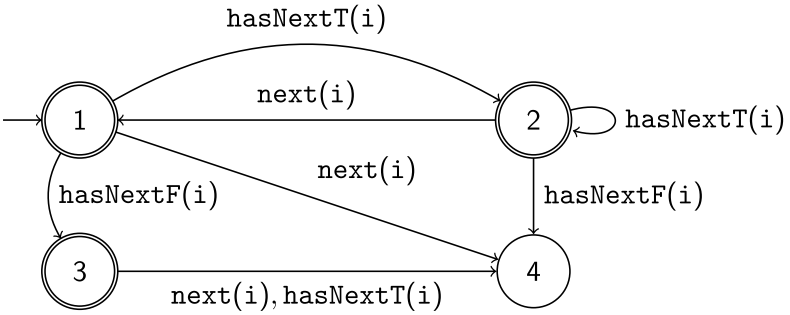
$[i \mapsto i1] \mapsto$
$[i \mapsto i2] \mapsto$



HasNext Example

hasNextT(*i1*) next(*i1*) hasNextT(*i1*) hasNextF(*i2*) next(*i2*) next(*i1*)

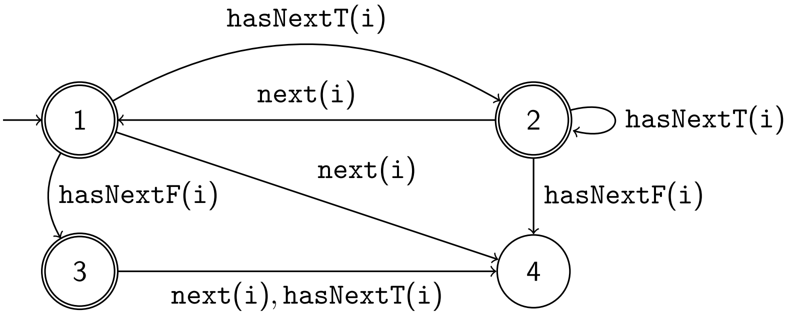
$[i \mapsto i1] \mapsto$ hasNextT(*i1*)
$[i \mapsto i2] \mapsto$



HasNext Example

hasNextT(*i1*) next(*i1*) hasNextT(*i1*) hasNextF(*i2*) next(*i2*) next(*i1*)

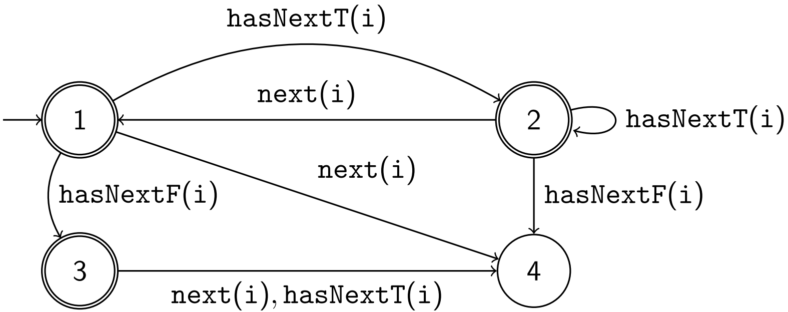
$[i \mapsto i1] \mapsto$ hasNextT(*i1*) next(*i1*)
$[i \mapsto i2] \mapsto$



HasNext Example

hasNextT($i1$) next($i1$) **hasNextT($i1$)** hasNextF($i2$) next($i2$) next($i1$)

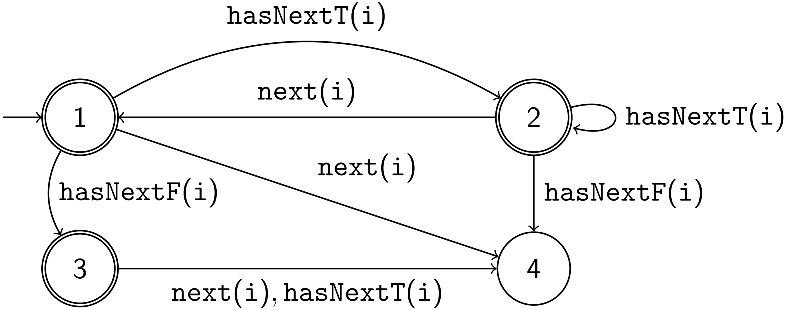
$[i \mapsto i1] \mapsto$ hasNextT($i1$) next($i1$) **hasNextT($i1$)**
$[i \mapsto i2] \mapsto$



HasNext Example

hasNextT(*i1*) next(*i1*) hasNextT(*i1*) hasNextF(*i2*) next(*i2*) next(*i1*)

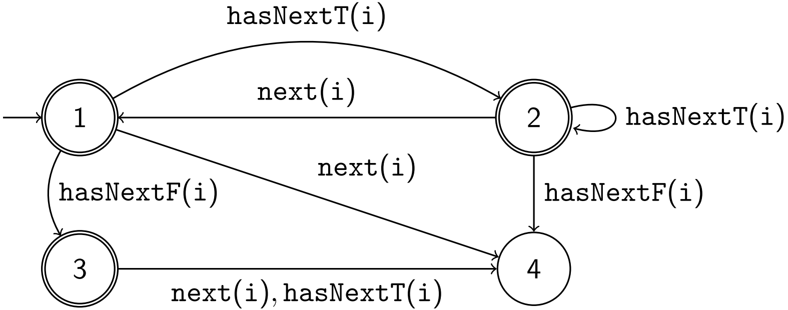
$[i \mapsto i1] \mapsto$ hasNextT(*i1*) next(*i1*) hasNextT(*i1*)
$[i \mapsto i2] \mapsto$ hasNextF(*i2*)



