Using patterns to infer first-order temporal specifications

Giles Reger

University of Manchester

March 25, 2013

Outline

The problem

system specifications are useful

The problem

formal system specifications are useful

The problem

formal system specifications are useful for testing, verification, maintenance, understanding,...

The problem

formal system specifications are useful for testing, verification, maintenance, understanding,...

but

are also difficult and costly to write

The problem

formal system specifications are useful for testing, verification, maintenance, understanding,...

but

are also difficult and costly to write

The problem

formal system specifications are useful for testing, verification, maintenance, understanding,...

but

are also difficult and costly to write

The problem

formal system specifications are useful for testing, verification, maintenance, understanding,...

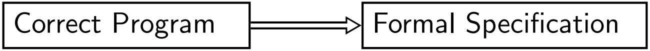
but

are also difficult and costly to write and are therefore often missing or incomplete.

The 'solution'

Infer specifications from 'correct' programs

i.e. extract them don't write them



The more specific problem

Given a set of dynamic traces, passively infer a temporal specification.

- **Dynamic** - The input will be recorded traces.

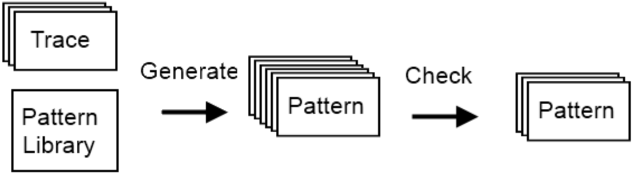
  Traces have the advantage (over source code) that they contain common behaviour.
- **Passive** - We cannot query or interact with the system.
- **Temporal** - We are only concerned with properties about the ordering of events. In this work a specification denotes a set of allowed traces of events.

Where I'm coming from

- Last year I gave a talk on a runtime verification technique for checking first-order temporal properties
- Today I'm using this technique for extracting specifications rather than testing them against a trace
- The natural setting for this is a pattern-based technique

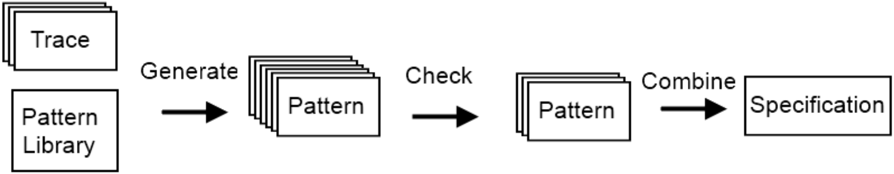
Where I'm coming from

- Last year I gave a talk on a runtime verification technique for checking first-order temporal properties
- Today I'm using this technique for extracting specifications rather than testing them against a trace
- The natural setting for this is a pattern-based technique



Where I'm coming from

- Last year I gave a talk on a runtime verification technique for checking first-order temporal properties
- Today I'm using this technique for extracting specifications rather than testing them against a trace
- The natural setting for this is a pattern-based technique



The generate-and-check approach in action

- We use a set of abstract patterns and an alphabet to generate a set of concrete patterns and then check these on the trace(s)
- For example, given these (abstract) patterns

$$(ab)^* \qquad (ab^+c)^*$$

  and this trace

  `open.use.close.open.use.use.close`

  we can identify these (concrete) patterns

$$(\texttt{open close})^*$$
$$(\texttt{open use}^+\ \texttt{close})^*$$

- This is (roughly) what has been done previously in this space

How to interpret data

- What do we do when events can carry data?

  `open(1).use(1).close(1).open(2).use(2).close(2)`

- This looks fine - ignore the data! (using it in the alphabet will quickly explode)

How to interpret data

- What do we do when events can carry data?

  `open(1).open(2).use(1).use(2).close(2).close(1)`

- But here we have two `open` events in a row - we won't detect our previous patterns

How to interpret data

- What do we do when events can carry data?

  `open(1).open(2).use(1).use(2).close(2).close(1)`

- But here we have two `open` events in a row - we won't detect our previous patterns
- There is a sense that the events for different data values can be *separated* i.e.

  file 1: `open(1).use(1).close(1)`
  file 2: `open(2).use(2).close(2)`

How to interpret data

- What do we do when events can carry data?

$$\texttt{open}(1).\texttt{open}(2).\texttt{use}(1).\texttt{use}(2).\texttt{close}(2).\texttt{close}(1)$$

- But here we have two open events in a row - we won't detect our previous patterns
- There is a sense that the events for different data values can be *separated* i.e.

$$\begin{array}{ll} \text{file 1:} & \texttt{open}(1).\texttt{use}(1).\texttt{close}(1) \\ \text{file 2:} & \texttt{open}(2).\texttt{use}(2).\texttt{close}(2) \end{array}$$

- We can also do this with multiple pieces of data, i.e.

