MARQ: Monitoring At Runtime with QEA

Giles Reger, Helena Cuenca Cruz and David Rydeheard

University of Manchester

Abstract. Runtime monitoring is the process of checking whether an execution trace of a running system satisfies a given specification. For this to be effective, monitors which run trace-checking algorithms must be efficient so that they introduce minimal computational overhead. We present the MARQ tool for monitoring properties expressed as Quantified Event Automata. This formalism generalises previous automata-based specification methods. MARQ extends the established parametric trace slicing technique and incorporates existing techniques for indexing and garbage collection as well as a new technique for optimising runtime monitoring: *structural specialisations* where monitors are generated based on structural characteristics of the monitored property. MARQ recently came top in two tracks in the 1st international Runtime Verification competition, showing that MARQ is one of the most efficient existing monitoring tools for both offline monitoring of trace logs and online monitoring of running systems.

1 Introduction

Runtime monitoring [14, 17] is the process of checking whether an execution trace produced by a running system satisfies a given specification. Here we present MARQ, a new runtime monitoring tool that uses the QEA specification language. Over the past few years a number of runtime monitoring approaches have been developed [2, 4, 6, 9, 11, 18, 19] but there has been little comparison of the relative *efficiency* and *expressiveness* of runtime monitoring tools; mainly due to a lack of agreed benchmarks. This prompted the recently held 1st international Runtime Verification competition [5], where MARQ won the offline monitoring and online monitoring for `Java` tracks [1]. This paper makes use of specifications and benchmarks from this competition.
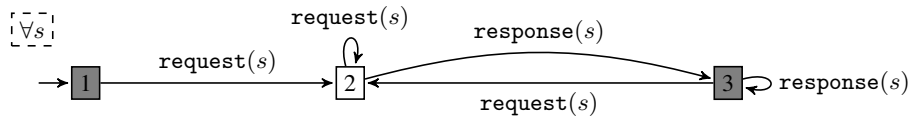
Runtime monitoring. Whilst techniques such as model checking are concerned with checking correctness against all possible runs of a system, runtime monitoring considers traces observed on individual runs of a system. Although incomplete, in the sense that only observed runs are checked, this approach has the advantage that actual behaviour is analysed. Scalability issues are then restricted to deciding which runs to analyse.

Typically runtime monitoring consists of three stages: firstly, a property denoting a set of valid traces is specified in a formal language. Secondly, the system of interest is *instrumented* to produce the required events recording information about the state of the system. Thirdly, a monitor is generated from the specification, which processes the trace to produce a verdict. This monitoring can occur *offline* on recorded executions or *online* whilst the monitored system is running. The offline case means that any system that produces logs can be monitored; whereas the online case requires specific mechanisms

for receiving events at runtime. The MARQ tool is suitable for offline monitoring and online monitoring of `Java` programs using `AspectJ` for instrumentation.

In both online and offline monitoring, *efficiency* is a key concern; monitoring a system online may introduce computational overheads and also interference with the monitored system. The aim is to produce monitors that minimise these effects. One advantage of online monitoring is that it can be included in production systems to guard against incorrect behaviour and potentially take corrective action. However, this becomes impractical if the monitoring tool introduces excessive overhead.

To illustrate the runtime monitoring process, consider a system where different services may be requested. A typical desired property is that every request should receive a response; in first-order Linear Temporal Logic this might be specified as $\forall s.\Box(\texttt{request}(s) \rightarrow \Diamond\texttt{response}(s))$. MARQ uses quantified event automata (QEA) [4, 21] to specify such properties as they admit an efficient monitoring algorithm via the *parametric trace slicing* approach discussed later. Fig. 1 gives a QEA for this property:

Fig. 1. A QEA describing a request-response property for services (see Sec. 2.1).

This uses an automaton to capture the allowed ordering of events and a quantification $\forall s$ to indicate the property should hold for all services s. The trace $\texttt{request}(A).$ $\texttt{request}(B).\texttt{response}(A)$ violates the property as service B has no response. As explained in Sec. 2.1, this verdict is computed by *slicing* the trace for values A and B and inspecting the subtraces separately.