HasNext Example

hasNextT(*i1*) next(*i1*) hasNextT(*i1*) hasNextF(*i2*) next(*i2*) next(*i1*)

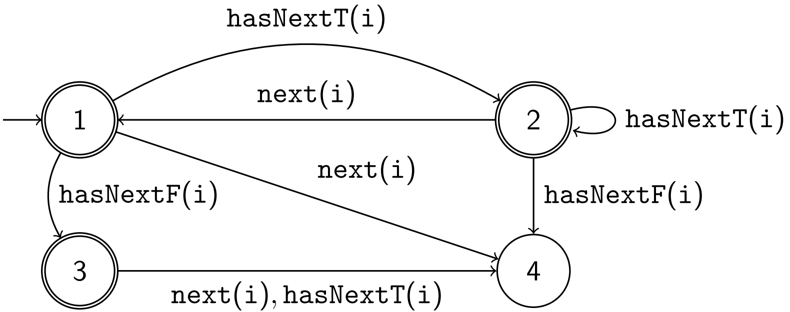
$[i \mapsto i1] \mapsto$ hasNextT(*i1*) next(*i1*) hasNextT(*i1*)
$[i \mapsto i2] \mapsto$ hasNextF(*i2*) next(*i2*)



HasNext Example

hasNextT($i1$) next($i1$) hasNextT($i1$) hasNextF($i2$) next($i2$) next($i1$)

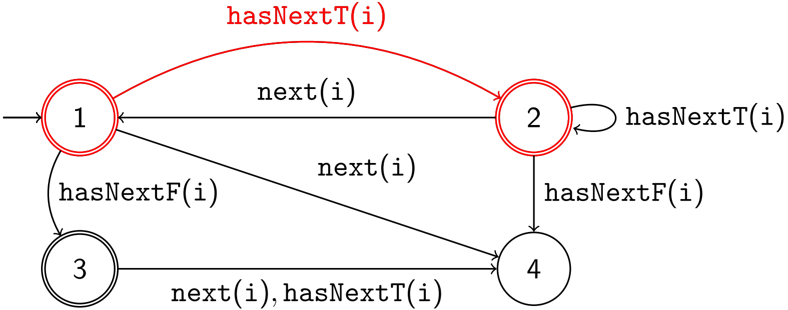
$[i \mapsto i1] \mapsto$ hasNextT($i1$) next($i1$) hasNextT($i1$) next($i1$)
$[i \mapsto i2] \mapsto$ hasNextF($i2$) next($i2$)



HasNext Example

hasNextT(*i1*) next(*i1*) hasNextT(*i1*) hasNextF(*i2*) next(*i2*) next(*i1*)

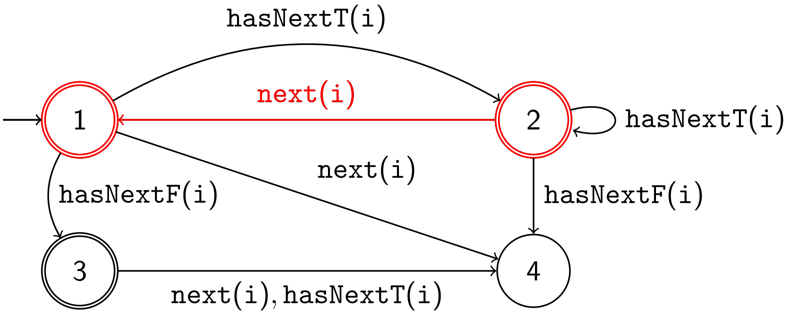
$[i \mapsto i1] \mapsto$ hasNextT(*i1*) next(*i1*) hasNextT(*i1*) next(*i1*)
$[i \mapsto i2] \mapsto$ hasNextF(*i2*) next(*i2*)



HasNext Example

hasNextT($i1$) next($i1$) hasNextT($i1$) hasNextF($i2$) next($i2$) next($i1$)

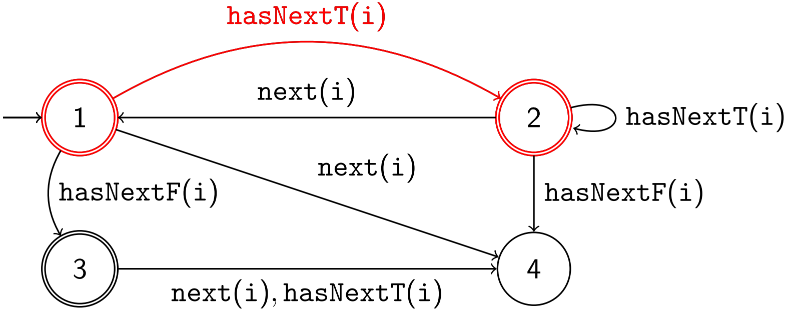
[$i \mapsto i1$] $\mapsto$ hasNextT($i1$) next($i1$) hasNextT($i1$) next($i1$)
[$i \mapsto i2$] $\mapsto$ hasNextF($i2$) next($i2$)



HasNext Example

hasNextT(*i1*) next(*i1*) hasNextT(*i1*) hasNextF(*i2*) next(*i2*) next(*i1*)