$$\texttt{create}(L_1).\texttt{add}(L_1, O_1).\texttt{create}(L_2).\texttt{add}(L_2, O_1).\texttt{remove}(L_1, O_1)$$

- becomes

$$\begin{array}{ll} \text{list } L_1 \text{ with object } O_1\text{:} & \texttt{create}(L_1).\texttt{add}(L_1, O_1).\texttt{remove}(L_1, O_1) \\ \text{list } L_2 \text{ with object } O_1\text{:} & \texttt{create}(L_2).\texttt{add}(L_2, O_1) \end{array}$$

Quantified Event Automata (QEA)

- The runtime verification technique mentioned earlier
- Also, our target temporal specification language
- Has this notion of slicing up the trace built in
- Good for this generate-and-check approach as efficient checking algorithms have been developed

What is it?

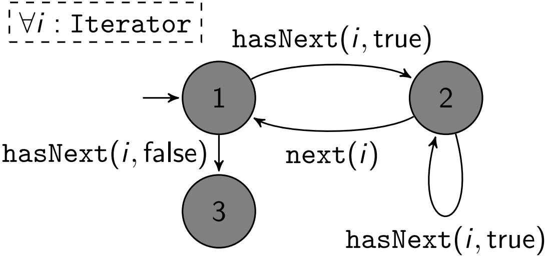
- A QEA defines a language of traces of events
- Events are defined as a name and a list of symbols i.e.

$\texttt{open}(f)$ $\texttt{login}(user, pwd)$ $\texttt{send}(msg, time, addr)$

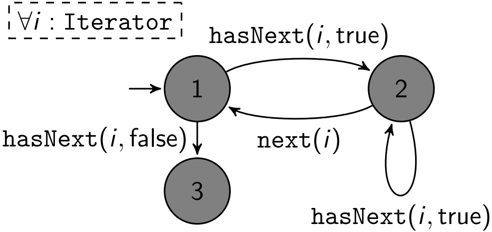
- A QEA consists of
  - A list of quantifications of variables X
  - A state machine over an alphabet of events using X

An example : HasNext

A call of `next` to an iterator is safe if it is preceded directly by a call of `hasNext` that returns true.



An example of (non)acceptance



Trace
hasNext(A,true)
hasNext(B,true)
next(A)
next(B)
hasNext(B,true)
next(A)

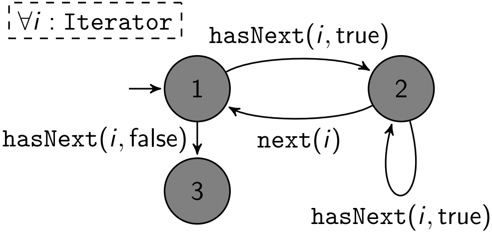
For $i = A$

$$\texttt{hasNext}(A, \mathit{true}).\texttt{next}(A).\texttt{next}(A)$$

For $i = B$

$$\texttt{hasNext}(B, \mathit{true}).\texttt{next}(B).\texttt{hasNext}(B, \mathit{true})$$

An example of (non)acceptance



Trace
hasNext(A,true)
hasNext(B,true)
next(A)
next(B)
hasNext(B,true)
next(A)

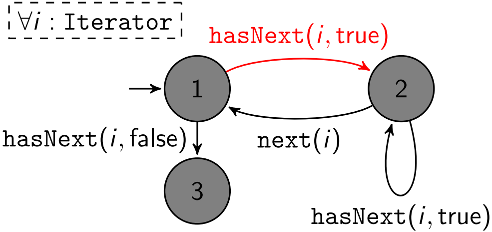
For $i = A$

$$\texttt{hasNext}(A, \mathit{true}).\texttt{next}(A).\texttt{next}(A)$$

For $i = B$

$$\texttt{hasNext}(B, \mathit{true}).\texttt{next}(B).\texttt{hasNext}(B, \mathit{true})$$

An example of (non)acceptance



Trace
hasNext(A,true)
hasNext(B,true)
next(A)
next(B)
hasNext(B,true)
next(A)

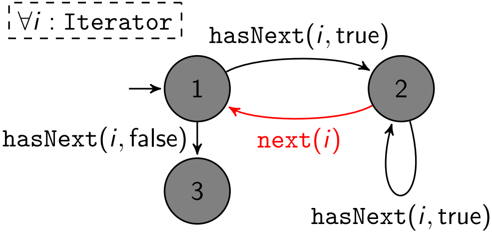
For $i = A$

$$\texttt{hasNext}(A, \mathit{true}).\texttt{next}(A).\texttt{next}(A)$$

For $i = B$

$$\texttt{hasNext}(B, \mathit{true}).\texttt{next}(B).\texttt{hasNext}(B, \mathit{true})$$

An example of (non)acceptance



Trace
hasNext(A,true)
hasNext(B,true)
next(A)
next(B)
hasNext(B,true)
next(A)