This is an example of a *parametric* or *first-order* property as it deals with events containing data. QEA allows data to be treated in both a *quantified* and a *free* way (see Sec. 2). Runtime monitoring approaches for parametric properties tend to focus either on *efficiency* or *expressiveness*. Those focussing on efficiency typically employ the *parametric trace slicing* technique [8, 23] illustrated above.

Contribution. As shown previously [4, 21], QEA is more expressive than the underlying languages of comparably efficient monitoring tools i.e. those that make us of parametric trace slicing [2, 8, 18]. Therefore, MARQ can monitor properties that make use of these additional features e.g. existential quantification and free variables. As well as additional expressiveness, the MARQ tool implements a range of optimisations for efficient monitoring, including a novel technique which selects optimisations according to the structure of the specified property.

Availability. MARQ is available from

<div align="center">

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

</div>

This includes instructions on how to perform online and offline monitoring and a collection of specifications used in the 1st international runtime verification competition.

Structure. We first describe how MARQ can be used to specify properties (Sec. 2) and perform monitoring (Sec. 3). This is followed by an overview of MARQ's implementation (Sec. 4). Then we look at how changing the way a specification is written can improve monitoring efficiency (Sec. 5). We finish by comparing MARQ with other tools (Sec. 6) and giving conclusions (Sec. 7).

2 Writing specifications in QEA

We consider the problem of using QEA to write parametric specifications for MARQ. This presentation is example-led, omitting details which can be found in [4, 21].

We are concerned with parametric properties in which events may carry data. A *parametric event* is a pair of an event name and a list of data values, and a parametric trace is a finite sequence of parametric events. A parametric property denotes a set of parametric traces. Quantified event automata (QEA) determine such sets of traces - those accepted by the automata.

A QEA is a list of quantified variables together with an event automata. An event automata is a finite state machine with transitions labelled with symbol parametric events, where data values can be replaced by variables. Transitions may also include guards and assignments over these variables.

2.1 The slicing approach

Let us revisit the QEA in Fig. 1. The event automaton consists of three states and five transitions. The shaded states are final states. The square states are *closed to failure* i.e. if no transition can be taken there is a transition to an implicit failure state; the alternative, seen below, is to have a circular state that is *closed to self* i.e. if no transition can be taken there is an implicit self-looping transition. The quantifier list $\forall s$ means that the property must hold for *all* values that s takes in the trace i.e. the values obtained when matching the symbolic events in the specification with concrete events in the trace.

To decide the verdict given the trace

$$\texttt{request}(A).\texttt{request}(B).\texttt{request}(C).\texttt{response}(A).\texttt{request}(C).\texttt{response}(C)$$
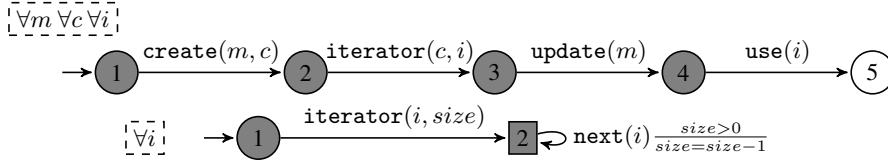
we *slice* the trace based on the values that can match s, giving the slices

$$
\begin{array}{rcl}
[s \mapsto A] & \mapsto & \texttt{request}(A).\texttt{response}(A) \\
[s \mapsto B] & \mapsto & \texttt{request}(B) \\
[s \mapsto C] & \mapsto & \texttt{request}(C).\texttt{request}(C).\texttt{response}(C)
\end{array}
$$

Then we ask whether each slice is accepted by the event automaton *instantiated* with the binding i.e. with s replaced by the appropriate value. The slice for $[s \mapsto B]$ does not reach a final state, therefore the whole trace is not accepted.

2.2 Two different kinds of variables

In QEA variables can either be quantified or free. This is an important distinction and we review the difference here. The variables of a QEA are those appearing in symbolic events labelling transitions. For example, the QEA in Fig. 1 has the single variable s.

$\forall m\ \forall c\ \forall i$

$$1 \xrightarrow{\texttt{create}(m,c)} 2 \xrightarrow{\texttt{iterator}(c,i)} 3 \xrightarrow{\texttt{update}(m)} 4 \xrightarrow{\texttt{use}(i)} 5$$

$\forall i$

$$1 \xrightarrow{\texttt{iterator}(i,size)} 2 \circlearrowright \texttt{next}(i)\ \frac{size>0}{size=size-1}$$

Fig. 2. Two QEAs demonstrating QEA features. **unsafemapiter** (top) specifies how iterators over collections created from maps should be used. **safeiter** (bottom) restricts the maximum number of calls to `next` on an iterator.

