MarQ: Monitoring At Runtime with QEA

Giles Reger

in collaboration with

Helena Cuenca Cruz, David Rydeheard

at University of Manchester, UK

April 17th, 2015

Outline

Runtime Monitoring

Quantified Event Automata

Efficient monitoring

Using MarQ

Motivation

- See lots of other talks for why we want formal guarantees about correctness of software systems

- Static verification has many successes but
  - It can have scalability issues
  - It often works with abstractions of the real system
  - It often needs to make assumptions about the environment and input data

- Runtime verification is a complementary technique that tackles these issues by ‘verifying’ a single run of the system
- Additionally, if performed at runtime it can be used to *stop* or *correct* bad behaviour

Runtime Monitoring Problems

Runtime Monitoring

Checking whether an execution trace τ produced at runtime satisfies a given a (typically temporal) specification ϕ

Online Runtime Monitoring

Performing runtime monitoring alongside the running system.

Offline Runtime Monitoring

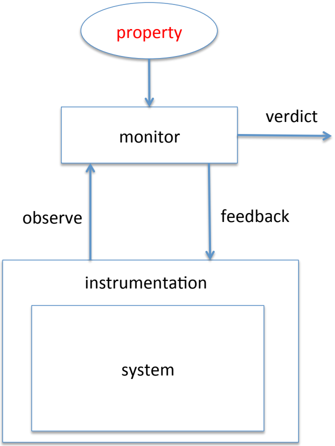
Performing runtime monitoring on a log file after running the system.

Parametric Runtime Monitoring

Runtime monitoring with *first-order* specifications i.e. ones that deal with data-carrying events

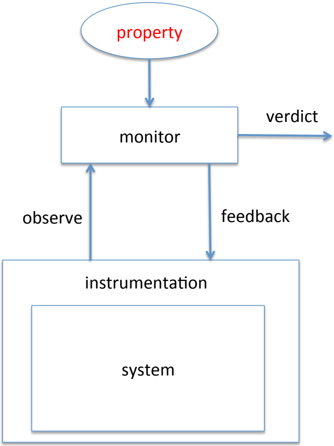
Online Runtime Monitoring Setup

Instrument the system to observe a trace of relevant events



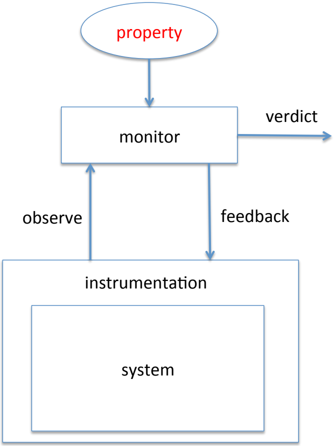
Online Runtime Monitoring Setup

The monitor uses the given property . . .



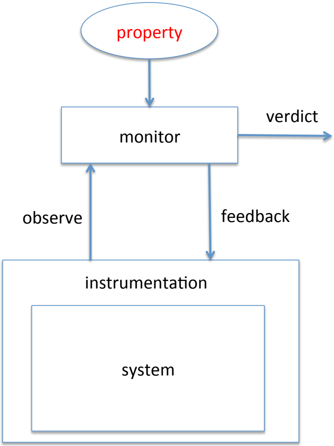
Online Runtime Monitoring Setup

. . . to process each event . . . possibly providing feedback. . .



Online Runtime Monitoring Setup

. . . and finally computing a verdict - did the system pass?



Online Runtime Monitoring in Practice

- Lots of pragmatic considerations
  - Instrumentation
  - Overhead
  - Interference
- Commonly shown to be useful for checking usage of libraries - successful application to large open source projects
- Recent industrial successes in the banking industry for monitoring reliability and correctness
- Applicable to safety-critical systems i.e. aerospace, automotive, medical.

Offline Runtime Monitoring in Practice

- Idea: record behaviour and check afterwards
  - Gives minimal/predictable overhead
  - Applies to more general domains/settings
  - Only get what is recorded
- Offline RV successes at NASA's JPL
  - Used on the LADEE mission to check command sequences sent to the spacecraft as part of a daily testing regime
  - Used daily on the MSL mission to check rules against Spacecraft telemetry logs sent from Curiosity

Parametric (or first-order) Properties

- Originally runtime verification considered properties over event names i.e. using propositional LTL or automata
- A *parametric event* consists of a name and a list of data values
- Examples:
  - An iterator i created from a collection c should not be used after c is updated
  - Every `start`(t) should have a corresponding `stop`(t)
  - If locks l_1 and l_2 are taken in one order by a thread t then later they should not be taken in the reverse order by any thread
- Introduces new challenges in terms of specification languages and monitoring algorithms

Contributions

- We introduce the MarQ runtime monitoring tool
- MarQ stands for Monitoring at runtime with QEA
- Quantified Event Automata (QEA) is a previously introduced specification language for parametric specifications

MarQ

- Can be used *offline* and *online*
- Supports all features of the QEA language
- Is efficient
    - Won the Offline and Java tracks of the CRV14, the first international competition on runtime verification.
    - Incorporates novel indexing, redundancy elimination and structural specialisation techniques

Outline

The Slicing idea

- Based on the notion of *parametric trace slicing*
  - Turns a quantified problem into a set of unquantified problems
- The basic idea of QEA
  1. Use a list of *quantifications* to define *trace slices* relating to separate valuations of quantified variables
  2. Use an *extended finite state machine* to check properties over those slices
  3. The quantifications define which trace slices need to be accepted by the state machine

Quantified Event Automata

Definition (Event Automaton)

An Event Automaton $\langle Q, \mathcal{A}, \delta, q_0, F \rangle$ is a tuple where

- Q is a set of states,
- $\mathcal{A} \subseteq \mathit{SymbolicEvent}$ is a alphabet of events,
- $\delta \subseteq (Q \times \mathcal{A} \times \mathit{Guard} \times \mathit{Assign} \times Q)$ is a set of transitions,
- q_0 is an initial state, and
- $F \subseteq Q$ is a set of final states.

Definition (Quantified Event Automaton)

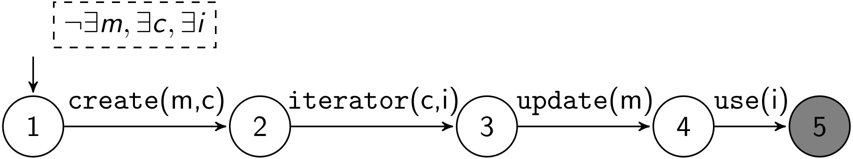
A QEA is a pair $\langle \Lambda, \mathrm{E} \rangle$ where

- $\Lambda \in (\{\forall, \exists\} \times \texttt{variables}(\mathrm{E}) \times \mathit{Guard})^*$ is a list of quantified variables with guards, and
- E is an Event Automaton

UnsafeMapIterator Example

Property : UnsafeMapIterator

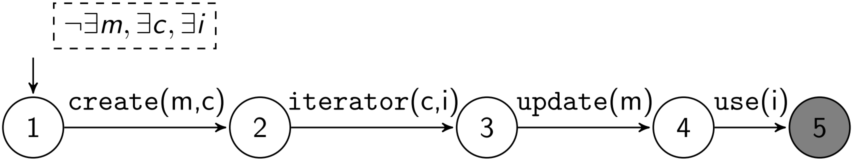
An iterator created from a collection created from a map should not be used after the map is updated.