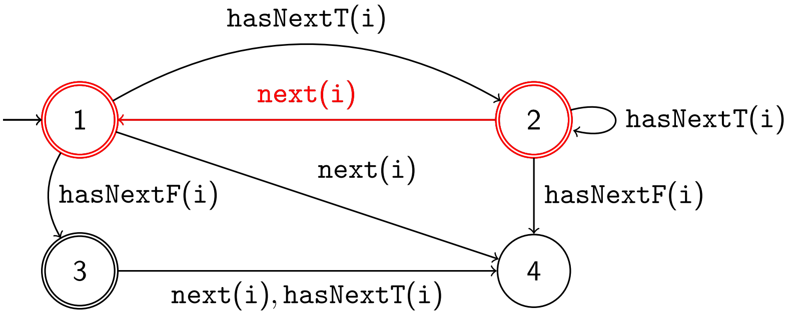
[*i* ↦ *i1*] ↦ hasNextT(*i1*) next(*i1*) hasNextT(*i1*) next(*i1*)
[*i* ↦ *i2*] ↦ hasNextF(*i2*) next(*i2*)



HasNext Example

hasNextT($i1$) next($i1$) hasNextT($i1$) hasNextF($i2$) next($i2$) next($i1$)

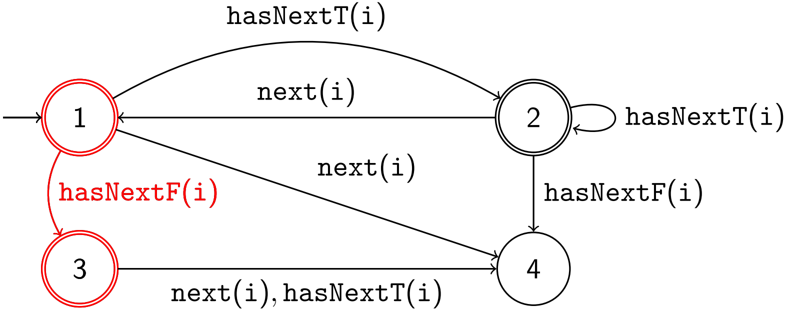
$[i \mapsto i1] \mapsto$ hasNextT($i1$) next($i1$) hasNextT($i1$) next($i1$) ✓
$[i \mapsto i2] \mapsto$ hasNextF($i2$) next($i2$)



HasNext Example

hasNextT(*i1*) next(*i1*) hasNextT(*i1*) hasNextF(*i2*) next(*i2*) next(*i1*)

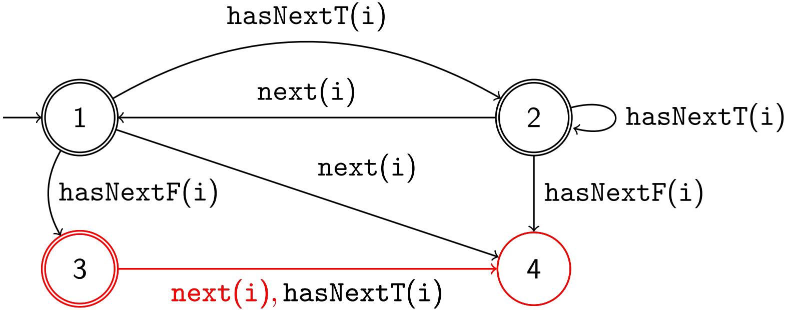
[*i* ↦ *i1*] ↦ hasNextT(*i1*) next(*i1*) hasNextT(*i1*) next(*i1*) ✓
[*i* ↦ *i2*] ↦ hasNextF(*i2*) next(*i2*)



HasNext Example

hasNextT($i1$) next($i1$) hasNextT($i1$) hasNextF($i2$) next($i2$) next($i1$)

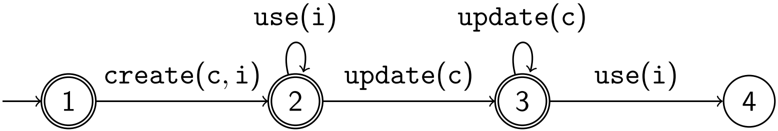
$[i \mapsto i1] \mapsto$ hasNextT($i1$) next($i1$) hasNextT($i1$) next($i1$) ✓
$[i \mapsto i2] \mapsto$ hasNextF($i2$) next($i2$) X



UnsafeIter Example

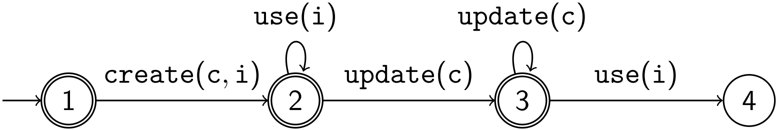
UnsafeIter

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.