Quantified variables. A QEA may have zero or more variables that are universally or existentially quantified. Quantifications have their usual interpretation i.e. universal means *for all values in the domain* where the domain is determined by the trace (see above). A quantification may be associated with a guard (a predicate over previously quantified variables), which excludes some bindings of quantified variables from consideration. A quantification list may be negated, which inverts the verdict obtained.

As an example, the **unsafemapiter** property in Fig. 2 uses multiple quantified variables. The property is that an iterator over a collection created from a map cannot be used after the map is updated. The style of specification here is to specify the events that lead to failure i.e. for every map m, collection c and iterator i if we see `create`(m,c).`iterator`(c,i). `update`(m).`use`(i) then the property has been violated. Note that the circular states are closed to self (see above) so any additional events are ignored.
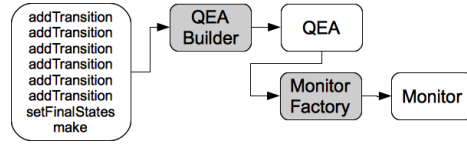
Free variables. Any variables in a QEA that are not quantified are called *free*. Quantified variables indicate that the automaton should be instantiated for each binding of those variables; free variables are local to each of these instantiations and are rebound as the trace is processed. The purpose of free variables is to store data that can be accessed by *guards* and *assignments* on transitions. Guards are predicates that restrict the availability of transitions and assignments can modify the values given to free variables.

The **safeiter** property in Fig. 2 uses a free variable $size$ to track the size of a collection being iterated over. The property is that the `next` method may only be called on an `Iterator` object as many times as their are objects in the base collection. This is a generalisation of the standard HasNext property often used in runtime monitoring [2, 18]. The value for $size$ is bound on the `iterator` event and then checked and updated in a loop on state 2; as this state is square if the guard is false there is an implicit failure.

2.3   Creating QEAs and Monitors in MARQ

To specify QEA as input to the MARQ system, we introduce a `QEABuilder` as illustrated in Fig. 3. A builder object is used to add transitions and record information about quantification and states. These are used to construct a QEA that is passed to the `MonitorFactory` as discussed below.

Fig. 4 shows how `QEABuilder` can be used to construct the **safeiter** property QEA seen in Fig. 2. First the builder object q is created. Then we declare the event names and variables as integers; quantified variables are negative, free variables are positive.

Fig. 3. Using the QEABuilder and MonitorFactory

```
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();
```

Fig. 4. Using QEABuilder to construct the **safeiter** QEA in Fig. 2

We then declare the universal quantification for i. The two transitions are added using `addTransition`. For the first transition we specify only the start state, event name, parameters and end state. For the second transition we include the guard and assignment. MARQ includes a library of guards and assignments. Additionally, it is possible for the user to define a new guard or assignment by implementing a simple interface. Currently MARQ supports the guards and assignments for dealing with equality and integer arithmetic. The last stage of defining the QEA is to specify the final (accepting) and the types of states, which can skip (circular) or next (square) as explained above.

Once we have constructed a QEA we can create a monitor object by a call to the `MonitorFactory`. This will inspect the structure of the QEA and produce an optimised monitor as described in Sec. 4.3. Optionally, we can also specify garbage and restart modes on monitor creation.

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

Garbage mode determines how garbage objects should be treated. As explained in Sec. 4.2 it is sometimes possible to remove bindings related to *garbage* objects in online monitoring. The default is to not collect garbage as this allows us to use more streamlined data structures and is applicable to the offline case. Restart mode determines what happens when an ultimate verdict is detected e.g. violation of a safety property. There is the option to remove or rollback the status of the offending binding. Both achieve a *signal-and-continue* approach used by other systems. The default is not to restart.