Demonstrating slicing

$\texttt{create}(M, C1).\texttt{iterator}(C1, I1).\texttt{use}(I1).\texttt{update}(M).\texttt{create}(M, C2).$
$\texttt{iterator}(C2, I2).\texttt{iterator}(C2, I3).\texttt{use}(I3).\texttt{update}(M).\texttt{use}(I2)$

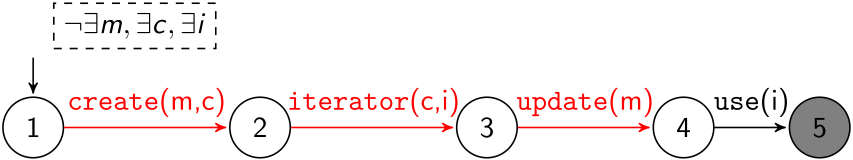
There are six possible bindings



Demonstrating slicing

create(M, $C1$).iterator($C1$, $I1$).use($I1$).update(M).create(M, $C2$).
iterator($C2$, $I2$).iterator($C2$, $I3$).use($I3$).update(M).use($I2$)

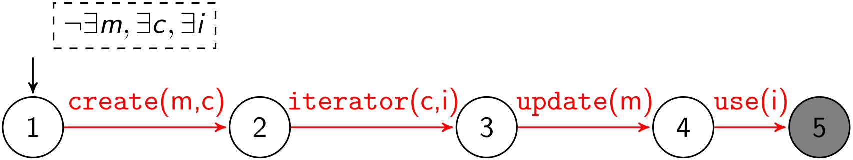
For $m = M$, $c = C1$, $i = I1$



Demonstrating slicing

create($M, C1$).iterator($C1, I1$).use($I1$).update(M).create($M, C2$).
iterator($C2, I2$).iterator($C2, I3$).use($I3$).update(M).use($I2$)

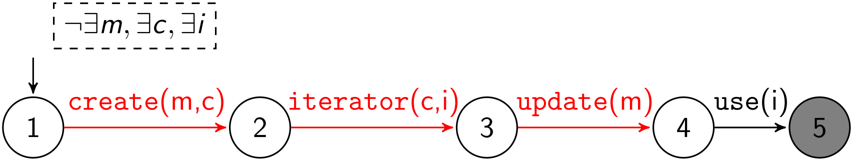
For $m = M$, $c = C2$, $i = I2$



Demonstrating slicing

create($M, C1$).iterator($C1, I1$).use($I1$).update(M).create($M, C2$).
iterator($C2, I2$).iterator($C2, I3$).use($I3$).update(M).use($I2$)

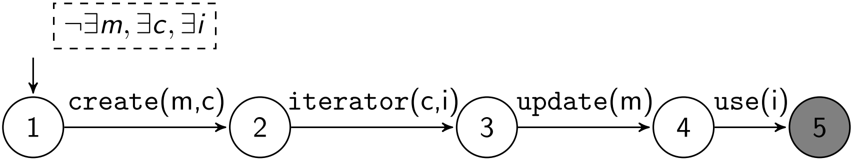
For $m = M$, $c = C2$, $i = I3$



Demonstrating slicing

create(M, $C1$).iterator($C1$, $I1$).use($I1$).update(M).create(M, $C2$).
iterator($C2$, $I2$).iterator($C2$, $I3$).use($I3$).update(M).use($I2$)

There exists a slice that reaches a final state. The quantifications mean that the trace violates the property.



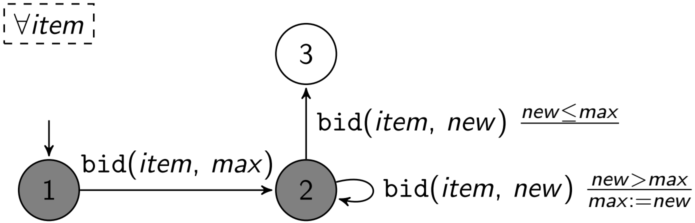
Free Variables

- Some variables in the Event Automaton may not be quantified
- These are called free variables
- Free variables are (re)bound as the trace is processed
- Allowing us to capture changing data values

Auction Bidding Example

Property : Auction Bidding

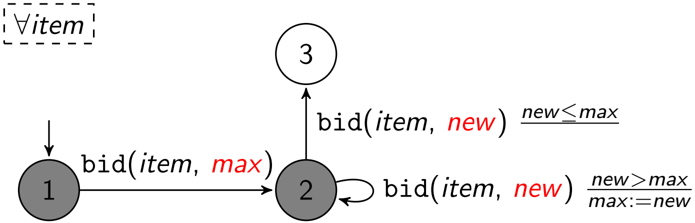
Amounts bid for an item should be strictly increasing.



Auction Bidding Example

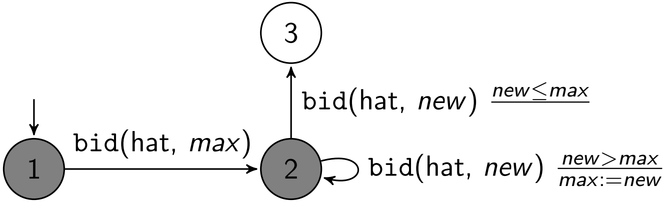
Property : Auction Bidding

Amounts bid for an item should be strictly increasing.



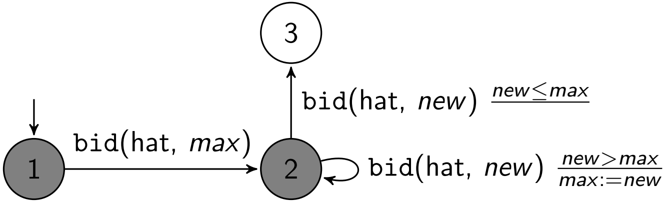
Bidding For A Hat

bid(hat, 5).bid(hat, 10).bid(hat, 7)



Bidding For A Hat

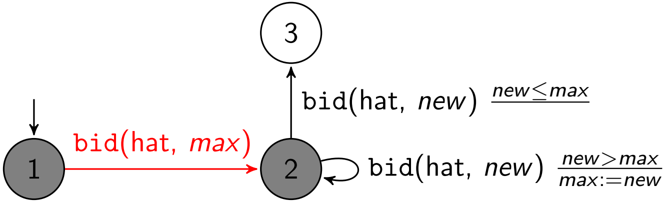
$\texttt{bid}(\text{hat}, 5).\texttt{bid}(\text{hat}, 10).\texttt{bid}(\text{hat}, 7)$



$\langle 1, [\ ]\rangle$

Bidding For A Hat

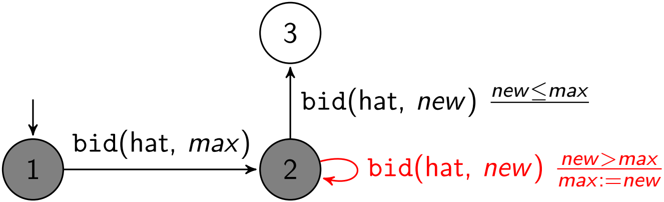
$\texttt{bid}(\text{hat}, 5).\texttt{bid}(\text{hat}, 10).\texttt{bid}(\text{hat}, 7)$



$\langle 1, [\,]\rangle \xrightarrow{\texttt{bid}(\text{hat},5)} \langle 2, [max \mapsto 5]\rangle$

Bidding For A Hat

$\texttt{bid}(\text{hat}, 5).\texttt{bid}(\text{hat}, 10).\texttt{bid}(\text{hat}, 7)$

