Considering Typestate Verification for Quantified Event Automata

Giles Reger

University of Manchester, Manchester, UK

ISoLa 2016 – Corfu, October 13, 2016

In this Talk

I will talk about

- A brief overview of Quantified Event Automata (QEA)
- A brief overview of typestate analysis
- What issues do we face when extending current typestate analyses to QEA?
- What might the solutions be?

Context

- I am one of the developers of QEA and the MarQ tool
- I have wanted to explore static analysis for a while
- Klaus sent me an email saying, come to Corfu it'll be sunny
- So I started thinking about the problem and wrote this paper
- I haven't started the implementation yet... so if I am wrong I haven't wasted my time yet!

Parametric Trace Slicing

Idea

- Project/Slice a trace based on the values of some parameters
- For example, given parameters p_1 and p_2 with domains P_1 and P_2 consider slices for $P_1 \times P_2$ where $e(v) \in (v_1, v_2)$-slice if $v \in \{v_1, v_2\}$
- Check the correctness of each slice independently

Key Previous Work

- JavaMOP and tracematches
- After projection, primarily dealts with finite-state properties
- Tightly linked with AspectJ for defining events
- Based on notions of matching not acceptance/violation

Definition of QEA

QEA

A QEA is a pair $\langle \Lambda(X), \mathcal{E}(X \cup Y) \rangle$ where $\Lambda(X \cup Y)$ is a finite list of quantifications over X and $\mathcal{E}(Y)$ is an event automaton (EA) over $X \cup Y$. An EA is an automaton $\langle Q, q_0, \Sigma(X \cup Y), \delta, F \rangle$ where $\delta \subset (Q \times \Sigma(X \cup Y) \times \mathit{Guard} \times \mathit{Assignment} \times Q)$.

EA Acceptance

A configuration $\langle q, \theta \rangle$ is a pair of a state and binding over Y. We define $\langle q, \theta \rangle \xrightarrow{e(\overline{v})} \langle q', \theta' \rangle$ by lifting transitions in δ as one would expect. Then acceptance is reachability of a state in F.

QEA Acceptance

Where domain $\mathit{Dom}(\tau)$ is extracted from the trace:

$$
\begin{array}{lll}
\tau \models_\theta (\forall x)\Lambda' & \text{iff} & \text{for all } d \text{ in } \mathit{Dom}(\tau)(x) : \tau \models_{\theta\dagger[x \mapsto d]} \Lambda' \\
\tau \models_\theta (\exists x)\Lambda' & \text{iff} & \text{for some } d \text{ in } \mathit{Dom}(\tau)(x) : \tau \models_{\theta\dagger[x \mapsto d]} \Lambda' \\
\tau \models_\theta \epsilon & \text{iff} & \tau \downarrow_{\mathrm{E}(\theta)} \in \mathcal{L}(\mathrm{E}(\theta))
\end{array}
$$

Key Differences to Previous work

Local Variables

- Instead of projecting to a propositional trace we keep data values
- Definition of per-slice behaviour has access to these data values
- Achieved by local/free variables and notions of guards/assignments
- **No longer finite-state**
- (JavaMOP had an ad-hoc notion of adding variables to the Aspect)

Existential Quantification

- Previous implicit notion was 'for each slice the property should hold'
- Now we have a complex quantification structure
- Note that a property $\exists x : \varphi[x]$ does not have bad prefixes

Relation to AspectJ

Pointcuts to Events

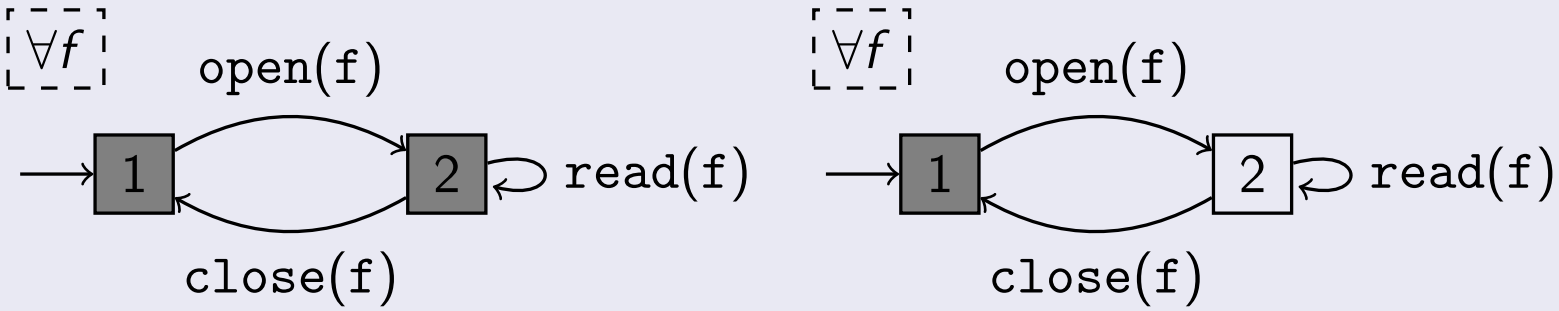
- As in JavaMOP, one can relate pointcuts to events
- However, as a difference, one can relate one pointcut to multiple events (tracematches had this)
- This means that the relationship is not implicit