3 Running MARQ online and offline

Here we demonstrate how our tool can be used for offline and online monitoring. Fig. 5 illustrates the two different monitoring settings. In the first case, offline monitoring, a
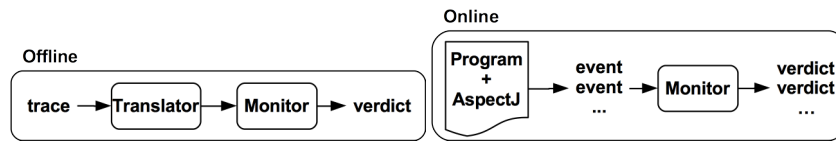
Fig. 5. Two different monitoring modes.

trace is given as a log file and processed by a translator and the monitor to produce a verdict. In the second case, online monitoring, a program is instrumented to emit events which are consumed by the monitor. The monitor then produces a verdict on each event. In the following we discuss how MARQ can be used in both settings. For complete examples of how to run MARQ in either mode see the website.

3.1 Offline monitoring

Offline monitoring can be performed in five lines of code that will take a trace and a translator to produce a verdict. MARQ can process traces in two different formats: CSV and XML. Alternatively, a custom parser for a new format could call the monitor directly as is done in online mode (see below).

The translator converts the string version of each event to the format used by the monitor. The default translator maps a list of strings to successive integers. If a different mapping is required, or it is necessary to ignore some parameters, a simple interface can be implemented to achieve this.

To monitor a trace we simply create the component parts, construct an appropriate `FileMonitor` (which reads in the trace) and call `monitor()` to produce a verdict.

```
String trace = ``trace_dir/trace.csv'';
QEA qea = builder.make(); // see Section 2
OfflineTranslator translator = new DefaultTranslator(``iterator'', ``next'');
CSVFileMonitor m = new CSVFileMonitor(trace_name, qea, translator);
Verdict v = m.monitor();
```

Calling `getStatus()` on the monitor after monitoring will print the final status of the monitor, giving the configurations associated with each binding of quantified variables.

3.2 Online monitoring

For online monitoring it is necessary to submit each event to the monitor at the time it is generated by the running system. Here we show how this can be done using `AspectJ` [16] where *aspects* define code that is to be *weaved* into the object at specified points. Weaving can occur at compile or load time, both are useful for runtime monitoring.