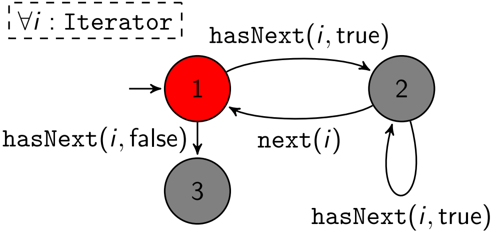
For $i = A$

$$\texttt{hasNext}(A, \mathit{true}).\texttt{next}(A).\texttt{next}(A)$$

For $i = B$

$$\texttt{hasNext}(B, \mathit{true}).\texttt{next}(B).\texttt{hasNext}(B, \mathit{true})$$

An example of (non)acceptance



Trace
hasNext(A,true)
hasNext(B,true)
next(A)
next(B)
hasNext(B,true)
next(A)

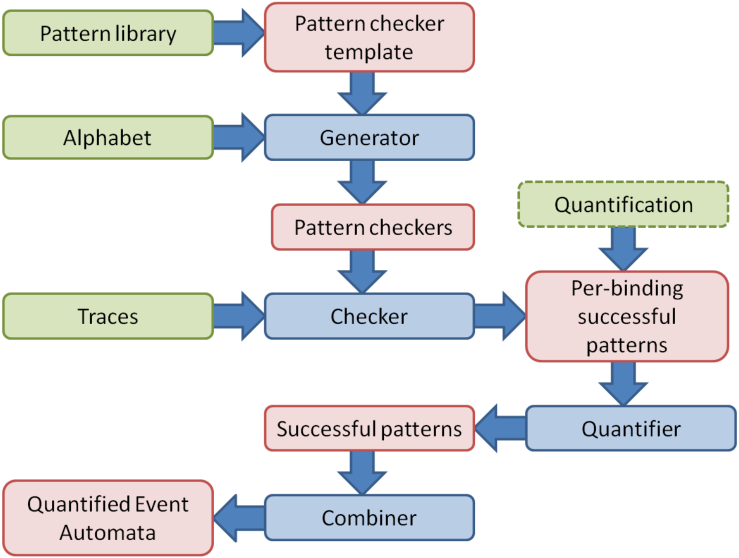
For $i = A$

$$\texttt{hasNext}(A, \mathit{true}).\texttt{next}(A).\texttt{next}(A)$$

For $i = B$

$$\texttt{hasNext}(B, \mathit{true}).\texttt{next}(B).\texttt{hasNext}(B, \mathit{true})$$

Overview of mining process



Defining what we mean by pattern

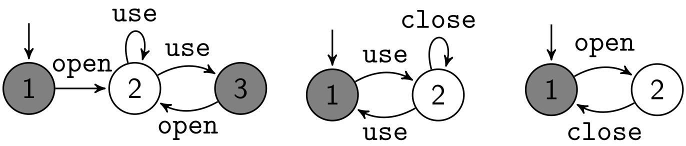
- Patterns need to be predicates on traces that can be combined
- It has been previously shown that using standard automata or regular expressions is inadequate for combination
- This inadequacy (shown next) leads to over-generalised combinations
- We therefore introduce a new kind of automata to use as a pattern (shown shortly)

The inadequacy

- Given the trace

  `open.use.use.close.open.use.close`

- We might mine these patterns

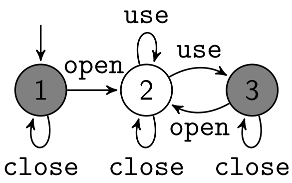


The inadequacy

- Given the trace

`open.use.use.close.open.use.close`

- We might mine these patterns



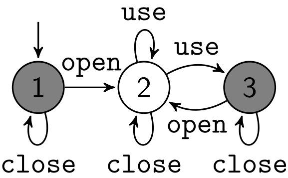
- Which uses the standard method of expanding alphabets

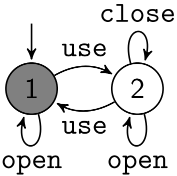
The inadequacy

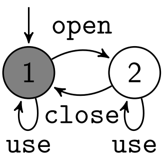
- Given the trace

`open.use.use.close.open.use.close`

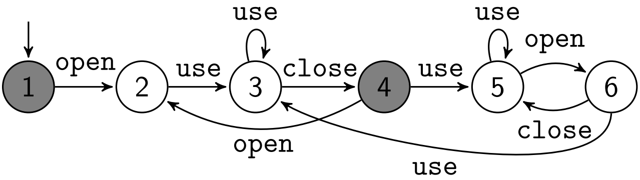
- We might mine these patterns







- Which uses the standard method of expanding alphabets to use standard automata intersection to give