Simplification

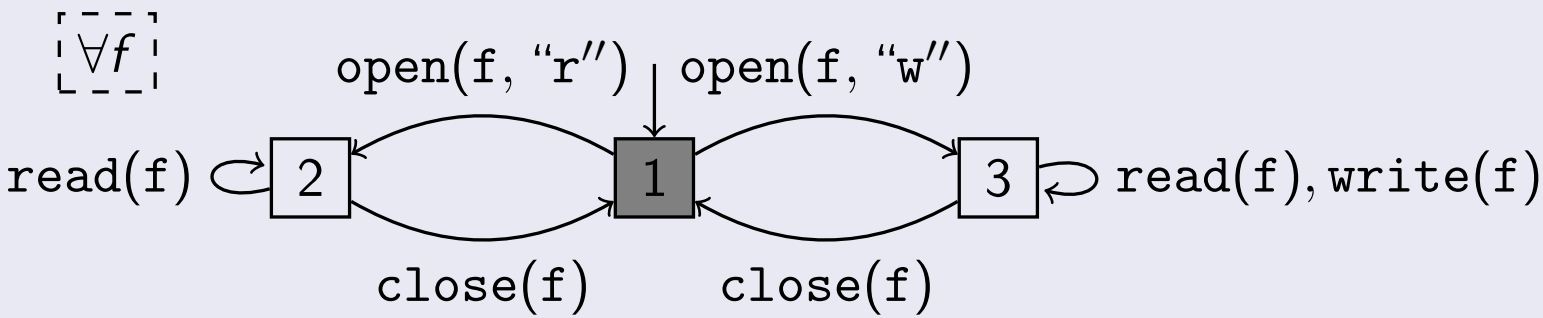
- However, in this talk I will assume they are one-to-one
- This is the common case and the alternative shouldn't complicate things too much