Fig. 6 gives an example aspect to be used to monitor the **safeiter** property from Fig. 2. Firstly we specify the event names, ensuring they are the same as those used in the QEA definition. We then create the monitor as described in the previous section. Two pointcuts are used to relate the events of the QEA to concrete `Java` calls. Note how we call `size` on the base collection of the iterator call to provide this information in the event. Finally, we check the verdict returned by each `step` call for failure. It

```
public aspect SafeIterAspect {

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

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

  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)); }
  }

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

Fig. 6. `AspectJ` for monitoring the **safeiter** property.

should be noted that MARQ is *not thread-safe*. We therefore synchronize on the monitor object for each call as we might be monitoring a concurrent program. One abstract event in the specification may relate to many different concrete events produced by the instrumentation, and vice versa. For example, in the **unsafemapiter** property in Fig. 2 we would relate the `create` event with the `values` and `keySet` methods from the `Map` interface. In the **withdrawal** specification given later (Fig. 8) there are different abstract events that would match with the same concrete event.

4   Efficient Monitoring

Details of the monitoring algorithm used in MARQ can be found in [21]. Here we highlight the major optimisations that ensure efficiency.

4.1   Indexing.

Monitoring requires information to be attached to bindings of quantified variables. When an event occurs in the trace, we need to find the *relevant bindings* and update the information attached to them. The collection of bindings generated by runtime monitoring can be very large and so efficient techniques to identify relevant bindings are necessary. These often involve indexing.

Purandare et al. [20] discuss three different kinds of indexing that use different parts of a parametric event to lookup the monitor state *relevant* to that event. The first two - *value-based*, using the values of an event, and *state-based*, using the states of the underlying propositional monitor - are used by JavaMOP [18] and tracematches [2] respectively. MARQ implements the third, symbol-based. When a binding is created it is used to partially instantiate the alphabet of the automaton and each partially instantiated

event is added to a map associating it with the binding. An observed concrete event is then *matched* against these partially instantiated events to locate the relevant bindings.

4.2 Garbage and redundancy.

Early work [3] showed that removing monitored objects that are no longer used in the monitored system can prevent memory leaks and improve performance. For example, in the case of the **safeiter** property, if the iterator object is garbage-collected the property cannot be violated for that iterator and the binding can be deleted. This can have a significant impact as millions of monitored short-lived objects can be generated in a typical run of a monitored system.

This idea belongs to a collection of optimisations for *redundancy elimination* - information about monitored objects can be safely omitted if it does not effect the outcome of the monitoring process. MARQ supports garbage collection in the same way as [15] i.e. when monitored objects are garbage collected it is checked whether the bindings they participate in can be removed. MARQ also implements a form of redundancy elimination that generalises the concept of *enable sets* [18]. Based on the automaton it is possible to conservatively precompute a reachability relationship that indicates whether recording a new binding will make an observable difference. This relationship is used to decide whether to create a new binding or not.

A further form of redundancy elimination is that of early detection of success or failure by identifying whether a certain verdict is *reachable* [7]. MARQ will return a verdict as soon as it is guaranteed that the verdict cannot change given any future events. To enable this the true and false verdicts are split into *weak* and *strong* variants.

4.3 Structural specialisations

Many common forms of specification use only a subset of available features i.e. conform to a particular structural restriction. Properties can be categorised according to their structural characteristics and a custom monitor for each category can be built. At first it might seem that the improvements will be insignificant. However, most monitoring activities consist of the repeated execution of a small number of operations. Therefore, when processing large event traces the small improvements accumulate, resulting in a significant reduction in time overhead.

Specialisations. As detailed in [10], we implement a number of specialisations of the monitoring algorithm that make assumptions about the structure of the monitored QEA. The first assumption we make is about the **number of quantified variables**. If we assume that a single quantified variable is used we can directly index on this value. Currently specialisations are restricted to this setting.