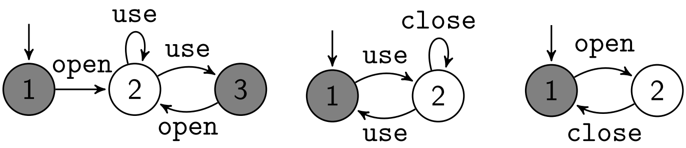


The inadequacy

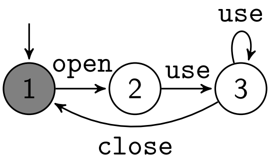
- Given the trace

`open.use.use.close.open.use.close`

- We might mine these patterns



- When we should hope for



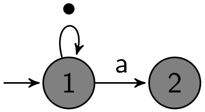
- The problem is that we have no information about where interleaving can happen

Patterns = open automata

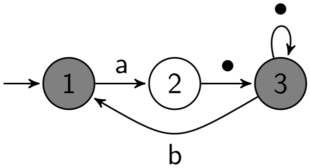
- For use as patterns we introduce *open automata*
- An open-automata is a state machine with a special • hole symbol that can label transitions
- The hole symbol matches any symbol not in the state machine's alphabet
- To avoid undesired interleaving we define intersection so that it expands alphabets only on holes

An example pattern library

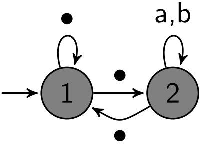
Let us assume our pattern library consists of these three patterns



Pattern p_1



Pattern p_2

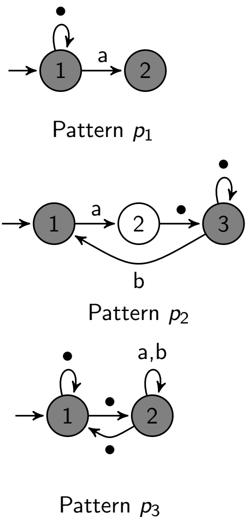


Pattern p_3

Checking patterns

$$\mathcal{A} = \{\texttt{open}(f), \texttt{read}(f), \texttt{write}(f), \texttt{close}(f), \texttt{delete}(f)\}$$

Pattern Library

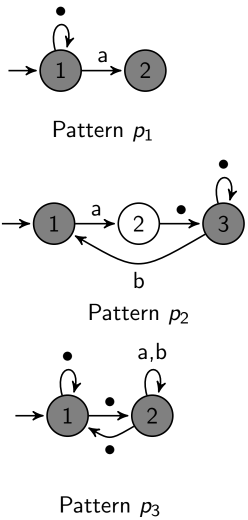


Checking patterns

$$\mathcal{A} = \{\texttt{open}(f), \texttt{read}(f), \texttt{write}(f), \texttt{close}(f), \texttt{delete}(f)\}$$

Binding	Pattern a	Pattern b	Passed
$[f \mapsto 1]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	open(f)	delete(f)	p_2
	read(f)	write(f)	p_3
	read(f)	close(f)	p_3
	write(f)	read(f)	p_3
$[f \mapsto 2]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	read(f)	write(f)	p_3
	write(f)	read(f)	p_3

Pattern Library

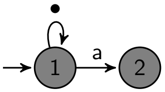


Checking patterns

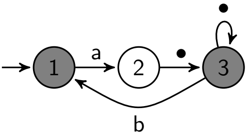
$$\sigma = \texttt{open}(1).\texttt{open}(2).\texttt{read}(1).\texttt{write}(2).\texttt{close}(1).\texttt{close}(2).\texttt{delete}(1)$$

Binding	Pattern		Passed
	a	b	
$[f \mapsto 1]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	open(f)	delete(f)	p_2
	read(f)	write(f)	p_3
	read(f)	close(f)	p_3
	write(f)	read(f)	p_3
$[f \mapsto 2]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	read(f)	write(f)	p_3
	write(f)	read(f)	p_3

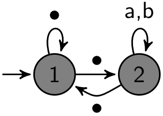
Pattern Library



Pattern p_1



Pattern p_2



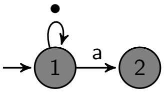
Pattern p_3

Checking patterns

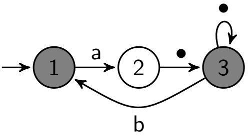
$$\sigma = \texttt{open}(1).\texttt{open}(2).\texttt{read}(1).\texttt{write}(2).\texttt{close}(1).\texttt{close}(2).\texttt{delete}(1)$$

Binding	Pattern		Passed
	a	b	
$[f \mapsto 1]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	open(f)	delete(f)	p_2
	read(f)	write(f)	p_3
	read(f)	close(f)	p_3
	write(f)	read(f)	p_3
$[f \mapsto 2]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	read(f)	write(f)	p_3
	write(f)	read(f)	p_3