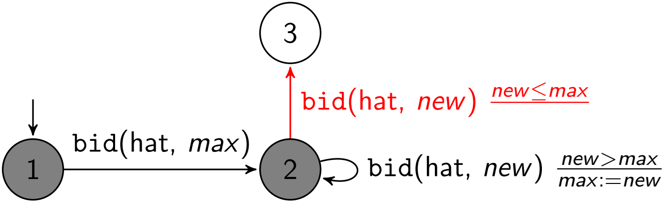


$$\langle 1, [\ ]\rangle \xrightarrow{\texttt{bid}(\text{hat},5)} \langle 2, [max \mapsto 5]\rangle$$

$$\xrightarrow{\texttt{bid}(\text{hat},10)} \langle 2, [new \mapsto 10, max \mapsto 10]\rangle$$

Bidding For A Hat

$\texttt{bid}(\text{hat}, 5).\texttt{bid}(\text{hat}, 10).\texttt{bid}(\text{hat}, 7)$



$$\langle 1, [\,]\rangle \xrightarrow{\texttt{bid}(\text{hat},5)} \langle 2, [max \mapsto 5]\rangle$$

$$\xrightarrow{\texttt{bid}(\text{hat},10)} \langle 2, [new \mapsto 10, max \mapsto 10]\rangle$$

$$\xrightarrow{\texttt{bid}(\text{hat},7)} \langle 3, [new \mapsto 7, max \mapsto 10]\rangle$$

Outline

Runtime Monitoring

Quantified Event Automata

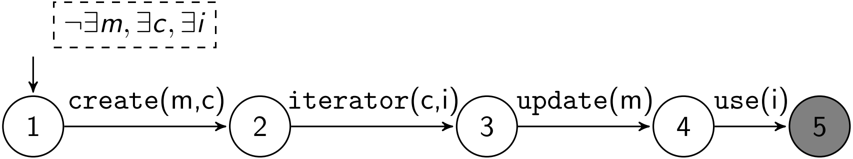
Efficient monitoring

Using MarQ

UnsafeMapIterator Example

Property : UnsafeMapIterator

An iterator created from a collection created from a map should not be used after the map is updated.



Monitoring algorithm

The monitoring algorithm is organised as follows

- There is a map M from bindings (of quantified variables) to sets of configurations (reached by that slice)
- For each incoming event
  - Decides if new bindings need to be created, possibly extending existing bindings
  - Updates configurations related to existing relevant bindings
  - Produces a verdict based on the quantifications and M

bindings		configurations
m=M1,c=C1	$\longrightarrow$	{ 2 }
m=M1,c=C1,i=I1	$\longrightarrow$	{ 3 }

Monitoring algorithm

The monitoring algorithm is organised as follows

- There is a map M from bindings (of quantified variables) to sets of configurations (reached by that slice)
- For each incoming event
  - Decides if new bindings need to be created, possibly extending existing bindings
  - Updates configurations related to existing relevant bindings
  - Produces a verdict based on the quantifications and M

bindings		configurations
m=M1,c=C1	⟶	{ 2 }
m=M1,c=C1,i=I1	⟶	{ 3 }

Monitoring algorithm

The monitoring algorithm is organised as follows

- There is a map M from bindings (of quantified variables) to sets of configurations (reached by that slice)
- For each incoming event `iterator`(C1,I2)
  - Decides if new bindings need to be created, possibly extending existing bindings
  - Updates configurations related to existing relevant bindings
  - Produces a verdict based on the quantifications and M

bindings		configurations
m=M1,c=C1	$\longrightarrow$	{ 2 }
m=M1,c=C1,i=I1	$\longrightarrow$	{ 3 }

Monitoring algorithm

The monitoring algorithm is organised as follows

- There is a map M from bindings (of quantified variables) to sets of configurations (reached by that slice)
- For each incoming event `iterator(C1,I2)`
  - Decides if new bindings need to be created, possibly extending existing bindings
  - Updates configurations related to existing relevant bindings
  - Produces a verdict based on the quantifications and M

bindings		configurations
m=M1,c=C1	$\longrightarrow$	{ 2 }
m=M1,c=C1,i=I1	$\longrightarrow$	{ 3 }
m=M1,c=C1,i=I2	$\longrightarrow$	{ 3 }

Monitoring algorithm

The monitoring algorithm is organised as follows

- There is a map M from bindings (of quantified variables) to sets of configurations (reached by that slice)
- For each incoming event `iterator`(C1,I2)
  - Decides if new bindings need to be created, possibly extending existing bindings
  - Updates configurations related to existing relevant bindings
  - Produces a verdict based on the quantifications and M

bindings		configurations
m=M1,c=C1	$\longrightarrow$	{ 2 }
m=M1,c=C1,i=I1	$\longrightarrow$	{ 3 }
m=M1,c=C1,i=I2	$\longrightarrow$	{ 3 }

Monitoring algorithm

The monitoring algorithm is organised as follows

- There is a map M from bindings (of quantified variables) to sets of configurations (reached by that slice)
- For each incoming event `update(M1)`
  - Decides if new bindings need to be created, possibly extending existing bindings
  - Updates configurations related to existing relevant bindings
  - Produces a verdict based on the quantifications and M

bindings		configurations
m=M1,c=C1	$\longrightarrow$	{ 2 }
m=M1,c=C1,i=I1	$\longrightarrow$	{ 3 }
m=M1,c=C1,i=I2	$\longrightarrow$	{ 3 }

Monitoring algorithm

The monitoring algorithm is organised as follows

- There is a map M from bindings (of quantified variables) to sets of configurations (reached by that slice)
- For each incoming event `update(M1)`
  - Decides if new bindings need to be created, possibly extending existing bindings
  - Updates configurations related to existing relevant bindings
  - Produces a verdict based on the quantifications and M

bindings		configurations
m=M1,c=C1	$\longrightarrow$	{ 2 }
m=M1,c=C1,i=I1	$\longrightarrow$	{ 3 }
m=M1,c=C1,i=I2	$\longrightarrow$	{ 3 }

Monitoring algorithm

The monitoring algorithm is organised as follows

- There is a map M from bindings (of quantified variables) to sets of configurations (reached by that slice)
- For each incoming event `update(M1)`
  - Decides if new bindings need to be created, possibly extending existing bindings
  - Updates configurations related to existing relevant bindings
  - Produces a verdict based on the quantifications and M

bindings		configurations
m=M1,c=C1	$\longrightarrow$	{ 2 }
m=M1,c=C1,i=I1	$\longrightarrow$	{ 4 }
m=M1,c=C1,i=I2	$\longrightarrow$	{ 4 }

Monitoring algorithm

The monitoring algorithm is organised as follows

- There is a map M from bindings (of quantified variables) to sets of configurations (reached by that slice)
- For each incoming event
  - Decides if new bindings need to be created, possibly extending existing bindings
  - Updates configurations related to existing relevant bindings
  - Produces a verdict based on the quantifications and M

bindings		configurations
m=M1,c=C1	⟶	{ 2 }
m=M1,c=C1,i=I1	⟶	{ 4 }
m=M1,c=C1,i=I2	⟶	{ 4 }

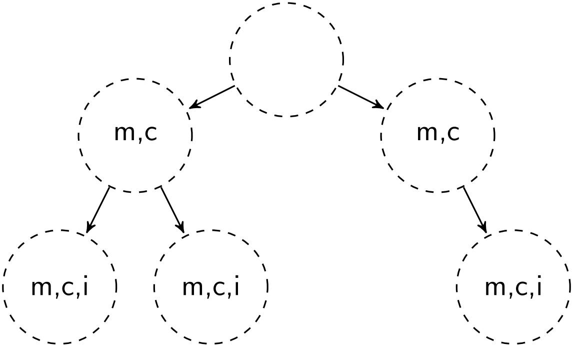
Efficient monitoring with MarQ

- Obviously for *online* monitoring *overhead* and *interference* are very important
- For *offline* monitoring we still need practically efficient trace processing