UnsafeIter Example

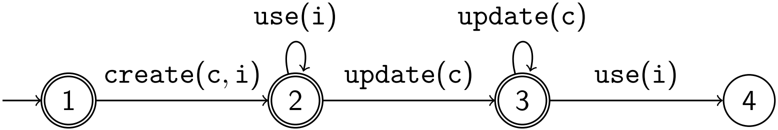
create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)



UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

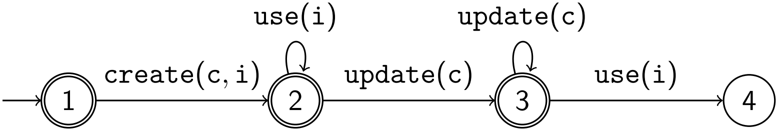
$[c \mapsto A, i \mapsto i1] \mapsto$
$[c \mapsto A, i \mapsto i2] \mapsto$



UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

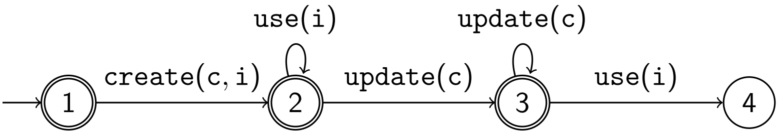
$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1)
$[c \mapsto A, i \mapsto i2] \mapsto$



UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

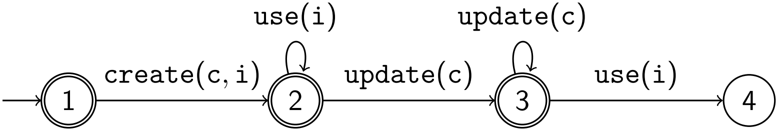
$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1) use(i1)
$[c \mapsto A, i \mapsto i2] \mapsto$



UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

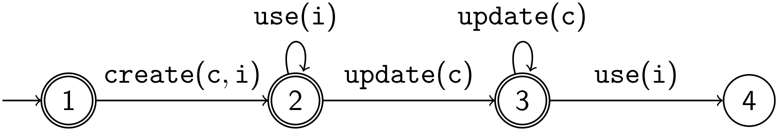
$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1) use(i1)
$[c \mapsto A, i \mapsto i2] \mapsto$ create(A, i2)



UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

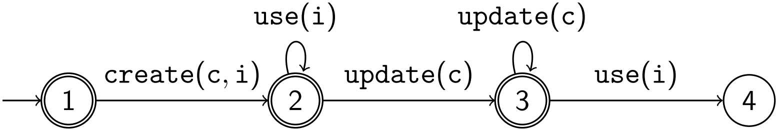
$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1) use(i1)
$[c \mapsto A, i \mapsto i2] \mapsto$ create(A, i2) use(i2)



UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

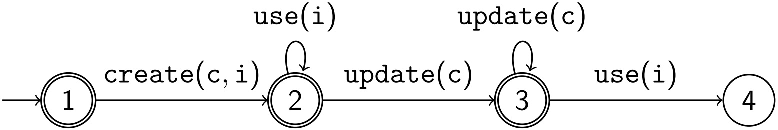
$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1) use(i1) update(A)
$[c \mapsto A, i \mapsto i2] \mapsto$ create(A, i2) use(i2) update(A)



UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

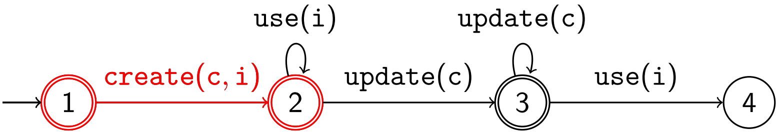
$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1) use(i1) update(A) use(i1)
$[c \mapsto A, i \mapsto i2] \mapsto$ create(A, i2) use(i2) update(A)



UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

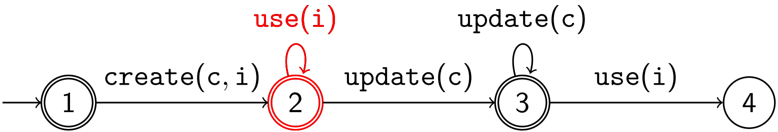
$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1) use(i1) update(A) use(i1)
$[c \mapsto A, i \mapsto i2] \mapsto$ create(A, i2) use(i2) update(A)



UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

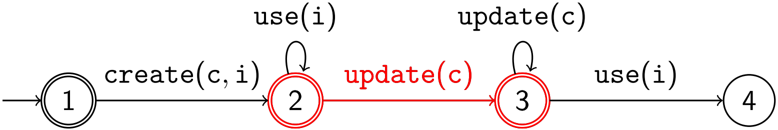
$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1) use(i1) update(A) use(i1)
$[c \mapsto A, i \mapsto i2] \mapsto$ create(A, i2) use(i2) update(A)



UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1) use(i1) update(A) use(i1)
$[c \mapsto A, i \mapsto i2] \mapsto$ create(A, i2) use(i2) update(A)

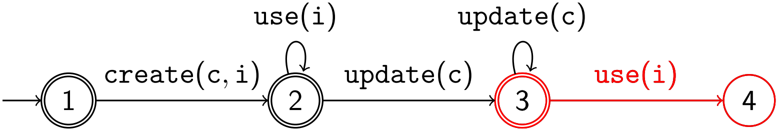


UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1) use(i1) update(A) use(i1) X

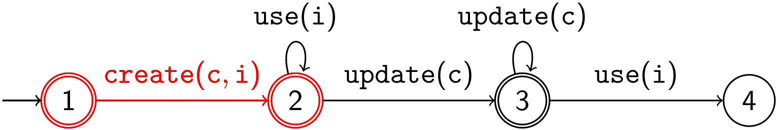
$[c \mapsto A, i \mapsto i2] \mapsto$ create(A, i2) use(i2) update(A)



UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1) use(i1) update(A) use(i1) X
$[c \mapsto A, i \mapsto i2] \mapsto$ create(A, i2) use(i2) update(A)