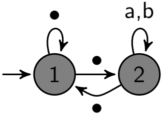
Pattern Library



Pattern p_1



Pattern p_2



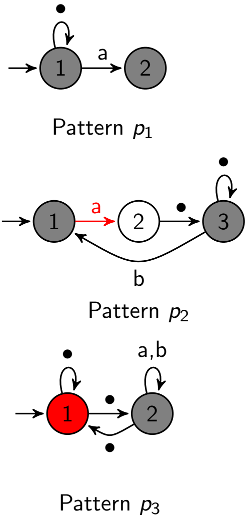
Pattern p_3

Checking patterns

$$\sigma = \texttt{open(1).open(2).read(1).write(2).close(1).close(2).delete(1)}$$

Binding	Pattern a	Pattern b	Passed
$[f \mapsto 1]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	open(f)	delete(f)	p_2
	read(f)	write(f)	p_3
	read(f)	close(f)	p_3
	write(f)	read(f)	p_3
$[f \mapsto 2]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	read(f)	write(f)	p_3
	write(f)	read(f)	p_3

Pattern Library

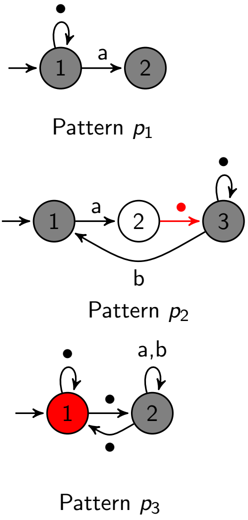


Checking patterns

$$\sigma = \texttt{open(1).open(2).read(1).write(2).close(1).close(2).delete(1)}$$

Binding	Pattern a	Pattern b	Passed
$[f \mapsto 1]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	open(f)	delete(f)	p_2
	read(f)	write(f)	p_3
	read(f)	close(f)	p_3
	write(f)	read(f)	p_3
$[f \mapsto 2]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	read(f)	write(f)	p_3
	write(f)	read(f)	p_3

Pattern Library

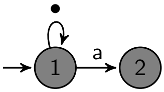


Checking patterns

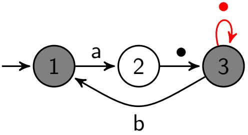
$$\sigma = \texttt{open}(1).\texttt{open}(2).\texttt{read}(1).\texttt{write}(2).\texttt{close}(1).\texttt{close}(2).\texttt{delete}(1)$$

Binding	Pattern		Passed
	a	b	
$[f \mapsto 1]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	open(f)	delete(f)	p_2
	read(f)	write(f)	p_3
	read(f)	close(f)	p_3
	write(f)	read(f)	p_3
$[f \mapsto 2]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	read(f)	write(f)	p_3
	write(f)	read(f)	p_3

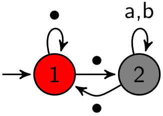
Pattern Library



Pattern p_1



Pattern p_2



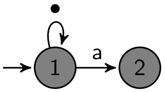
Pattern p_3

Checking patterns

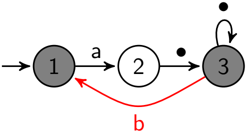
$$\sigma = \texttt{open(1).open(2).read(1).write(2).close(1).close(2).delete(1)}$$

Binding	Pattern a	Pattern b	Passed
$[f \mapsto 1]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	open(f)	delete(f)	p_2
	read(f)	write(f)	p_3
	read(f)	close(f)	p_3
	write(f)	read(f)	p_3
$[f \mapsto 2]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	read(f)	write(f)	p_3
	write(f)	read(f)	p_3

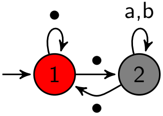
Pattern Library



Pattern p_1



Pattern p_2



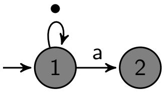
Pattern p_3

Checking patterns

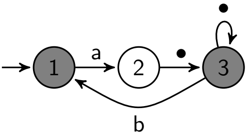
$$\sigma = \texttt{open}(1).\texttt{open}(2).\texttt{read}(1).\texttt{write}(2).\texttt{close}(1).\texttt{close}(2).\texttt{delete}(1)$$

Binding	Pattern		Passed
	a	b	
$[f \mapsto 1]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	open(f)	delete(f)	p_2
	read(f)	write(f)	p_3
	read(f)	close(f)	p_3
	write(f)	read(f)	p_3
$[f \mapsto 2]$	delete(f)	-	p_1
	open(f)	close(f)	p_2
	read(f)	write(f)	p_3
	write(f)	read(f)	p_3

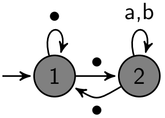
Pattern Library



Pattern p_1



Pattern p_2



Pattern p_3

Quantify

- The quantifications tell us to select the successful patterns *for all files*

Binding	Pattern		Passed	$\forall f$?
	a	b		
$[f \mapsto 1]$	delete(f)	-	p_1	
	open(f)	close(f)	p_2	
	open(f)	delete(f)	p_2	
	read(f)	write(f)	p_3	
	read(f)	close(f)	p_3	
	write(f)	read(f)	p_3	
$[f \mapsto 2]$	delete(f)	-	p_1	
	open(f)	close(f)	p_2	
	read(f)	write(f)	p_3	
	write(f)	read(f)	p_3	