- MarQ achieves efficient monitoring in three ways
  - Indexing
  - Redundancy elimination
  - Structural specialisation
- We give a flavour of these techniques here

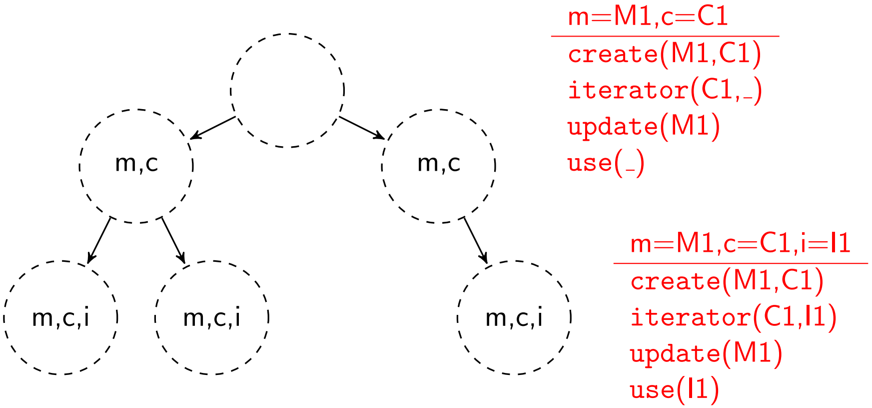
Indexing

- The monitor needs to track the status of different valuations of quantified variables
- Given an event it needs to *quickly* find what needs to be updated. This is the indexing problem.
- MarQ uses *symbol-based indexing*



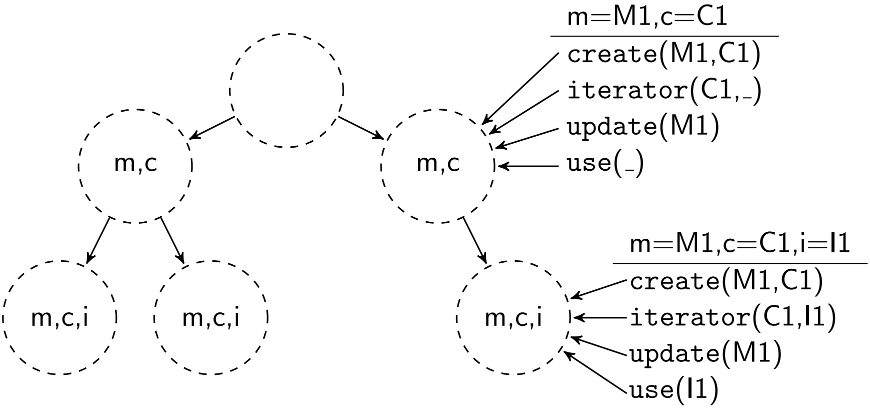
Indexing

- MarQ uses *symbol-based indexing*



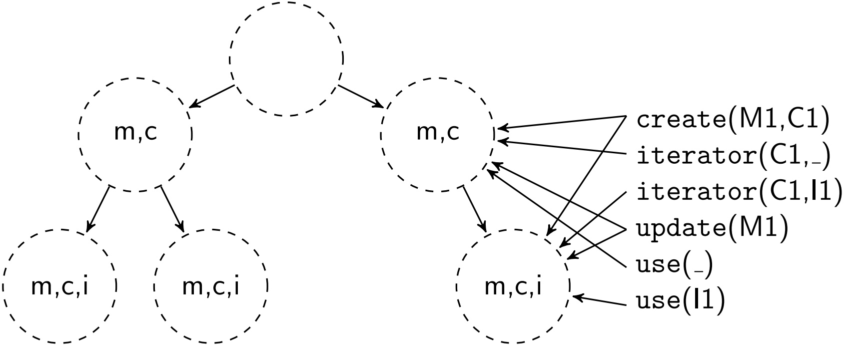
Indexing

- MarQ uses *symbol-based indexing*



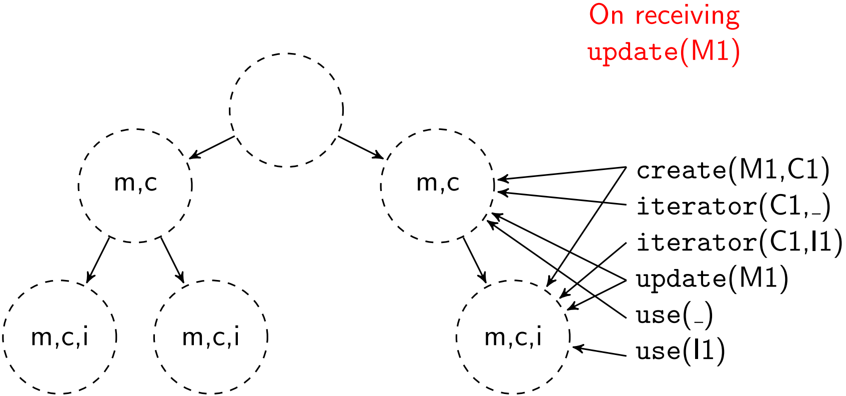
Indexing

- MarQ uses *symbol-based indexing*



Indexing

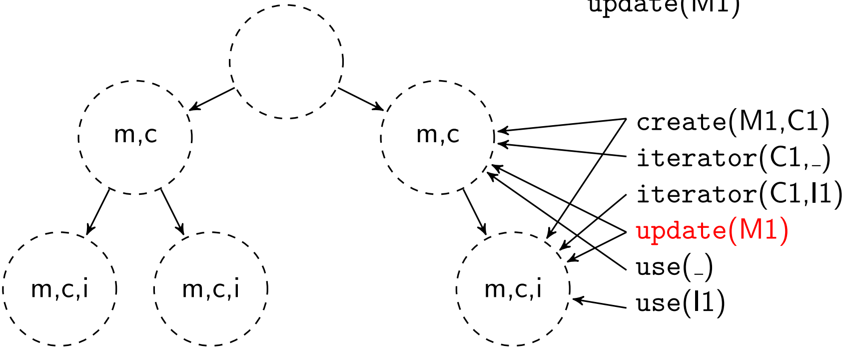
- MarQ uses *symbol-based indexing*



Indexing

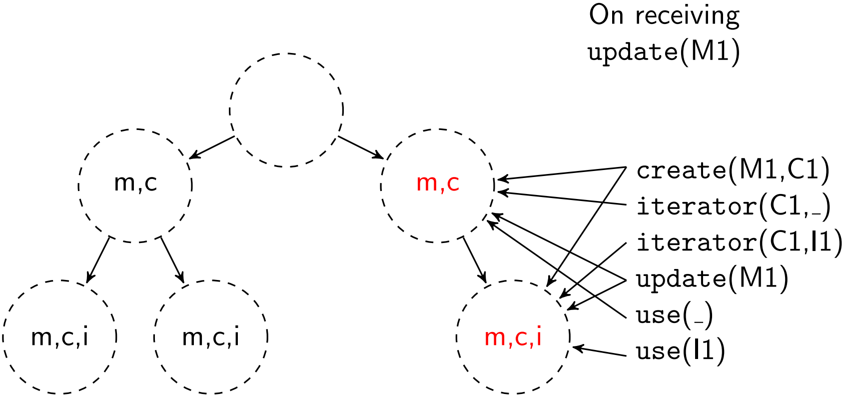
- MarQ uses *symbol-based indexing*

On receiving
update(M1)