The remaining specialisations simplify the monitoring algorithm by removing checks and replacing complex data structures with simpler versions. The structural assumptions are as follows:

- **Use of free variables:** if free variables are not used then the structures for storing these, and support for guards and assignments, can be removed.
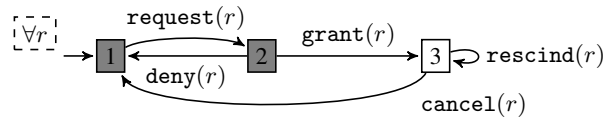
Fig. 7. A QEA giving the lifecycle of a resource object.

– **Determinism/non-determinism:** when the monitored QEA is deterministic we only need to track a single state or configuration per binding.
– **Fixed quantified variable:** if the unique quantified variable (recall we only consider one) always appears in a fixed position the matching process is simplified.

Whilst currently restricted to a single quantification, these specialisations cover a large number of useful cases and commonly occurring specification patterns. Future work will look at extending these to multiple quantifications.

Example. Let us illustrate this approach using the **resourcelife** property given as a QEA in Fig. 7. This specifies the possible actions over a resource: it can be requested and then either denied or granted, and when granted it can be rescinded before it is cancelled. We randomly construct a valid trace of 1M events using 5k resources and measure monitoring time for different monitor specialisations.
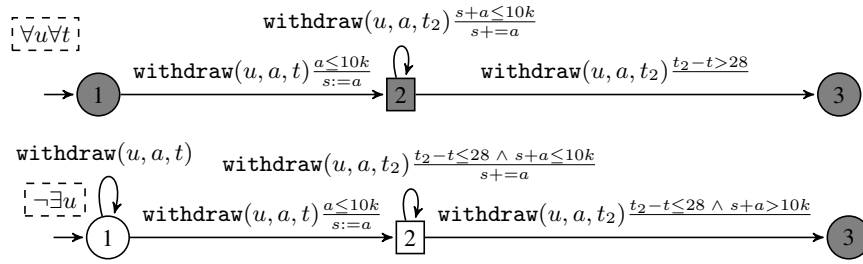
Monitoring this trace with the naive incremental monitoring algorithm described in [4] takes 96k ms. By noticing that there is a single quantified variable and directly indexing on this we can reduce this to 202 ms, which is $477 \times$ faster. We expect this large speedup as we have gone from a case without indexing to one using indexing. Removing support for non-determinism we can reduce this further to 172 ms, $1.2 \times$ faster than the previous monitor. Removing support for free variables reduces this to 106 ms, $1.6 \times$ faster than the previous monitor. Overall we achieve a $913 \times$ speedup, and ignoring the vast speedup from moving to indexing we still achieve a $1.9 \times$ speedup.

5 Writing specifications for efficient monitoring

Many different QEAs can specify the same property. However, choosing the right specification can determine the efficiency of the monitoring process.

The time complexity of MARQ's monitoring algorithm is dependent on characteristics of the trace (length, distribution of events) and of the specification (types of variables and transitions). One of the main factors is the number of quantified variables used in the specification. If there are n quantified variables there are a maximum of $\prod_i^n d_i$ bindings of quantified variables where d_i is the size of the domain of the ith quantified variable; this is exponential in n. Redundancy elimination can reduce this dramatically, but if it is possible to rewrite the specification to eliminate a quantified variable it can dramatically improve the performance of monitoring.

Here we discuss optimisations that eliminate quantified variables. In the future we plan to explore automatically simplifying a QEA to improve monitoring efficiency.

Fig. 8. Two QEAs for the property that a user must withdraw less than \$10k in a 28 day period.

5.1 Introducing non-determinism

Consider the **withdrawal** property that states that a user does not withdraw more than \$10k from an account in a 28 day period. Fig. 8 gives two equivalent QEAs for this property. The first QEA quantifies over a user u and a time point t that reflects the beginning of a 28-day period. The second QEA introduces non-determinism on the first state to remove the quantification over t. Whenever a `withdraw` event occurs this non-determinism records the start of a new 28-day period by making a transition to state 2 but also taking a transition to state 1, allowing another period to start on the next event.

To make the translation it was necessary to invert the automaton and extend the guards on transitions out of state 2. Adding the negation means that the event automaton now accepts incorrect, rather than correct traces. State 3 in the new QEA is the implicit failure state from the first QEA - the guard $t_2 - t \leq 28 \wedge s + a > 10k$ is the negation of the conjunction of guards labelling transitions out of state 2 in the old QEA. The $t_2 - t \leq 28$ conjunct in the guard of the looping transition on state 2 ensures that we discard the information for a time period when it exceeds 28 days; this is not necessary for correctness but important for efficiency.

Even though support for non-determinism adds some overhead (see Sec. 4.3), this is negligible in comparison to the savings made by removing bindings.

5.2 Introducing counters