Quantify

- The quantifications tell us to select the successful patterns *for all files*

Binding	Pattern a	Pattern b	Passed	$\forall f$?
$[f \mapsto 1]$	delete(f)	-	p_1	✓
	open(f)	close(f)	p_2	
	open(f)	delete(f)	p_2	
	read(f)	write(f)	p_3	
	read(f)	close(f)	p_3	
	write(f)	read(f)	p_3	
$[f \mapsto 2]$	delete(f)	-	p_1	✓
	open(f)	close(f)	p_2	
	read(f)	write(f)	p_3	
	write(f)	read(f)	p_3	

Quantify

- The quantifications tell us to select the successful patterns *for all files*

Binding	Pattern a	Pattern b	Passed	$\forall f$?
$[f \mapsto 1]$	delete(f)	-	p_1	✓
	open(f)	close(f)	p_2	✓
	open(f)	delete(f)	p_2	
	read(f)	write(f)	p_3	
	read(f)	close(f)	p_3	
	write(f)	read(f)	p_3	
$[f \mapsto 2]$	delete(f)	-	p_1	✓
	open(f)	close(f)	p_2	✓
	read(f)	write(f)	p_3	
	write(f)	read(f)	p_3	

Quantify

- The quantifications tell us to select the successful patterns *for all files*

Binding	Pattern a	Pattern b	Passed	$\forall f$?
$[f \mapsto 1]$	delete(f)	-	p_1	✓
	open(f)	close(f)	p_2	✓
	open(f)	delete(f)	p_2	✗
	read(f)	write(f)	p_3	
	read(f)	close(f)	p_3	
	write(f)	read(f)	p_3	
$[f \mapsto 2]$	delete(f)	-	p_1	✓
	open(f)	close(f)	p_2	✓
	read(f)	write(f)	p_3	
	write(f)	read(f)	p_3	

Quantify

- The quantifications tell us to select the successful patterns *for all files*

Binding	Pattern a	Pattern b	Passed	$\forall f$?
$[f \mapsto 1]$	delete(f)	-	p_1	✓
	open(f)	close(f)	p_2	✓
	open(f)	delete(f)	p_2	✗
	read(f)	write(f)	p_3	✓
	read(f)	close(f)	p_3	
	write(f)	read(f)	p_3	
$[f \mapsto 2]$	delete(f)	-	p_1	✓
	open(f)	close(f)	p_2	✓
	read(f)	write(f)	p_3	✓
	write(f)	read(f)	p_3	

Quantify

- The quantifications tell us to select the successful patterns *for all files*

Binding	Pattern a	Pattern b	Passed	$\forall f$?
$[f \mapsto 1]$	delete(f)	-	p_1	✓
	open(f)	close(f)	p_2	✓
	open(f)	delete(f)	p_2	✗
	read(f)	write(f)	p_3	✓
	read(f)	close(f)	p_3	✗
	write(f)	read(f)	p_3	
$[f \mapsto 2]$	delete(f)	-	p_1	✓
	open(f)	close(f)	p_2	✓
	read(f)	write(f)	p_3	✓
	write(f)	read(f)	p_3	

Quantify

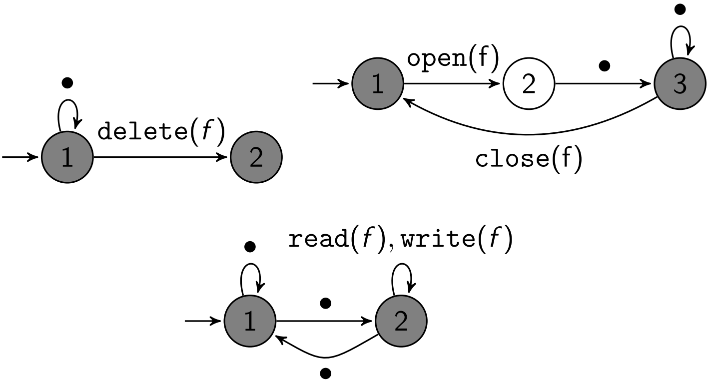
- The quantifications tell us to select the successful patterns *for all files*

Binding	Pattern		Passed	$\forall f$?
	a	b		
$[f \mapsto 1]$	delete(f)	-	p_1	✓
	open(f)	close(f)	p_2	✓
	open(f)	delete(f)	p_2	✗
	read(f)	write(f)	p_3	✓
	read(f)	close(f)	p_3	✗
	write(f)	read(f)	p_3	✓
$[f \mapsto 2]$	delete(f)	-	p_1	✓
	open(f)	close(f)	p_2	✓
	read(f)	write(f)	p_3	✓
	write(f)	read(f)	p_3	✓