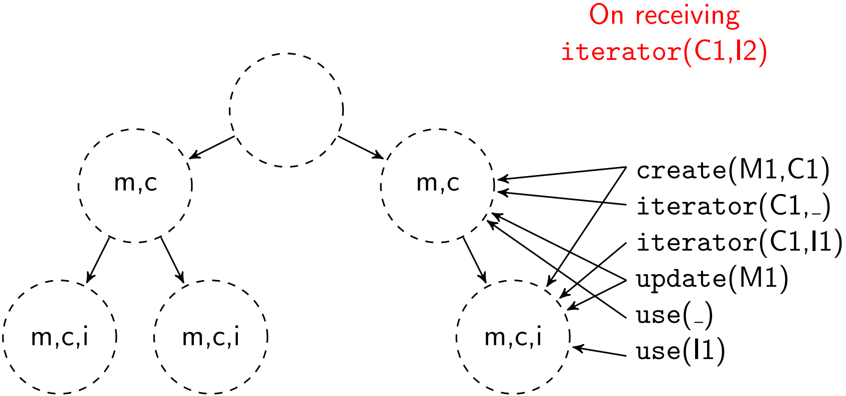
Indexing

- MarQ uses *symbol-based indexing*



Indexing

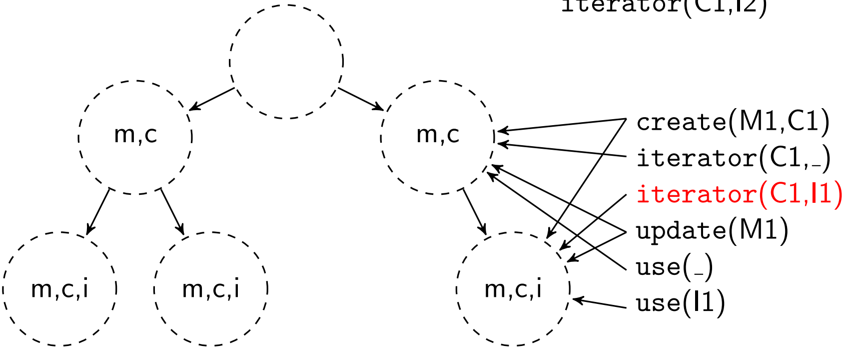
- MarQ uses *symbol-based indexing*



Indexing

- MarQ uses *symbol-based indexing*

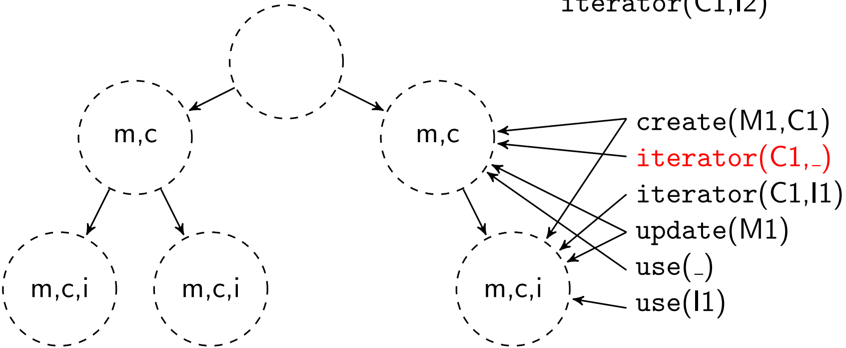
On receiving
iterator(C1,I2)



Indexing

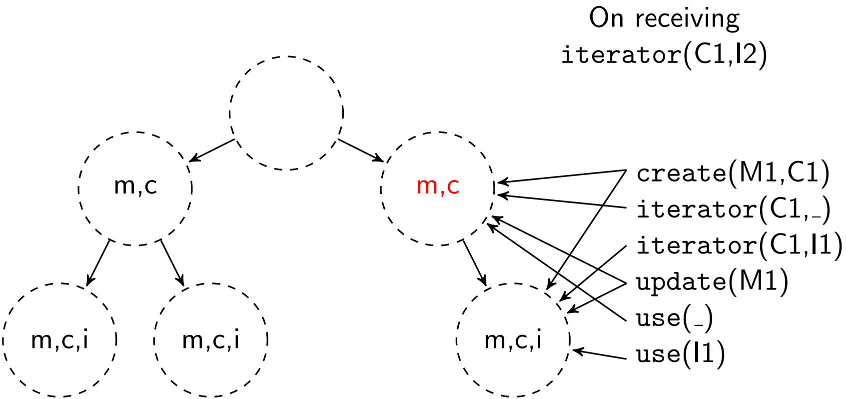
- MarQ uses *symbol-based indexing*

On receiving
iterator(C1,I2)



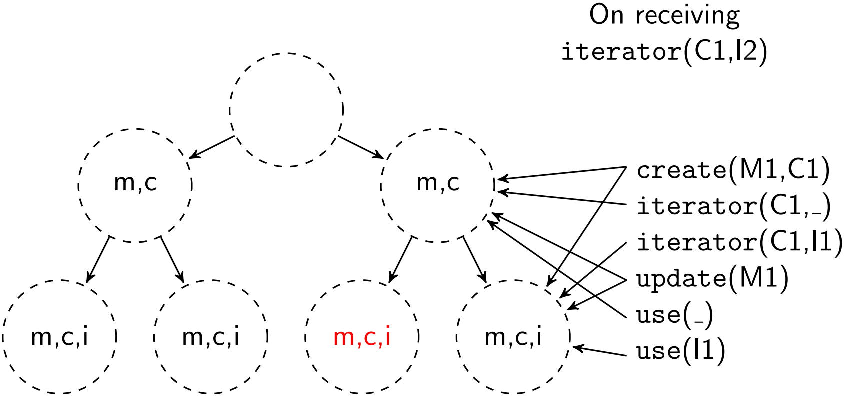
Indexing

- MarQ uses *symbol-based indexing*

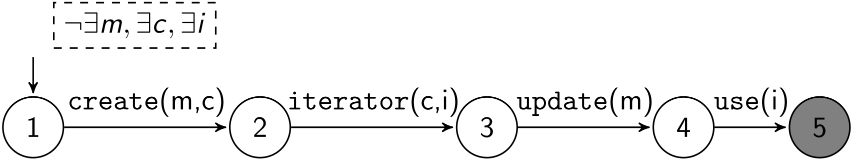


Indexing

- MarQ uses *symbol-based indexing*



Redundancy elimination



- Given this specification there are two observations
  1. A binding m=M,c=C does not need to be extended for use(I) as this does not tell us anything new
  2. If a monitored object is garbage collected it can no longer contribute to a failing trace slice
- Generalising these gives us a theory of *redundancy elimination* that ignores or removes bindings of quantified variables

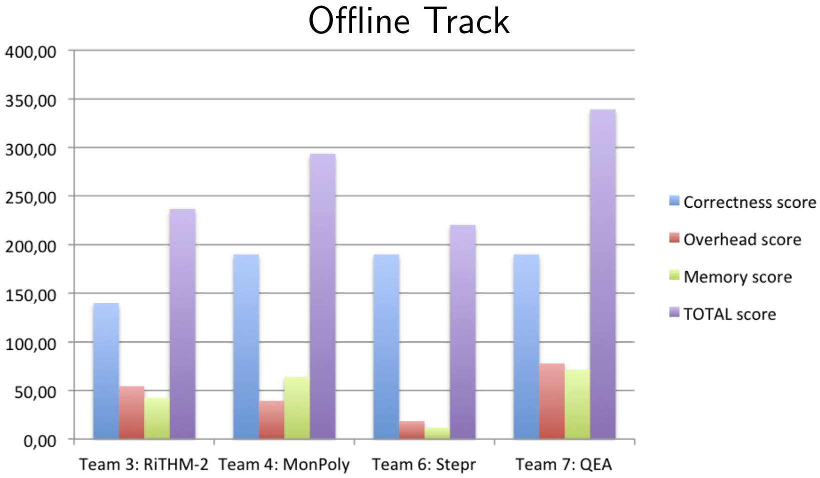
Structural specialisation

- Not all features of the QEA specification language are needed for every specification i.e. non-determinism, free variables
- However supporting these features makes the monitoring algorithm more complex