The next example we consider is the **persistenthash** property that states that the `hashCode` of an object should remain the same whilst it is inside a structure that uses hashing e.g. a `HashMap` or `HashSet`. Fig. 9 gives two equivalent QEAs for this property. The first QEA quantifies over the structure. To remove this quantification, the second QEA introduces a counter to track how many structures the object is inside.

5.3 Stripping existential quantification

Finally, we consider the **publishers** property that every publisher p that sends messages to subscribers s gets at least one reply. Fig. 10 gives two equivalent QEAs for this property. The second strips the tailing existential quantification, making the variable s a free variable. This has the same effect as any subscriber that led to a trace being accepted by the first QEA would cause the trace to be accepted by the second.
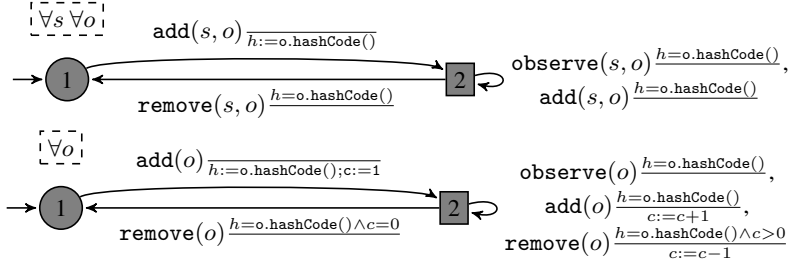
Fig. 9. Two equivalent QEAs for the persistence of hashes for objects in hashing structures.
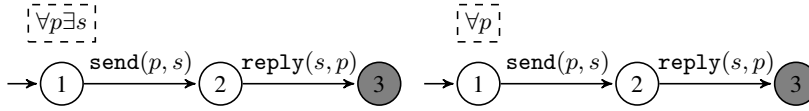

Fig. 10. Two equivalent QEAs for the property that every publisher has a subscriber.

5.4 Performance improvements

We briefly demonstrate that these translations achieve a significant performance improvement. Table 5.4 shows that this translation can speed monitoring up by an order of magnitude, demonstrating that the way in which a property is specified can have a dramatic impact on efficiency. It should be noted that the exponential dependence on the number of quantified variables is common to all tools that use parametric trace slicing.

Table 1. Performance improvements for translated QEAs

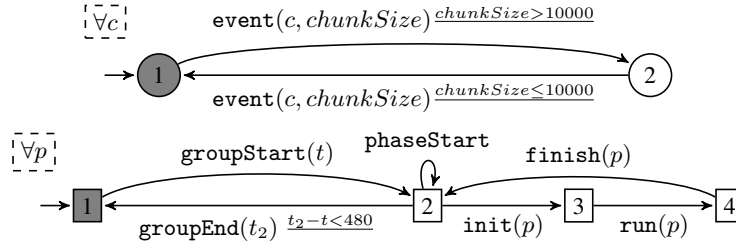| Property	Trace length	Runtime (milliseconds)		Speedup
		Original	Translated	
withdrawal	150k	3,050	2,106	1.44
persistenthash	4M	12,267	864	14.12
publishers	200k	355	37	9.59

6 Comparative evaluation

In this section we compare MARQ with other runtime monitoring tools. We make use of benchmarks taken from the 1st international Runtime Verification competition [5]. We consider the performance of tools on selected benchmarks and reflect on the specification languages used by each tool. We also report the results of the competition [1].

6.1 Offline monitoring

We evaluate MARQ for its use in offline monitoring i.e. the processing of recorded logs. We consider four properties, two previously introduced, and the two given in Fig. 11.

Table 2. Selected timing results for offline monitoring (in seconds).

Property	Trace length	RiTHM2	MonPoly	STePr	MARQ	Speedup	
						Min	Avg.
maxchunk (Fig. 11)	1.4M	**0.59**	8.4	8.86	3.58	0.16	**1.66**
withdrawal (Fig. 8)	315k	-	**1.53**	3.67	2.57	0.6	**1.01**
processes (Fig. 11)	823k	2.39	2.0	2.91	**0.63**	3.17	**3.86**
resourcelife (Fig. 7)	10M	5.18	3405	9.96	**2.04**	2.54	**558.8**
Competition score		236.91	293.54	220.40	**339.15**		

Fig. 11. Two QEAs used for evaluation. The first ensures that the chunk size eventually drops below 10k. The second captures a complex property about processes running in groups and phases.

We compare against three other competing tools in the offline track of the competition. MonPoly [6] monitors properties in Metric First-Order Temporal Logic and is designed to search for violating instantiations of a property. STePr is based on the LOLA language [11], which is a functional stream computation language. RiTHM2 [19] monitors properties in LTL and is designed with real-time systems in mind.

Table 2 gives the monitoring runtime with minimum and average speedup using MARQ. Firstly, note that these benchmarks involve very large traces, in some cases with millions of events. RiTHM2 cannot express the **withdrawal** property. MARQ always performs better on average. In the case of **maxchunk**, which has a very simple structure, the RiTHM2 tool performs the best. MonPoly struggled with the liveness elements of the **resourcelife** property.

The first three properties in Table. 2 were supplied by the teams behind RiTHM2, MonPoly and STePr respectively. We compare how these properties are specified in their native language with how they are specified in QEA. However, note that QEA graphical models must currently be represented using `Java` code as shown in Fig. 4.

In the RiTHM2 tool the **maxchunk** property is specified as

For all Connections, $\Box((\text{Connection.Chunksize} > 10000) \Rightarrow \Diamond(\text{Connection.Chunksize} \leq 10000))$

which is very similar to the QEA specification given in Fig. 11. Response properties of this kind are common specification patterns that most languages handle easily.

MonPoly specifies the **withdrawal** property as

ALWAYS FORALL s, u.
$(s \leftarrow \text{SUM } a; u \text{ ONCE}[0, 28] \text{ withdraw}(u, a) \text{ AND tp}(i)) \text{ IMPLIES } s \leq 10000$

which makes use of SUM and ONCE to capture the property concisely. The SUM aggregate operator takes the sum of values for a for a given u over a specified period and ONCE[0,28] defines the 28 day window. It should be noted that MonPoly cannot handle true liveness, only finitely-bounded intervals. It deals with this by putting a very large bound in the place to simulate infinity.

STePr specifies the **processes** property as