- There are four successful patterns

Successful patterns

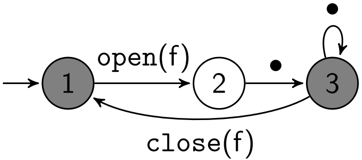
- Our four successful patterns become three due to symmetry

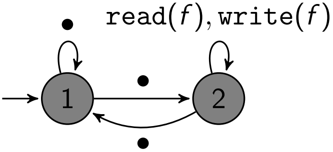


- Each of these tell us something about how their events order with each other *and other events not mentioned*

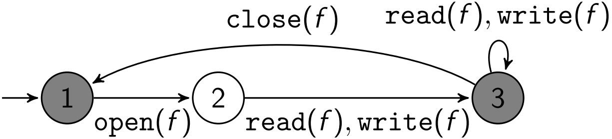
Combining successful patterns

- We can then construct an (open) automaton that accepts the intersection of their languages
- First combining these two patterns



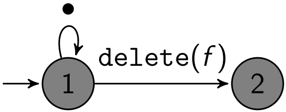


- Gives us

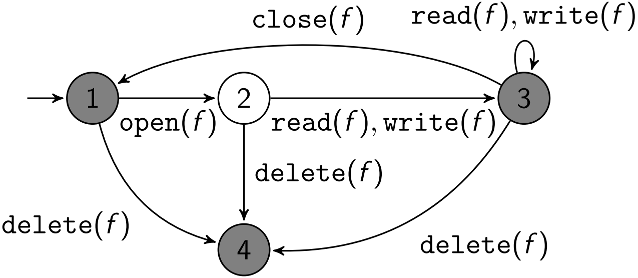


Combining successful patterns

- We can then construct an (open) automaton that accepts the intersection of their languages
- Then combining the result with

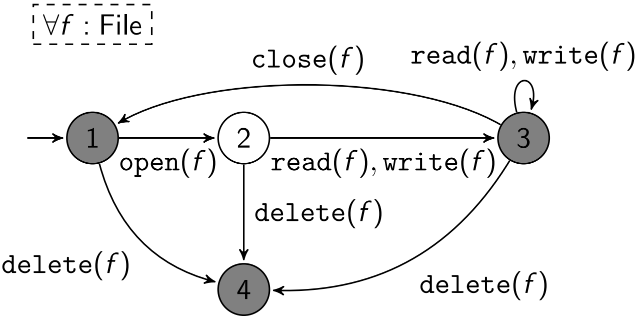


- Gives us



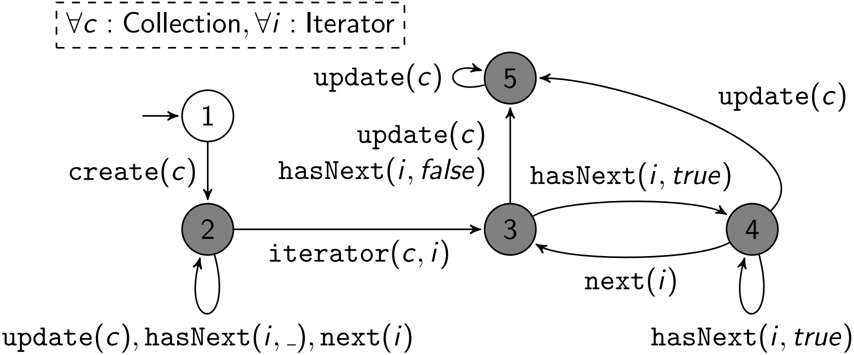
Combining successful patterns

- We can then construct an (open) automaton that accepts the intersection of their languages
- Leading to the Quantified Event Automaton



Something from the real world

- This example with files has been necessarily simple, here is an example of a property mined from the Java standard library
- This states that an iterator created from a collection cannot be used after the collection is updated



The approach I didn't use

Automata-learning / regular-inference

- Passive via state merging
  - Construct an automaton that accepts exactly the traces
  - Merge 'equivalent' states for some notion of equivalence
- Active via L* (Angluin's) - requires an oracle

- So why have I chosen to use this pattern-based approach? There are two main reasons
  1. Algorithms from runtime verification allow large traces to be processed efficiently
  2. The pattern-based approach supports a language with any level of expressiveness - we only need a trace checking function

The approach I didn't use

Automata-learning / regular-inference

- Passive via state merging
  - Construct an automaton that accepts exactly the traces
  - Merge 'equivalent' states for some notion of equivalence
- Active via L^* (Angluin's) - requires an oracle

- So why have I chosen to use this pattern-based approach? There are two main reasons
  1. Algorithms from runtime verification allow large traces to be processed efficiently
  2. The pattern-based approach supports a language with any level of expressiveness - we only need a trace checking function