Example 1: Opening and Closing Files

FileSafety and FileGeneral

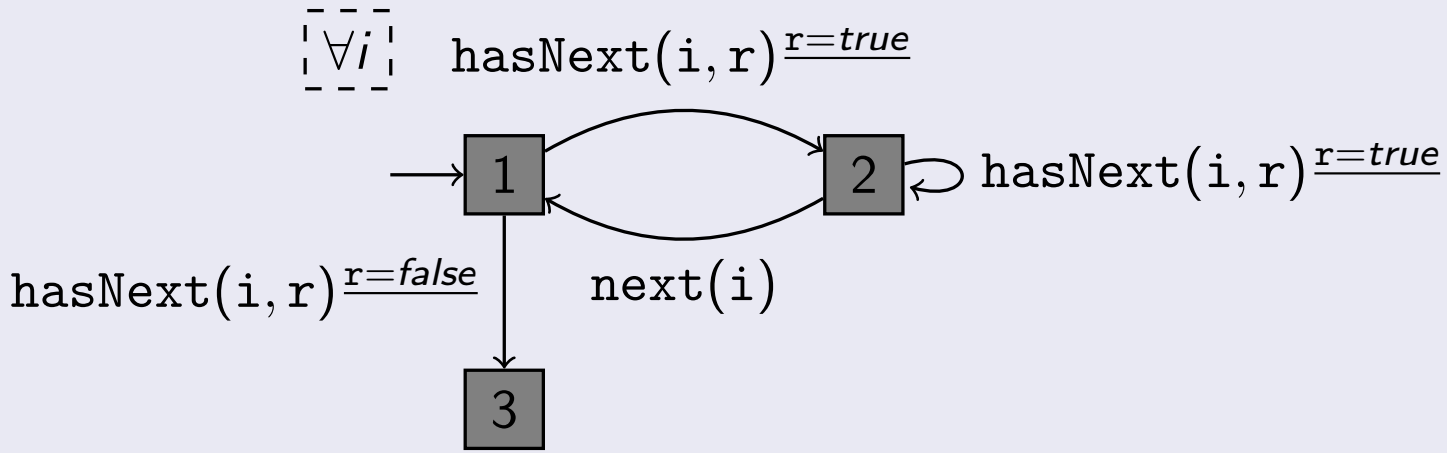


FileModal

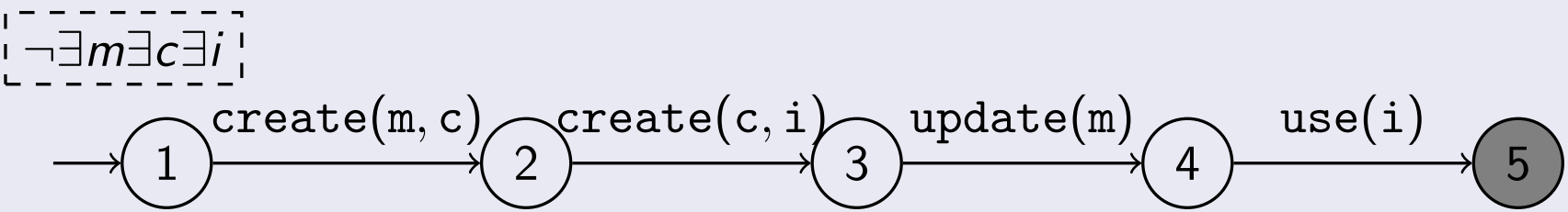


Example 2: Iterators

HasNext

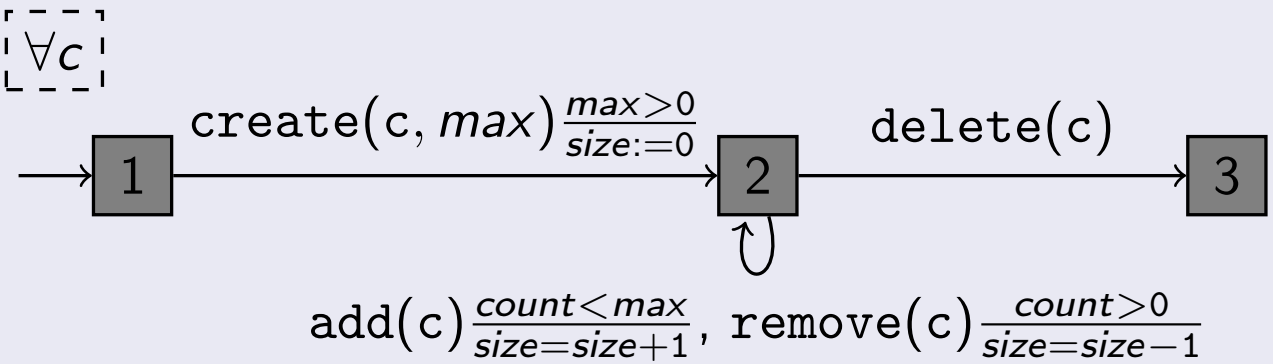


UnsafeMapIterator



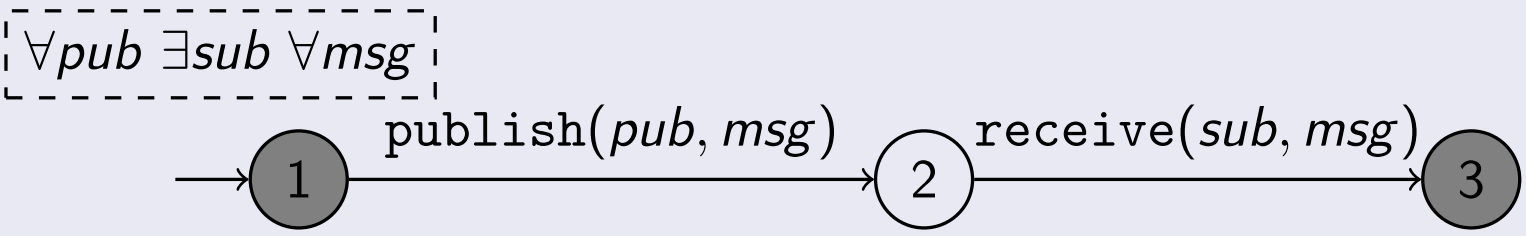
Example 3: Collections

BoundedCollection



Example 4: PublisherSubscriber

PublisherSingleSubscriber



What is TypeState Analysis?

Strom and Yemini (1986)

- The idea: types should have typestates where certain actions are only allowed when the variable of that type is in a certain typestate
- Lots of work since then
- The idea should be familiar to RV

Clara

- I focus primarily on the Clara approach as it is the only relevant topic here and time is limited
- Developed by Bodden et al. as a method for partially evaluating monitors ahead of time
- By removing joinpoints from AspectJ instrumented systems that could be statically shown to not contribute to matching

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

A

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  System.out.println("There are "+map.keySet().size()+
              " unique keys");
}
```

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

A No. There are no iterators created.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  System.out.println("There are "+map.keySet().size()+
              " unique keys");
}
```

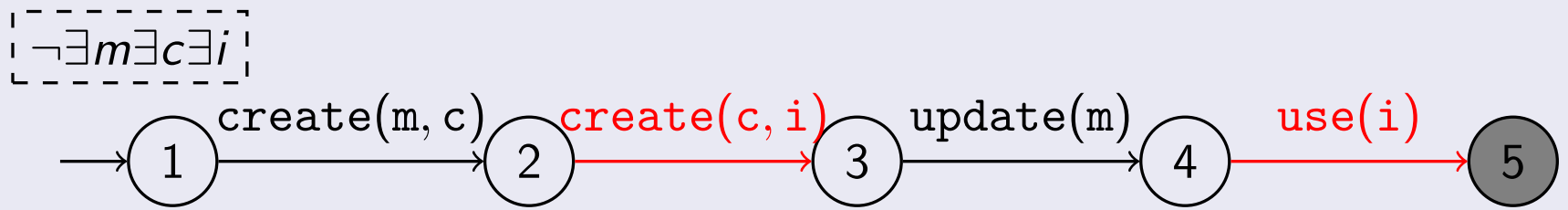
Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

S Syntactic Quick Check: use missing symbols to reduce property, potentially show there are no possible matches/violations

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  System.out.println("There are "+map.keySet().size()+
                     " unique keys");
}
```

UnsafeMapIterator



Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

A

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = Arrays.asList(args).iterator();
  while(iter.hasNext()){
      String arg = iter.next();
      if(map.keySet().contains(Integer.parseInt(arg)) &&
         map.containsValue(arg)){
          System.out.println(arg+" is a key and value");
      }
  }
}
```