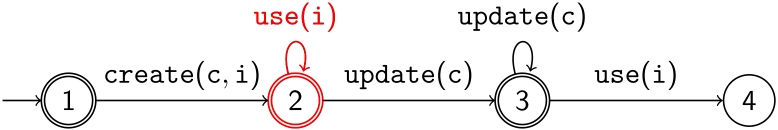


UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1) use(i1) update(A) use(i1) X

$[c \mapsto A, i \mapsto i2] \mapsto$ create(A, i2) use(i2) update(A)

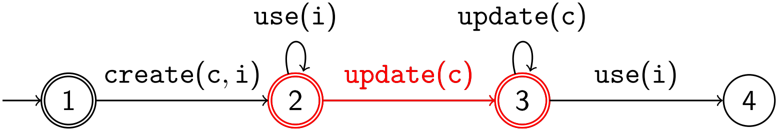


UnsafeIter Example

create(A, i1) use(i1) create(A, i2) use(i2) update(A) use(i1)

$[c \mapsto A, i \mapsto i1] \mapsto$ create(A, i1) use(i1) update(A) use(i1) X

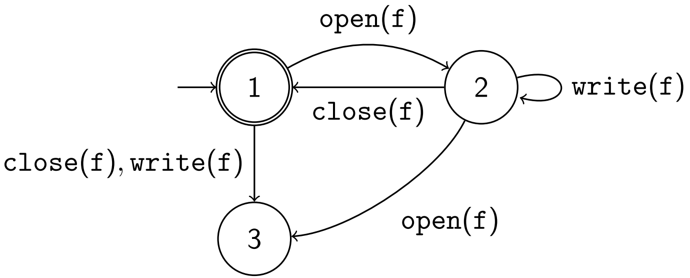
$[c \mapsto A, i \mapsto i2] \mapsto$ create(A, i2) use(i2) update(A) ✓



OpenClose Example

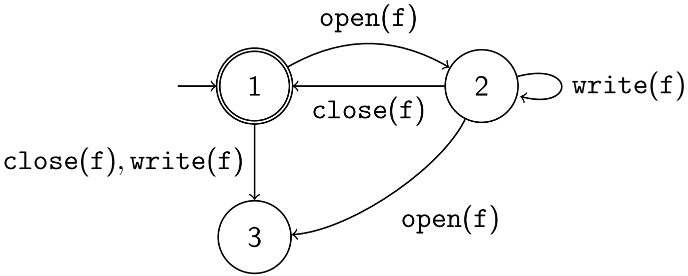
OpenClose

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.



OpenClose Example

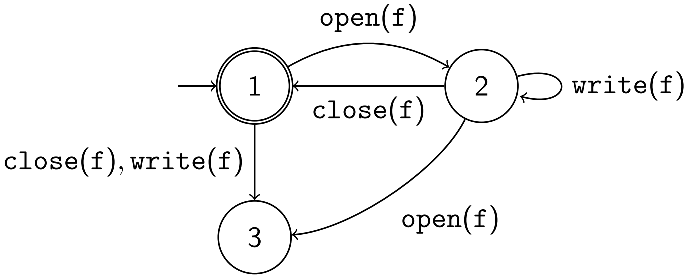
`open(A) open(B) write(A) write(B) close(B)`



OpenClose Example

open(A) open(B) write(A) write(B) close(B)

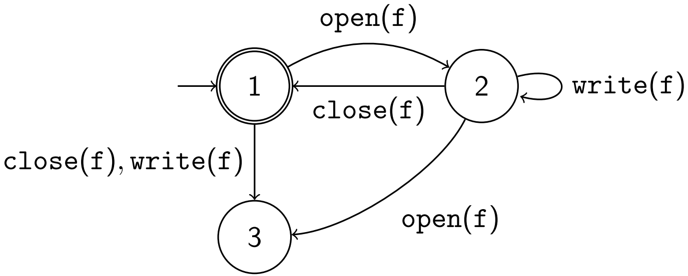
$[f \mapsto A] \mapsto$
$[f \mapsto B] \mapsto$



OpenClose Example

open(A) open(B) write(A) write(B) close(B)

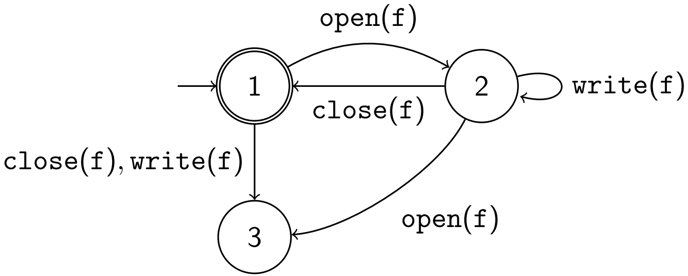
$[f \mapsto A] \mapsto$ open(A)
$[f \mapsto B] \mapsto$



OpenClose Example

open(A) open(B) write(A) write(B) close(B)

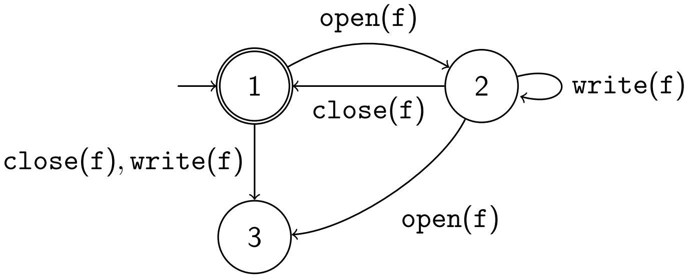
$[f \mapsto A] \mapsto$ open(A)
$[f \mapsto B] \mapsto$ open(B)