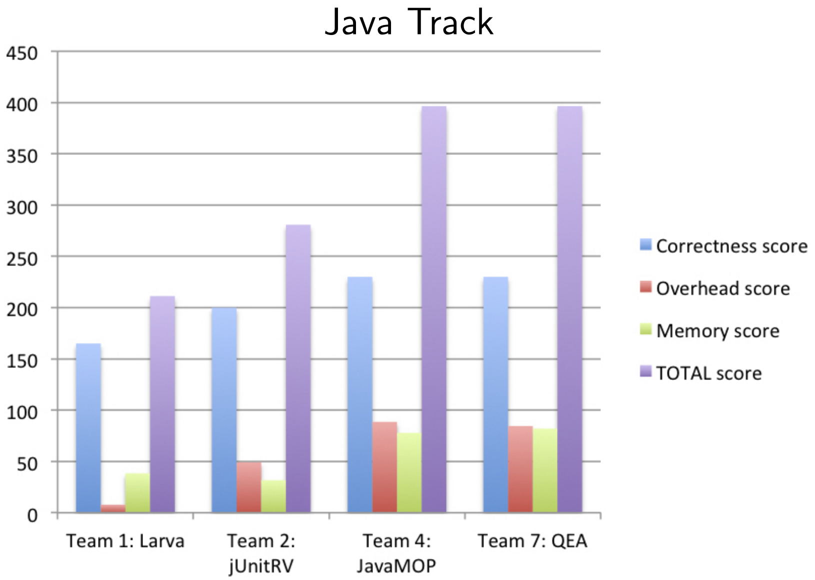
- Structural specialisation produces a monitoring algorithm based on the structure of the specification using special data structures

- Motivation: monitoring is the frequent repetition of the same small bit of code

Results of the 1st compeititon on Runtime Verification



Results of the 1st compeititon on Runtime Verification



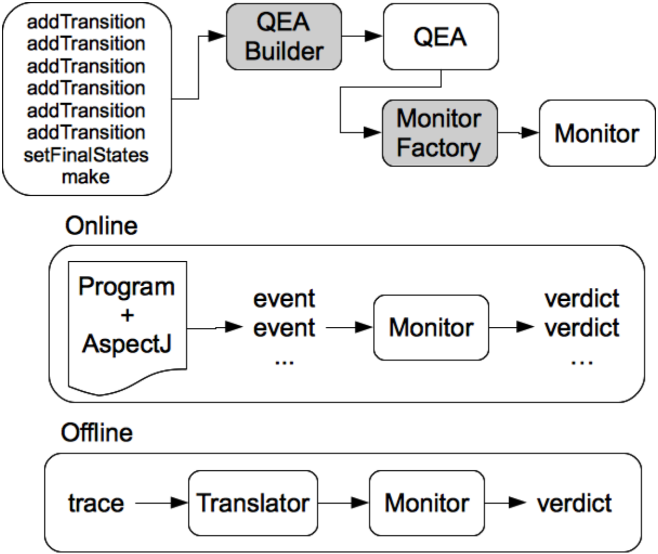
Writing specifications with efficiency in mind

- Efficiency of the monitoring algorithm depends on the *structure* of the specification
- Therefore the way the specification is written matters

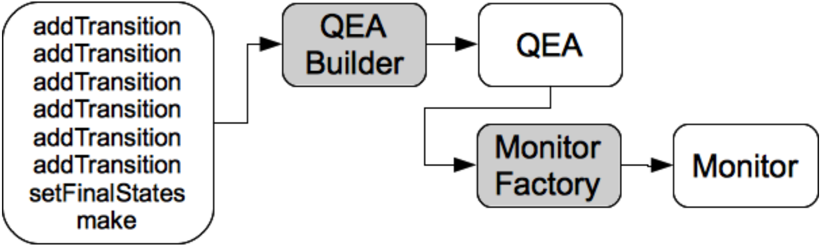
- Can achieve an order of magnitude speedup by changing the way the property is expressed

Outline

Separating specification and usage



Writing QEA specifications in MarQ



- The QEA Builder uses an API to construct a QEA
- The MonitorFactory constructs a monitor from that specification

The QEA Builder

```
QEABuilder q = new QEABuilder("safeiter");

int ITERATOR = 1; int NEXT = 2;
final int i = -1; final int size = 1;
q.addQuantification(FORALL, i)

q.addTransition(1,ITERATOR, i, size, 2);
q.addTransition(2,NEXT, i,
    isGreaterThanConstant(size,0),
    decrement(size),
    2);

q.addFinalStates(1, 2); q.setSkipStates(1);

QEA qea = q.make();
```

The QEA Builder

```
QEABuilder q = new QEABuilder("safeiter");

int ITERATOR = 1; int NEXT = 2;
final int i = -1; final int size = 1;
q.addQuantification(FORALL, i)

q.addTransition(1,ITERATOR, i, size, 2);
q.addTransition(2,NEXT, i,
    isGreaterThanConstant(size,0),
    decrement(size),
    2);

q.addFinalStates(1, 2); q.setSkipStates(1);

QEA qea = q.make();
```

The QEA Builder

```
QEABuilder q = new QEABuilder("safeiter");

int ITERATOR = 1; int NEXT = 2;
final int i = -1; final int size = 1;
q.addQuantification(FORALL, i)

q.addTransition(1,ITERATOR, i, size, 2);
q.addTransition(2,NEXT, i,
    isGreaterThanConstant(size,0),
    decrement(size),
    2);

q.addFinalStates(1, 2); q.setSkipStates(1);

QEA qea = q.make();
```

The QEA Builder

```
QEABuilder q = new QEABuilder("safeiter");

int ITERATOR = 1; int NEXT = 2;
final int i = -1; final int size = 1;
q.addQuantification(FORALL, i)

q.addTransition(1,ITERATOR, i, size, 2);
q.addTransition(2,NEXT, i,
    isGreaterThanConstant(size,0),
    decrement(size),
    2);

q.addFinalStates(1, 2); q.setSkipStates(1);

QEA qea = q.make();
```

The QEA Builder

```
QEABuilder q = new QEABuilder("safeiter");

int ITERATOR = 1; int NEXT = 2;
final int i = -1; final int size = 1;
q.addQuantification(FORALL, i)

q.addTransition(1,ITERATOR, i, size, 2);
q.addTransition(2,NEXT, i,
    isGreaterThanConstant(size,0),
    decrement(size),
    2);

q.addFinalStates(1, 2); q.setSkipStates(1);

QEA qea = q.make();
```

The QEA Builder

```
QEABuilder q = new QEABuilder("safeiter");

int ITERATOR = 1; int NEXT = 2;
final int i = -1; final int size = 1;
q.addQuantification(FORALL, i)

q.addTransition(1,ITERATOR, i, size, 2);
q.addTransition(2,NEXT, i,
    isGreaterThanConstant(size,0),
    decrement(size),
    2);

q.addFinalStates(1, 2); q.setSkipStates(1);

QEA qea = q.make();
```