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

A No. No one slice contains all necessary events.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = Arrays.asList(args).iterator();
  while(iter.hasNext()){
      String arg = iter.next();
      if(map.keySet().contains(Integer.parseInt(arg)) &&
         map.containsValue(arg)){
          System.out.println(arg+" is a key and value");
      }
  }
}
```

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

S Orphan-shadows Analysis: per-slice Quick Check. Slices defined over heap model abstraction (allocation points give objects)

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = Arrays.asList(args).iterator();
  while(iter.hasNext()){
      String arg = iter.next();
      if(map.keySet().contains(Integer.parseInt(arg)) &&
         map.containsValue(arg)){
          System.out.println(arg+" is a key and value");
      }
  }
}
```

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

A

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.keySet().iterator();
  while(iter.hasNext()){
    Integer key = iter.next();
    System.out.println(key+" \t:\t"+map.get(key));
  }
}
```

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

A No. There are no updates after iteration.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.keySet().iterator();
  while(iter.hasNext()){
    Integer key = iter.next();
    System.out.println(key+" \t:\t"+map.get(key));
  }
}
```

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

S Nop-shadows Analysis. Flow-sensitive! Per shadow (joinpoint) detect reached/ sets of reachable states. Nop-shadow if it takes all reached states to states in the same reachable set.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.keySet().iterator();
  while(iter.hasNext()){
    Integer key = iter.next();
    System.out.println(key+" \t:\t"+map.get(key));
  }
}
```

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

A

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.valueSet().iterator();
  while(iter.hasNext()){
    Integer key = iter.next();
    if(map.keySet().contains(Integer.parseInt(arg))){
      map.remove(key);
    }
  }
}
```

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

A Maybe. We cannot tell statically.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.valueSet().iterator();
  while(iter.hasNext()){
    Integer key = iter.next();
    if(map.keySet().contains(Integer.parseInt(arg))){
      map.remove(key);
    }
  }
}
```

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

S Nop-shadows Analysis cannot remove any shadows. Behaviour dependent on input.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.valueSet().iterator();
  while(iter.hasNext()){
    Integer key = iter.next();
    if(map.keySet().contains(Integer.parseInt(arg))){
      map.remove(key);
    }
  }
}
```

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

A

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.valueSet().iterator();
  map.insert(0,"empty");
  while(iter.hasNext()){
    Integer key = iter.next();
    if(map.keySet().contains(Integer.parseInt(arg))){
      map.remove(key);
    }
  }
}
```

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

A Yes. This insertion must violate the property.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.valueSet().iterator();
  map.insert(0,"empty");
  while(iter.hasNext()){
    Integer key = iter.next();
    if(map.keySet().contains(Integer.parseInt(arg))){
      map.remove(key);
    }
  }
}
```

Demonstrating Clara Static Analyses

Q Does the following violate the UnsafeMapIterator property?