OpenClose Example

open(A) open(B) write(A) write(B) close(B)

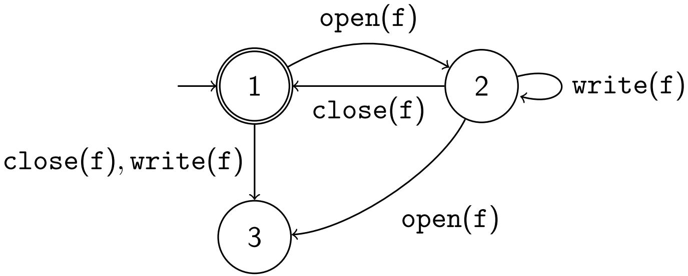
$[f \mapsto A] \mapsto$ open(A) write(A)
$[f \mapsto B] \mapsto$ open(B)



OpenClose Example

open(A) open(B) write(A) write(B) close(B)

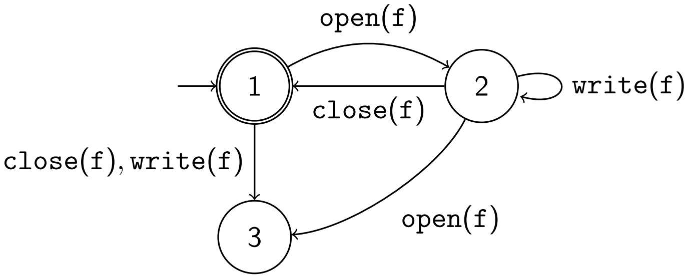
$[f \mapsto A] \mapsto$ open(A) write(A)
$[f \mapsto B] \mapsto$ open(B) write(B)



OpenClose Example

open(A) open(B) write(A) write(B) close(B)

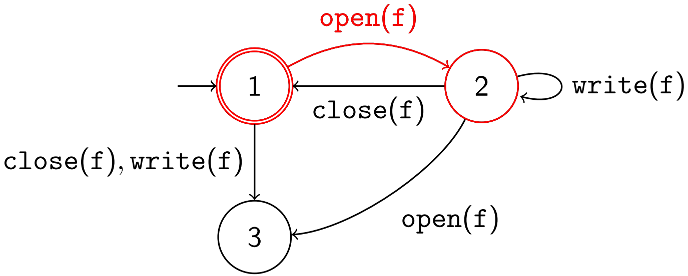
$[f \mapsto A] \mapsto$ open(A) write(A)
$[f \mapsto B] \mapsto$ open(B) write(B) close(B)



OpenClose Example

open(A) open(B) write(A) write(B) close(B)

$[f \mapsto A] \mapsto$ open(A) write(A)
$[f \mapsto B] \mapsto$ open(B) write(B) close(B)

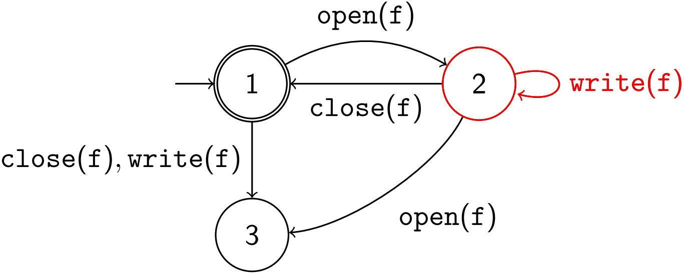


OpenClose Example

`open(A) open(B) write(A) write(B) close(B)`

$[f \mapsto A] \mapsto$ `open(A)` `write(A)` X

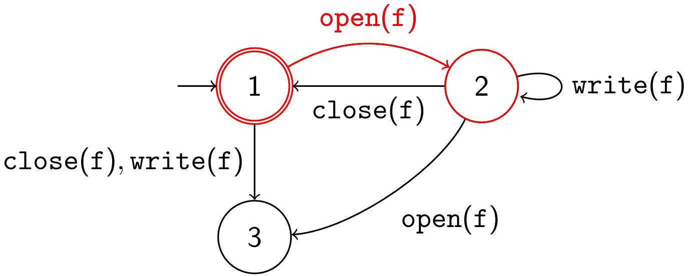
$[f \mapsto B] \mapsto$ `open(B) write(B) close(B)`



OpenClose Example

open(A) open(B) write(A) write(B) close(B)

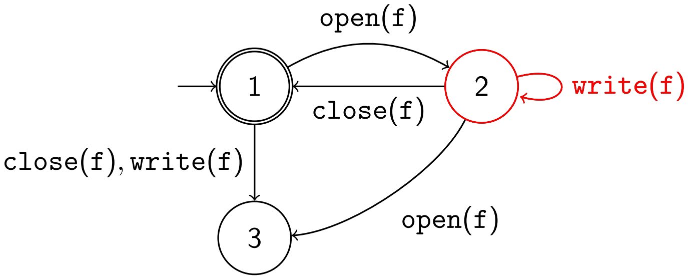
$[f \mapsto A] \mapsto$ open(A) write(A) X
$[f \mapsto B] \mapsto$ open(B) write(B) close(B)



OpenClose Example

open(A) open(B) write(A) write(B) close(B)