The QEA Builder

```
QEABuilder q = new QEABuilder("safeiter");

int ITERATOR = 1; int NEXT = 2;
final int i = -1; final int size = 1;
q.addQuantification(FORALL, i)

q.addTransition(1,ITERATOR, i, size, 2);
q.addTransition(2,NEXT, i,
    isGreaterThanConstant(size,0),
    decrement(size),
    2);

q.addFinalStates(1, 2); q.setSkipStates(1);

QEA qea = q.make();
```

The QEA Builder

```
QEABuilder q = new QEABuilder("safeiter");

int ITERATOR = 1; int NEXT = 2;
final int i = -1; final int size = 1;
q.addQuantification(FORALL, i)

q.addTransition(1,ITERATOR, i, size, 2);
q.addTransition(2,NEXT, i,
    isGreaterThanConstant(size,0),
    decrement(size),
    2);

q.addFinalStates(1, 2); q.setSkipStates(1);

QEA qea = q.make();
```

The QEA Builder

```
QEABuilder q = new QEABuilder("safeiter");

int ITERATOR = 1; int NEXT = 2;
final int i = -1; final int size = 1;
q.addQuantification(FORALL, i)

q.addTransition(1,ITERATOR, i, size, 2);
q.addTransition(2,NEXT, i,
    isGreaterThanConstant(size,0),
    decrement(size),
    2);

q.addFinalStates(1, 2); q.setSkipStates(1);

QEA qea = q.make();
```

The QEA Builder

```
QEABuilder q = new QEABuilder("safeiter");

int ITERATOR = 1; int NEXT = 2;
final int i = -1; final int size = 1;
q.addQuantification(FORALL, i)

q.addTransition(1,ITERATOR, i, size, 2);
q.addTransition(2,NEXT, i,
    isGreaterThanConstant(size,0),
    decrement(size),
    2);

q.addFinalStates(1, 2); q.setSkipStates(1);

QEA qea = q.make();
```

The Monitor Factory

```
public interface Monitor{
    public Verdict step(int name, Object[] args);
}
```

```
Monitor monitor = MonitorFactory.create(
    qea,
    GarbageMode.LAZY,     //Optional
    RestartMode.REMOVE); // Optional
```

- A monitor is an object accepting events and producing verdicts
- The factory will analyse the specification and produce the best monitor it can (i.e. using *structural specialisation*)
- Optional modes
  - Garbage: handling monitored objects that are garbage collected
  - Restart: what to do when a violation occurs

The Monitor Factory

```
public interface Monitor{
    public Verdict step(int name, Object[] args);
}

Monitor monitor = MonitorFactory.create(
    qea,
    GarbageMode.LAZY,     //Optional
    RestartMode.REMOVE); // Optional
```

- A monitor is an object accepting events and producing verdicts
- The factory will analyse the specification and produce the best monitor it can (i.e. using *structural specialisation*)
- Optional modes
  - Garbage: handling monitored objects that are garbage collected
  - Restart: what to do when a violation occurs

The Monitor Factory

```
public interface Monitor{
    public Verdict step(int name, Object[] args);
}

Monitor monitor = MonitorFactory.create(
    qea,
    GarbageMode.LAZY,     //Optional
    RestartMode.REMOVE); // Optional
```

- A monitor is an object accepting events and producing verdicts
- The factory will analyse the specification and produce the best monitor it can (i.e. using *structural specialisation*)
- Optional modes
  - Garbage: handling monitored objects that are garbage collected
  - Restart: what to do when a violation occurs

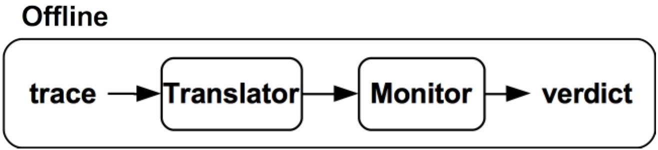
The Monitor Factory

```
public interface Monitor{
    public Verdict step(int name, Object[] args);
}

Monitor monitor = MonitorFactory.create(
    qea,
    GarbageMode.LAZY,     //Optional
    RestartMode.REMOVE); // Optional
```

- A monitor is an object accepting events and producing verdicts
- The factory will analyse the specification and produce the best monitor it can (i.e. using *structural specialisation*)
- Optional modes
  - Garbage: handling monitored objects that are garbage collected
  - Restart: what to do when a violation occurs

Offline monitoring with MarQ



- We first create a `Translator` object for the trace
- Then using a `QEA` object and trace string we create a monitor
- Finally we call `monitor` to get a result

Offline monitoring with MarQ

```
OfflineTranslator translator =
    new DefaultTranslator(``a'', ``b'',``c'');
String trace = ``trace_dir/trace.csv'';
QEA qea = builder.make();
CSVFileMonitor m =
    new CSVFileMonitor(trace_name, qea, translator);
Verdict v = m.monitor();
```

- We first create a `Translator` object for the trace
- Then using a `QEA` object and trace string we create a monitor
- Finally we call `monitor` to get a result

Offline monitoring with MarQ

```
OfflineTranslator translator =
    new DefaultTranslator(``a'', ``b'',``c'');
String trace = ``trace_dir/trace.csv'';
QEA qea = builder.make();
CSVFileMonitor m =
    new CSVFileMonitor(trace_name, qea, translator);
Verdict v = m.monitor();
```

- We first create a `Translator` object for the trace
- Then using a `QEA` object and trace string we create a monitor
- Finally we call `monitor` to get a result

Offline monitoring with MarQ

```
OfflineTranslator translator =
    new DefaultTranslator(``a'', ``b'',``c'');
String trace = ``trace_dir/trace.csv'';
QEA qea = builder.make();
CSVFileMonitor m =
    new CSVFileMonitor(trace_name, qea, translator);
Verdict v = m.monitor();
```

- We first create a `Translator` object for the trace
- Then using a `QEA` object and trace string we create a monitor
- Finally we call `monitor` to get a result

Offline monitoring with MarQ

```
OfflineTranslator translator =
    new DefaultTranslator(``a'', ``b'',``c'');
String trace = ``trace_dir/trace.csv'';
QEA qea = builder.make();
CSVFileMonitor m =
    new CSVFileMonitor(trace_name, qea, translator);
Verdict v = m.monitor();
```

- We first create a `Translator` object for the trace
- Then using a `QEA` object and trace string we create a monitor
- Finally we call `monitor` to get a result

Trace formats

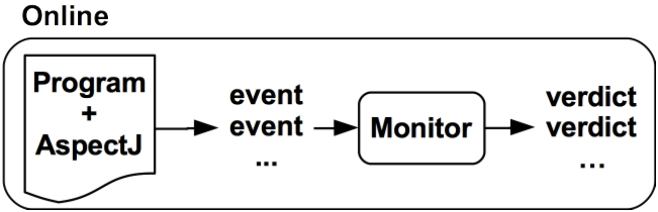
- We support three trace formats proposed by the RV competition
    - CSV
    - JSON
    - XML
- We also provide tools for translating between formats
- Parsing is surprisingly important.
    - MarQ has an optimised parser for the CSV format
    - Also aparallel option that separates trace processing and monitoring.
- Parsers produce abstract event objects handled by translators before being passed to the monitor

Translators

- Translators map abstract events to monitor `step` calls
- They tackle three pragmatic issues