S Certain-match Analysis. Use information from Nop-shadow analysis to find certainly reached bad states.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.valueSet().iterator();
  map.insert(0,"empty");
  while(iter.hasNext()){
    Integer key = iter.next();
    if(map.keySet().contains(Integer.parseInt(arg))){
      map.remove(key);
    }
  }
}
```

Static Analyses Summary

Quick Check

- Find the set of symbols not occurring in the method
- Reduce the property

Orphan Shadow Analysis

- When taking a per-slice view some symbols become orphans i.e. they do not belong to a feasible path to a match

Nop-Shadow Analysis

- Flow-sensitive analysis to find shadows that make no difference to paths to matching states

Certain-Match Analysis

- Use information from Nop-Shadow analysis to find certain-matches

What are the Challenges?

Non-Safety Properties

- Ugly states taken to final states via an endOfTrace event
- QEA may under approximate final states due to local variables
- Need to add a notion of endOfTrace to analysis

Local Variables/Non-Finite State

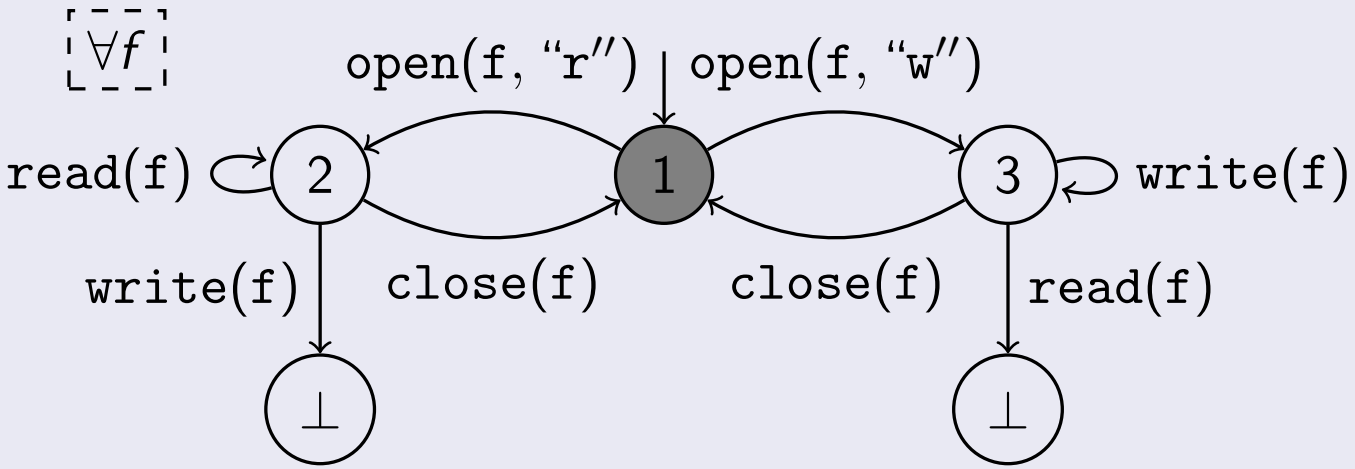
- Different levels of expressiveness e.g. local guards, empty assignments
- Local variables change notion of reachable state, can make nop-shadown analysis unsound
- Interferes with notion of slicing

Existential Quantification

- Current analyses only talk about reaching some match
- Existential quantification changes the whole game

Local Guards and QuickCheck

FileModal (Updated)

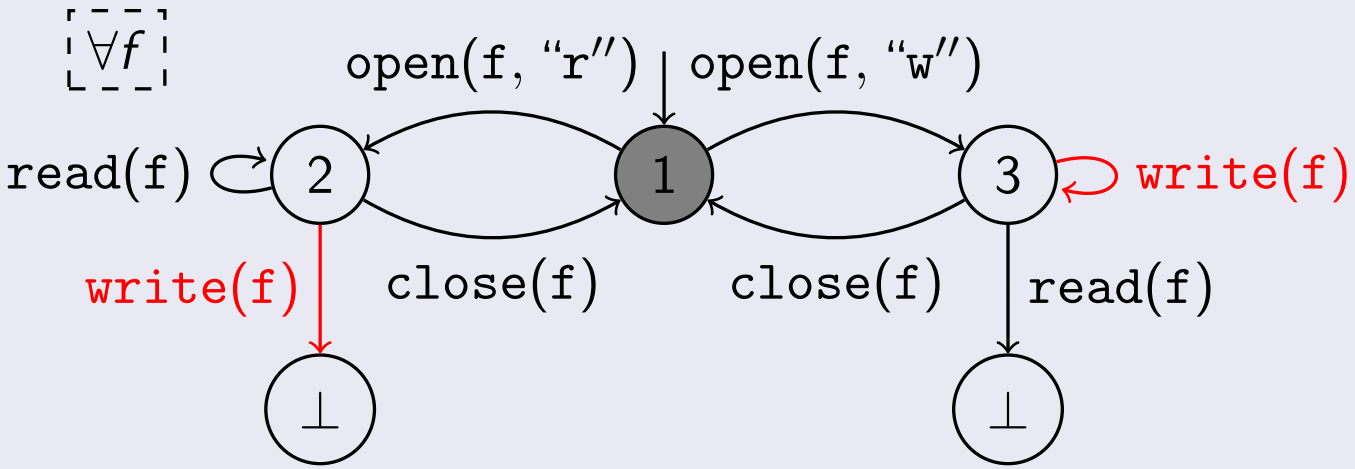


Example Code

```
processLines(String name){
  File f = new File(name,"r");
  String line;
  while(line = f.read()){ processLine(line); }
  f.close()
}
```

Local Guards and QuickCheck

FileModal (Updated)

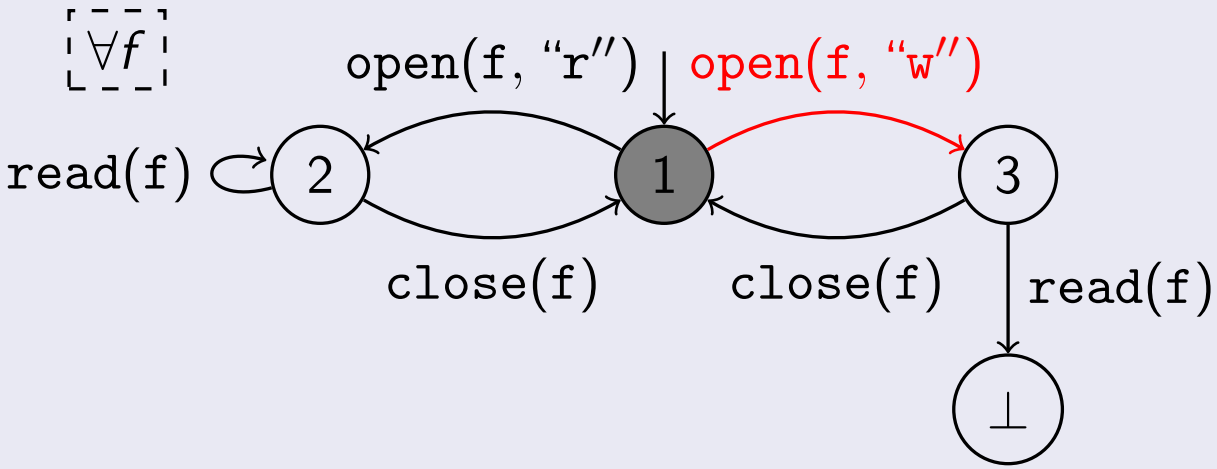


Example Code