The approach I didn't use

Automata-learning / regular-inference

- Passive via state merging
  - Construct an automaton that accepts exactly the traces
  - Merge 'equivalent' states for some notion of equivalence
- Active via L* (Angluin's) - requires an oracle

- So why have I chosen to use this pattern-based approach? There are two main reasons
  1. Algorithms from runtime verification allow large traces to be processed efficiently
  2. The pattern-based approach supports a language with any level of expressiveness - we only need a trace checking function

An overview of further work

- Extending to full expressiveness of QEA
- Developing a methodology for finding suitable pattern libraries
- Methods for identifying likely alphabets

Conclusion

We have

- Introduced the idea of *specification inference*
- Outlined a pattern-based approach that deals with data effectively and allows for efficient mining
- Presented an example of how this works
- Discussed some outstanding issues

Any Questions?

Where do patterns come from?

- A pattern library needs to be defined before mining
- There are two interesting questions
  1. How do we do this?
  2. Does it matter?
- Let us consider the second by posing the question - Is there a pattern library that will always give the desired specification? i.e. one that gives a solution to the specification inference problem
- To answer this we must consider what a solution looks like

Why there is no solution

- Complete learnability requires an impracticable amount of information - we must generalise

Why there is no solution

- Complete learnability requires an impracticable amount of information - we must generalise
- We see two traces

```
open.read.close.open.read.close
open.read.write.save.close
```

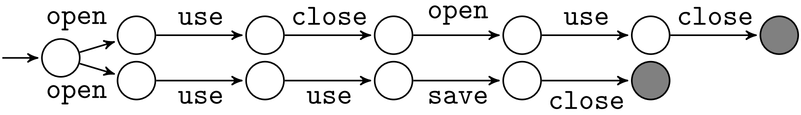
- What property can we infer?

Why there is no solution

- Complete learnability requires an impracticable amount of information - we must generalise
- We see two traces

```
open.read.close.open.read.close
open.read.write.save.close
```

- What property can we infer?

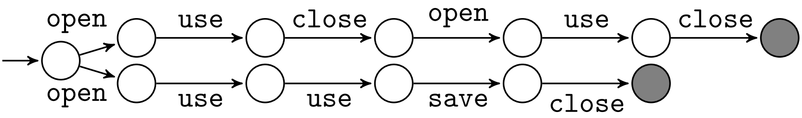


Why there is no solution

- Complete learnability requires an impracticable amount of information - we must generalise
- We see two traces

```
open.read.close.open.read.close
open.read.write.save.close
```

- What property can we infer?



- Now we receive this trace

```
open.read.close.open.close
```

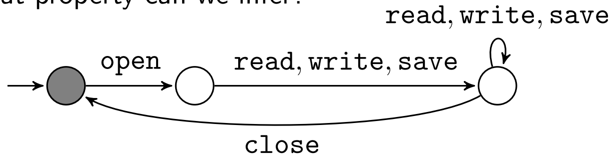
- Our specification is too specific

Why there is no solution

- Complete learnability requires an impracticable amount of information - we must generalise
- We see two traces

open.read.close.open.read.close
open.read.write.save.close

- What property can we infer?



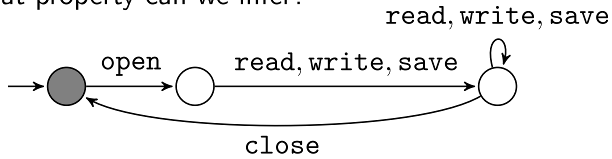
Why there is no solution

- Complete learnability requires an impracticable amount of information - we must generalise
- We see two traces

  `open.read.close.open.read.close`

  `open.read.write.save.close`

- What property can we infer?



- But if we have to save after writing we incorrectly accept

  `open.read.write.save.write.close`

- Our specification is too general

Why there is no solution

- Complete learnability requires an impracticable amount of information - we must generalise
- It is known that without negative information we cannot exactly identify a language
- As combination is the straight-forward intersection of languages the pattern library can cause
    - over-specification, for example if it contains a pattern accepting exactly the input traces
    - over-generalisation, when the desired specification is very specific
- Additionally, a pattern library may be too small or lack the coverage to identify a specification at all

Making patterns

- Intuition and experience
  - There exist studies identifying common shapes in specifications
  - There are also exist common methods for combining specifications that can be formed as patterns
- Exploration
  - Given a known specification we can perform the reverse of combination to give us patterns
- Automatic generation/enumeration
  - We can use different methods to automatically generate patterns either by enumeration up to a certain size or by performing operations on a set of existing patterns
  - Disadvantage - likely to introduce noise
- A full methodology for pattern library definition remains further work

Where to start

- Our process begins having collected a set of traces
- But to do this we assume we already know the alphabet of the specification (this is another input)
- This is similar to other processes like ours
- Identifying likely alphabets remains further work