1. Name mappings
   - Map between event names used in the trace and specification
   - May not be one-to-one
2. Reordering parameters
   - The events recorded in the trace may not have the same structure as in the specification
   - Parameters may be reordered or removed
3. Interpreting values
   - Data values in events may need to be treated semantically
   - i.e. parsing into integers
   - May need to ensure values are *interned*

Online monitoring with MarQ



- Currently MarQ supports monitoring `Java` programs via `AspectJ`
- Instrumentation should handle verdicts
- However, we do not yet automatically generate `AspectJ`
- This is work in process, along with targetting other languages

Instrumentation with AspectJ

```
public aspect SafeIterAspect {

  private int ITERATOR = 1; private int NEXT = 2;
  private Monitor monitor;

  SafeIterAspect(){
    QEA qea = SafeIter.get();
    monitor = MonitorFactory.create(qea);
  }

  ...
```

Instrumentation with AspectJ

```
public aspect SafeIterAspect {

  private int ITERATOR = 1; private int NEXT = 2;
  private Monitor monitor;

  SafeIterAspect(){
    QEA qea = SafeIter.get();
    monitor = MonitorFactory.create(qea);
  }

  ...
```

Instrumentation with AspectJ

```
pointcut iter(Collection c) :
  (call(Iterator Collection+.iterator()) && target(c));
pointcut next(Iterator i) :
  (call(* Iterator.next()) && target(i));

after (Collection c) returning (Iterator i) : iter(c) {
    synchronized(monitor){
      check(monitor.step(ITERATOR,i,c.size()));
    }
}
before(Iterator i) : next(i) {
    synchronized(monitor){
      check(monitor.step(NEXT,i));
    }
}
```

Instrumentation with AspectJ

```
pointcut iter(Collection c) :
  (call(Iterator Collection+.iterator()) && target(c));
pointcut next(Iterator i) :
  (call(* Iterator.next()) && target(i));

after (Collection c) returning (Iterator i) : iter(c) {
    synchronized(monitor){
      check(monitor.step(ITERATOR,i,c.size()));
    }
}
before(Iterator i) : next(i) {
    synchronized(monitor){
      check(monitor.step(NEXT,i));
    }
}
```

Instrumentation with AspectJ

```
pointcut iter(Collection c) :
  (call(Iterator Collection+.iterator()) && target(c));
pointcut next(Iterator i) :
  (call(* Iterator.next()) && target(i));

after (Collection c) returning (Iterator i) : iter(c) {
    synchronized(monitor){
      check(monitor.step(ITERATOR,i,c.size()));
    }
}
before(Iterator i) : next(i) {
    synchronized(monitor){
      check(monitor.step(NEXT,i));
    }
}
```

Instrumentation with AspectJ

```
pointcut iter(Collection c) :
  (call(Iterator Collection+.iterator()) && target(c));
pointcut next(Iterator i) :
  (call(* Iterator.next()) && target(i));

after (Collection c) returning (Iterator i) : iter(c) {
    synchronized(monitor){
      check(monitor.step(ITERATOR,i,c.size()));
    }
}
before(Iterator i) : next(i) {
    synchronized(monitor){
      check(monitor.step(NEXT,i));
    }
}
```

Instrumentation with AspectJ

```
pointcut iter(Collection c) :
  (call(Iterator Collection+.iterator()) && target(c));
pointcut next(Iterator i) :
  (call(* Iterator.next()) && target(i));

after (Collection c) returning (Iterator i) : iter(c) {
    synchronized(monitor){
      check(monitor.step(ITERATOR,i,c.size()));
    }
}
before(Iterator i) : next(i) {
    synchronized(monitor){
      check(monitor.step(NEXT,i));
    }
}
```

Instrumentation with AspectJ

```
 ...

 private void check(Verdict verdict){
     if(verdict==Verdict.FAILURE){ <report error here> }
 }
}
```

Future work

- External specification language
- Automated generation of instrumentation code
- Transformations from other Runtime Monitoring specification languages into QEA i.e. temporal logics, rule-based systems
- Automated transformations into syntactic classes with better efficiency guarantees
- Violation explanation

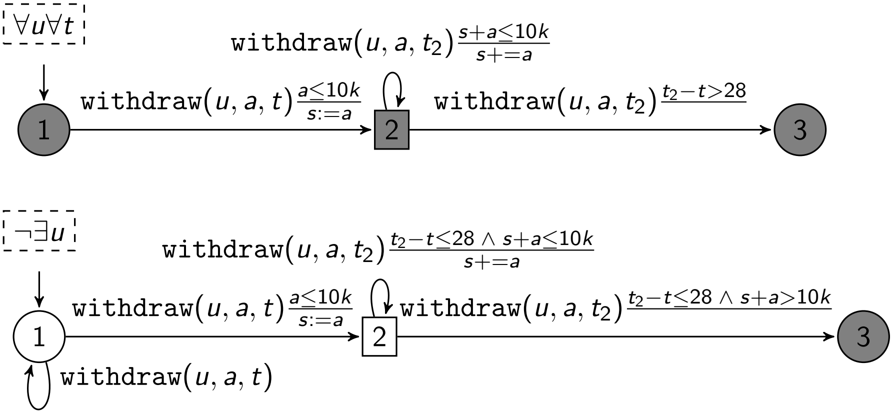
And finally...

- The tool is available online

  `https://github.com/selig/qea`

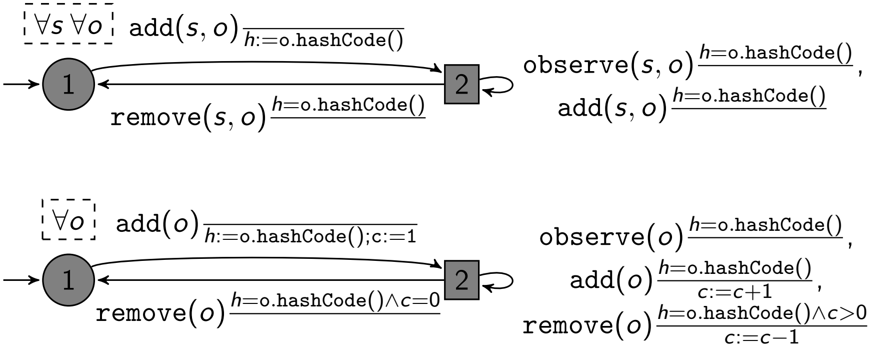
- Thanks for listening

Non-determinism



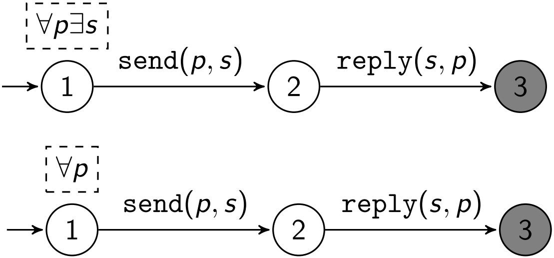
- By introducing non-determinism we can remove a quantifier
- The idea is to have a path through the automata for each t
- There is a trade-off between complexity in quantification and non-determinism

Counting



- By making certain assumptions about how trace slices for a quantifier are related we can track instances with a counter
- Transformation needs more information than is in the specification

Quantifier stripping



- Existential quantifiers on the right can be stripped
- This is a trivial case, more complex cases involve the introduction of guards and non-determinism

Big impact

Property	Trace length	Runtime (milliseconds)		Speedup
		Original	Translated	
withdrawal	150k	3,050	2,106	1.44
persistenthash	4M	12,267	864	14.12
publishers	200k	355	37	9.59

- Can achieve an order of magnitude speedup
- A phenomenon not often talked about in RV papers
- Further work: automate these transformations