```
processLines(String name){
  File f = new File(name,"r");
  String line;
  while(line = f.read()){ processLine(line); }
  f.close()
}
```

Local Guards and QuickCheck

FileModal (Updated)

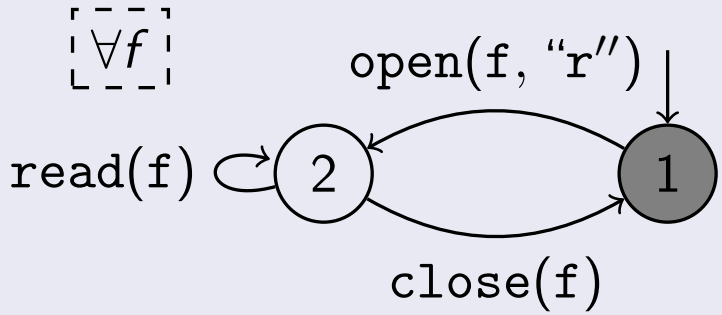


Example Code

```
processLines(String name){
  File f = new File(name,"r");
  String line;
  while(line = f.read()){ processLine(line); }
  f.close()
}
```

Local Guards and QuickCheck

FileModal (Updated)

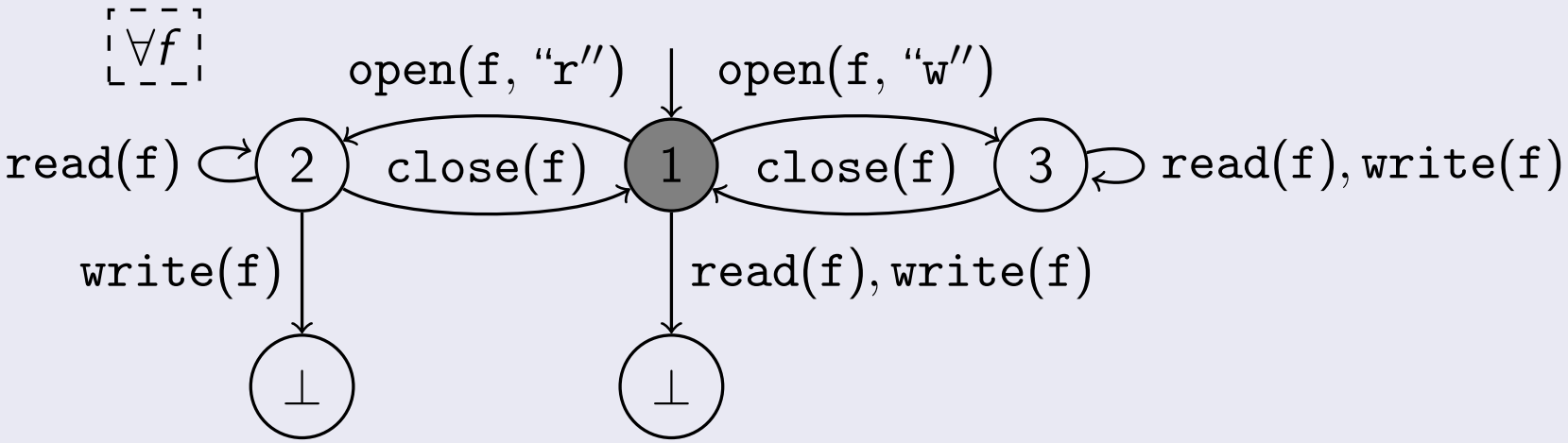


Example Code

```
processLines(String name){
  File f = new File(name,"r");
  String line;
  while(line = f.read()){ processLine(line); }
  f.close()
}
```

Local Guards and Nop-Shadow Analysis

FileModal (Updated)

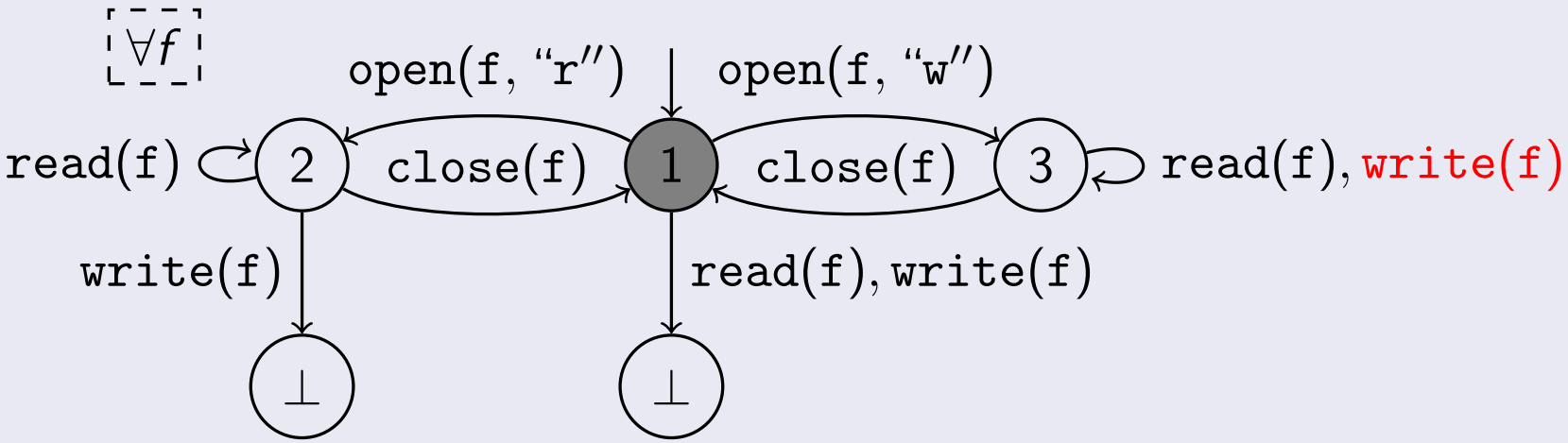


Example Code