$[f \mapsto A] \mapsto$ open(A) write(A) X
$[f \mapsto B] \mapsto$ open(B) write(B) close(B)

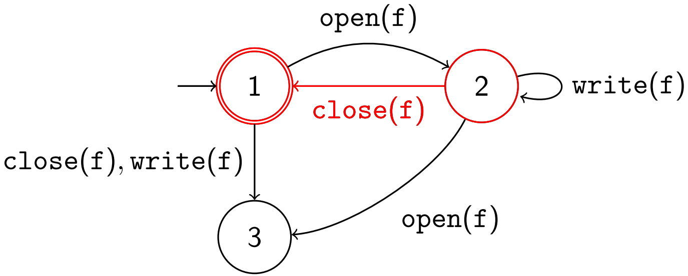


OpenClose Example

`open(A) open(B) write(A) write(B) close(B)`

$[f \mapsto A] \mapsto$ open(A) write(A) ✗

$[f \mapsto B] \mapsto$ open(B) write(B) close(B) ✓



Overview

1. Parametric Trace Slicing
2. Online Monitoring and Garbage
3. Static Analysis
4. What's Next?

The (basic) online monitoring algorithm

Not particularly important - but notice it depends on the size of Lookup, which is dependent on the number of objects being monitored.

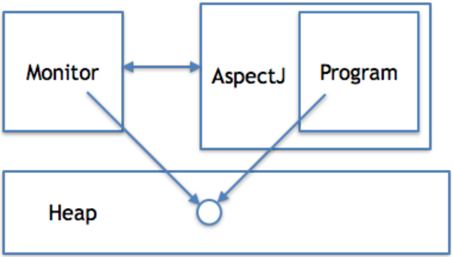
Let Lookup be a map from valuations to states initial mapping the empty valuation to the initial state
for event $e(\theta) \in \tau$ **do**
    for θ' in dom(Lookup) from biggest to smallest **do**
        if θ is consistent with θ' **then**
            if $\theta' \sqsubseteq \theta$ **then**
                Update Lookup(θ') using e
            else if $\theta \sqcup \theta'$ is not in dom(Lookup) **then**
                Add $\theta \sqcup \theta'$ to Lookup using Lookup(θ') updated using e
if an entry in Lookup is in a non-accepting state **then** Fail
else Accept

Typical Monitoring Setup

Events are generated by `AspectJ` and references to monitored objects are passed directly to the monitor

The monitor stores bindings of these objects associated with the current state of the associated automaton and searches these for each new event

So the monitor holds direct references into the memory of the monitored program



Garbage-Related Issues

Monitoring Overhead

- Overhead is dependent on number of monitored objects
- There are optimisations that reduce the dependency but it still exists
- Keeping objects that no longer contribute is inefficient

Memory leaks

- Keeping objects alive after they should die is a memory leak and can significantly change the behaviour of the monitored program

Anticipation

- If we remove an object we need to ensure that no associated slices are in a non-accepting state where acceptance is now unreachable
- Conversely, we have the chance of detecting such cases before the end of the program

Weak Reference Solution

This is the typical approach (taken by `tracematches`, JavaMOP, RuleR, optionally in MarQ, and other tools as well)

Wrap every monitored object in a `java.lang.ref.WeakReference`

In some cases can use implicitly collected objects such as `java.util.WeakHashMap` (or more likely custom-variants)

But in other cases, explicit clearing of such objects is required

In either case it is sometimes necessary to detect when an object becomes garbage in case further action is required (e.g. if file A became garbage in the OpenClose example).

Explicit Garbage Event Solution

Optional in MarQ

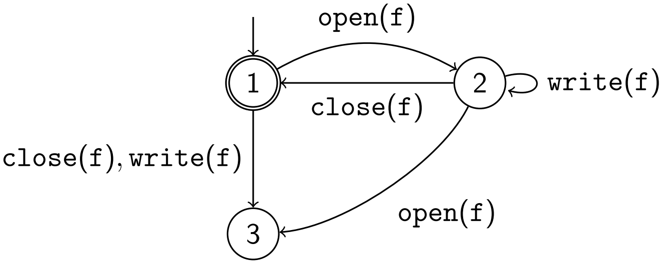
Idea:

- Separate identification of garbage from how it is handled in the monitor
- Implicitly extend QEA with so-called garbage events
- Generate garbage events whenever garbage is observed
- To generate garbage events, create a special object that is only referenced by the monitored object via a collection such that its collection triggers an event
  - We can think of this as a monitor that only detects garbage and whose verdicts are those objects that become garbage

Explicitly Adding Garbage Events

A state is a failure state if no accepting state can be reached. A state is a success state if no non-accepting state can be reached.

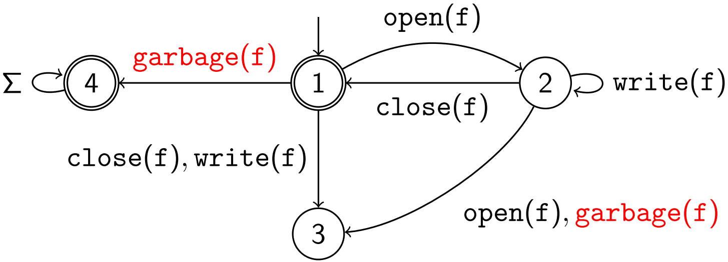
Add a garbage event to each state to either a failure or success state



Explicitly Adding Garbage Events

A state is a failure state if no accepting state can be reached. A state is a success state if no non-accepting state can be reached.