```
writeLines(String name, List<String> toWrite){
    File f = new File(name,"w");
    for(String line: toWrite){
        f.write(line)
    }
    f.close()
 }
```

Local Guards and Nop-Shadow Analysis

FileModal (Updated)

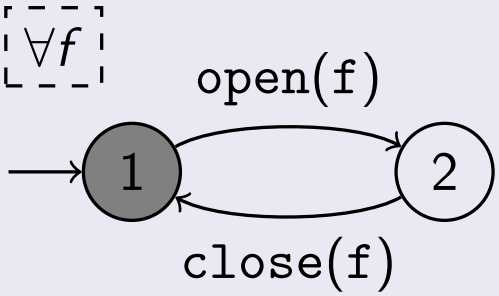


Example Code

```
writeLines(String name, List<String> toWrite){
    File f = new File(name,"w");
    for(String line: toWrite){
        f.write(line)
    }
    f.close()
 }
```

Adding End of Trace

FileGeneral (Updated)

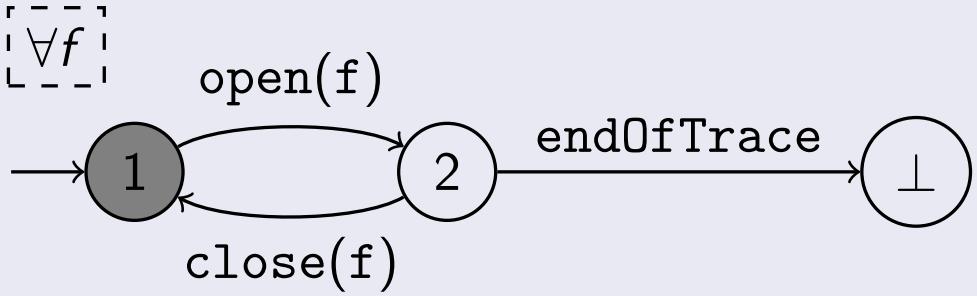


Example Code

```
public void writeSetToFile(Set<Integer> set, String name){
   File file = new File(name);
   file.open();
   for(Integer i : set){
     if(i.equals(0)){
        System.out.println(“Error”); System.exit(0);
     }
     file.write(i);
    }
    file.close();
 }
```

Adding End of Trace

FileGeneral (Updated)

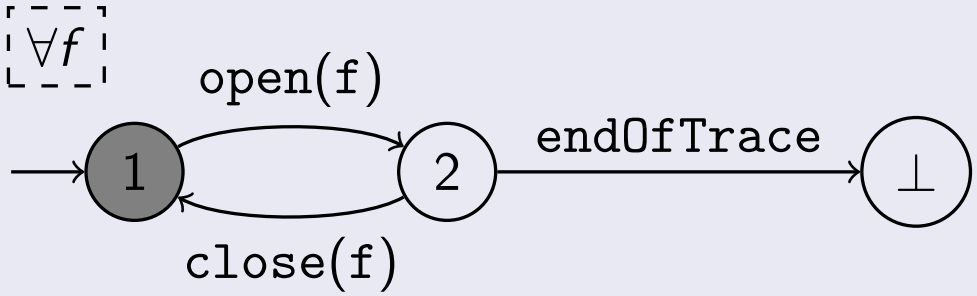


Example Code

```
public void writeSetToFile(Set<Integer> set, String name){
   File file = new File(name);
   file.open();
   for(Integer i : set){
     if(i.equals(0)){
        System.out.println(“Error”); System.exit(0);
     }
     file.write(i);
    }
    file.close();
 }
```

Adding End of Trace

FileGeneral (Updated)



Example Code

```
public void writeSetToFile(Set<Integer> set, String name){
   File file = new File(name);
   file.open();
   for(Integer i : set){
     if(i.equals(0)){
        System.out.println(“Error”); System.exit(0);
     }
     file.write(i);
    }
    file.close();
 }
```

Liveness Analysis

Example Code

```
void processMap(Map m){
    Collection c = map.keySet();
    Iterator i = c.iterator();
    while(i.hasNext()){
      Object o = i.next();
      if(check(o)){
        runUpdate(o,m);
      }
    }

}
```

Discussion

- `m` escapes the method, must assume it could be updated
- What else can we do here?
- `c` and `i` are only live during this method, they will eventually be garbage collected
- But we can detect this here and inform the monitor

Liveness Analysis

Example Code

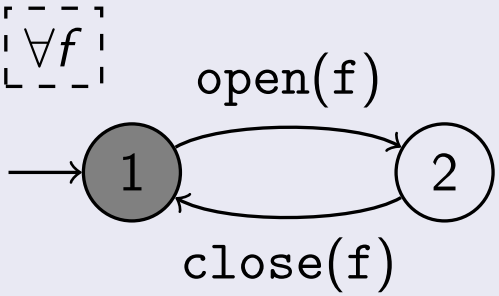
```
void processMap(Map m){
    Collection c = map.keySet();
    Iterator i = c.iterator();
    while(i.hasNext()){
      Object o = i.next();
      if(check(o)){
        runUpdate(o,m);
      }
    }
    monitor.garbage(c,i);
}
```

Discussion

- `m` escapes the method, must assume it could be updated
- What else can we do here?
- `c` and `i` are only live during this method, they will eventually be garbage collected
- But we can detect this here and inform the monitor

Liveness Analysis

FileGeneral (Updated)



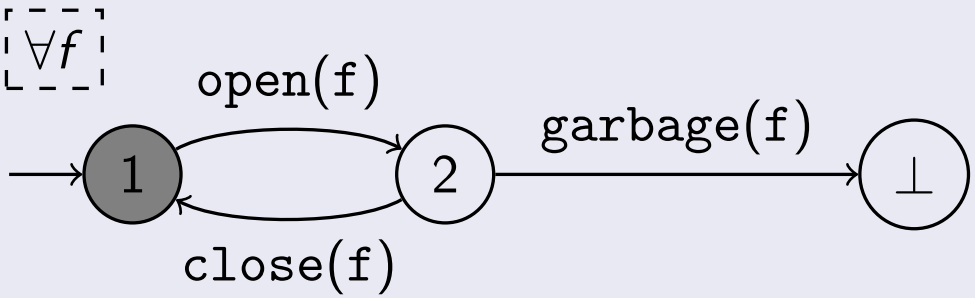
Example Code

```
public void writeSetToFile(Set<Integer> set, String name){
   File file = new File(name);
   file.open();
   for(Integer i : set){
     file.write(i);
    }

 }
```

Liveness Analysis

FileGeneral (Updated)



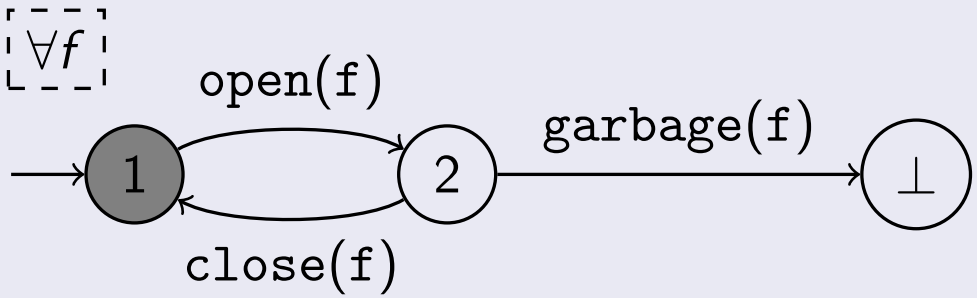
Example Code

```
public void writeSetToFile(Set<Integer> set, String name){
   File file = new File(name);
   file.open();
   for(Integer i : set){
     file.write(i);
    }

 }
```

Liveness Analysis

FileGeneral (Updated)

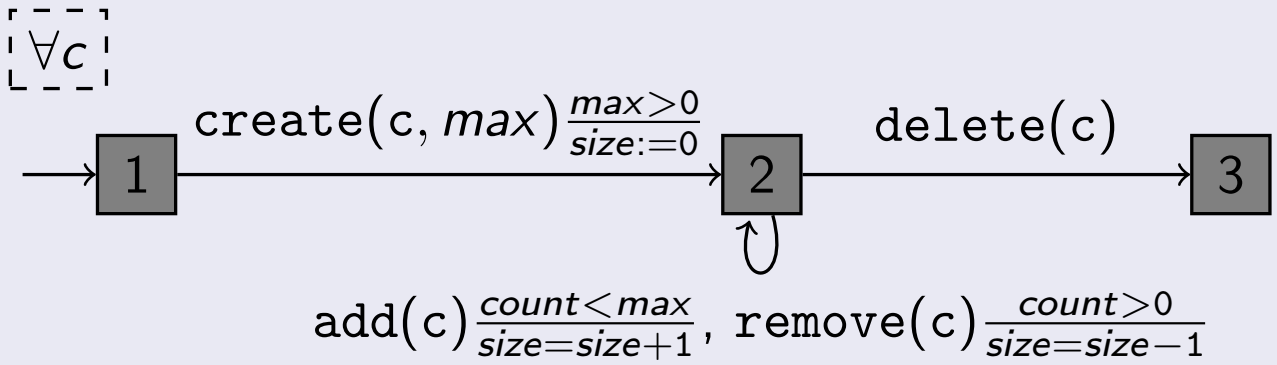


Example Code

```
public void writeSetToFile(Set<Integer> set, String name){
   File file = new File(name);
   file.open();
   for(Integer i : set){
     file.write(i);
    }
    garbage(f)
 }
```

But what about here?

BoundedCollection



Example Code

```
Collection fill(int value){
  Collection c = new Collection(value);
  for(int i=0;i<=value;i++){
    c.add(i);
  }
  return c;
}
```

Main Idea

- Push local variables into ghost code, Monitor the ghost code

Existential Quantification

Special Case

- Single quantification i.e. $\exists x : \varphi[x]$
- If we can find the single witness anywhere then we are done
- Search for $\top$ matches in the same way that we previously search for $\bot$

General Case

- Alternating quantification
- Maybe there is a special case for $\forall x \exists y$ if an instance of x has limited scope
- Perhaps some post-processing on potential matches
- But this comes at things from the wrong direction, partial evaluation is only partial

Summary

- QEA Extends the notion of parametric trace slicing with extra things
- Want to extend the Clara approach with those extra things
- Not got my hands dirty yet
- Lots of Challenges
- This additional idea of liveness analysis for detecting garbage early