Add a garbage event to each state to either a failure or success state



Overview

1. Parametric Trace Slicing
2. Online Monitoring and Garbage
3. Static Analysis
4. What's Next?

Where Static Analysis fits in

We assume that events relate to program points, usually method calls (e.g. via `AspectJ`)

We will

1. Consider ways to statically determine pairs of program points A and B where objects created at point A will become unreachable at point B

2. Consider various ways in which this information can improve runtime monitoring based on parametric trace slicing

Small Example Program

```
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());
  }
  records.removeAll();
}
```

A points where an object is introduced

- `new File(fileName)`
- `records.iterator()` (factory method)

B points where an object becomes unreachable

- End of loop e.g. after last usage

Small Example Program

```
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());
  }
  records.removeAll();
}
```

- statically satisfies HasNext as `iterator` is local assuming we identify `iterator()` as a factory method
- statically satisfies UnsafeIter for this iterator but need to track collection as it escapes
- statically violates OpenClose as the local `file` is not closed

Escape Analysis

Determines if an object escapes a method

Uses pointer-analysis to track abstract objects

Typically flow-insensitive and intraprocedural

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

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

Objects only accessed, so `file` and `iterator` do not escape.

Requires us to identify `iterator` as a factory method

Free-me Analysis

Works on the call-flow graph of a program. Designed for explicit freeing.

Flow insensitive pointer analysis to identify abstract objects

- Start with set of assignments
- Propagate via assignments, accesses etc
- Represent globally reachable objects as one

Method summaries

- Summarise a method by how it treats its input variables
- An input variable is either returned, becomes globally reachable, or becomes reachable from another input parameter
- Can also identify pure and factory methods

Liveness analysis

- Backwards flow-sensitive analysis to detect reachability

Statically Generating Garbage Events

Once we have points A and B we can insert explicit garbage events at B points. Unlike free-me analysis, we can organise things so that it does not matter if we create multiple garbage events for the same object.

This allows

- Earlier generation of garbage events
- Earlier anticipation of failure

However, this is limited to shortly lived objects (i.e. that become locally unreachable) and such objects are often garbage collected reasonably quickly.

In the extreme case, we could use this information to inline monitoring and make it stack-based. However, in such cases static techniques would hopefully be able to statically check the property.

Supporting Offline Monitoring

Where else can this idea help?

In Offline monitoring it is necessary to record the identity of objects. Typically this is done using `IdentityHashCode` but this is not unique across garbage collections.

Idea: record garbage events to allow to replay garbage collection offline.

This now becomes a point of correctness rather than efficiency

Minimally Monitoring Abstract Objects

If an object O is created in method M and O does not escape M then we can enumerate the N paths O can take through M and once we have observed all N paths we can stop monitoring M.

Minimally Monitoring Abstract Objects

If an object O is created in method M and O does not escape M then we can enumerate the N paths O can take through M and once we have observed all N paths we can stop monitoring M.

The requirement for O to escape M can be relaxed such that we stop monitoring an object if it takes a path that has already been monitored, allowing for some paths to always require monitoring e.g. if O escapes.

Minimally Monitoring Abstract Objects

If an object O is created in method M and O does not escape M then we can enumerate the N paths O can take through M and once we have observed all N paths we can stop monitoring M.

The requirement for O to escape M can be relaxed such that we stop monitoring an object if it takes a path that has already been monitored, allowing for some paths to always require monitoring e.g. if O escapes.

One can also restrict this to path prefixes

Minimally Monitoring Abstract Objects

If an object O is created in method M and O does not escape M then we can enumerate the N paths O can take through M and once we have observed all N paths we can stop monitoring M.

The requirement for O to escape M can be relaxed such that we stop monitoring an object if it takes a path that has already been monitored, allowing for some paths to always require monitoring e.g. if O escapes.

One can also restrict this to path prefixes

This is similar to earlier work that attempted to detect loops where only a constant number of iterations of that loop required monitoring.

Minimally Monitoring Abstract Objects

If an object O is created in method M and O does not escape M then we can enumerate the N paths O can take through M and once we have observed all N paths we can stop monitoring M.

The requirement for O to escape M can be relaxed such that we stop monitoring an object if it takes a path that has already been monitored, allowing for some paths to always require monitoring e.g. if O escapes.

One can also restrict this to path prefixes

This is similar to earlier work that attempted to detect loops where only a constant number of iterations of that loop required monitoring.

This extends the idea of explicitly adding garbage events to the idea of statically noticing redundant objects i.e. those whose behaviour has been necessarily monitored previously.

Overview

1 Parametric Trace Slicing

2 Online Monitoring and Garbage

3 Static Analysis

4 What's Next?

Implement it

Plan to

- implement ideas in an analysis agnostic way i.e. using a set of pairs of program points
- make use of existing implementations for static analysis to suggest such pairs
- integrate into the MarQ monitoring tool

Missing QEA features

- Free variables: reachability can be over-aproximated in analysis
- Existential quantification: unclear if anything can be done

Risks, Limitations and Conclusions

Risks and Limitations

- As mentioned, mostly applies to short-lived objects that are garbage collected quickly anyway as very difficult to lift to an inter-procedural analysis
- However, in most cases an under-approximation of unreachable objects can be useful
- Cases where it can be applied might also be able to be fully statically verified using typestate analysis

Conclusions

- Need to try it out and see