```
G(
        groupStart ⇒ WY(¬groupStart WS groupEnd)
    ∧ groupEnd ⇒ Y(¬groupEnd S groupStart)
    ∧ phaseStart ⇒ ¬groupEnd S groupStart)
    ∧ phaseStart ⇒ ¬(init(x) ∨ run(x)) WS finish(x)
    ∧ run(x) ⇒ Y(¬run(x) S init(x))
    ∧ finish(x) ⇒ Y(¬finish(x) S run(x))
    ∧ init(x) ⇒ WY(¬(init(x) ∨ run(x)) WS finish(x))
    ∧ (¬groupStart WS groupEnd) ⇒ ¬finish(x) ∧ ¬init(x) ∧ ¬run(x)
    ∧ groupEnd ∧ (¬groupStart S (groupStart ∧ time = x)) ⇒ time − x < 480000
)
```

which is more complex than the QEA given in Fig. 11. MonPoly requires a similarly complicated formalisation as both tools use temporal logic where each subproperty must be specified separately; whereas QEA can capture the intuition of the property.

Table 2 also gives the scores from the competition (see `http://rv2014.imag.fr/monitoring-competition/results` for a breakdown). This shows that MARQ outperformed the other tools in this competition.

6.2 Online monitoring

We consider MARQ's use in online monitoring for `Java`. We report the competition results and compare specification languages.

The other tools competing in this track of the competition were as follows. Larva [9] monitors properties specified as Dynamic Automata with Timers and Events. JavaMOP [18], like MARQ, is based on the parametric trace slicing approach. Both Larva and JavaMOP automatically generate `AspectJ` code. JUnitRV [12] extends the JUnit framework to perform monitoring where events are defined via a reflection library; they also include the monitoring modulo theories approach [13].

Table 3. Breakdown of results in CSRV14 online `Java` track (higher is better).

Tool	Correctness	Overhead	Memory	Total
Larva	165	7.79	38.43	211.22
JUnitRV	200	49.15	31.67	280.82
JavaMOP	230	88.56	77.89	396.45
MARQ	230	84.5	82.01	**396.51**

Table 3 gives a breakdown of the results from the CSRV14 competition. The correctness score reflects the properties that each tool was able to capture. On these benchmarks, Larva and JUnitRV struggled with expressiveness and running time (overhead). The results for MARQ and JavaMOP are similar, with JavaMOP running slightly faster

and MARQ consuming slightly less memory. Although the scores put MARQ just ahead of JavaMOP the authors would argue that, given the variability in results, this shows that the tools are equally matched on the kinds of benchmarks used in the competition.

Both Larva and JavaMOP divide specifications into a part that defines events and a part that declares a property over these events. For each tool events are specified as `AspectJ` pointcuts. JUnitRV uses a reflection library to capture events.

Larva supports free variables in its language, making the specification of **safeiter** very similar to that of QEA. However, it has limited support for multiple quantifications meaning that to capture the **unsafemapiterator** property Larva uses a free variable to track the iterators created from each collection; although this could be seen as an optimisation. JavaMOP provides multiple *plugin* languages for giving properties over events. This means that the **unsafemapiterator** property can be specified using a regular expression as follows:

<div align="center">ere : createColl updateMap* createIter useIter* updateMap updateMap* useIter</div>

The JavaMOP language has no native support for free variables, requiring programming in the `AspectJ` part of the language to capture properties such as **safeiter** and **persistenthash**. Both Larva and JavaMOP lack a natural way to relate a concrete event to multiple abstract events. JUnitRV can use at least future time LTL or explicit Mealy Machines but a description of the language is not available to the authors.

7 Conclusion

We have introduced the MARQ tool for runtime monitoring with QEA. We have shown how to use MARQ in both online and offline monitoring. Efficiency of the tool is discussed at length, both how to produce efficient QEA specification and how the tool performs relative to others that are available.

MARQ is an ongoing project and there is much work to be done. Firstly, we need to introduce an external language for specifying QEAs. Secondly, we aim to extend the notion of structural specialisations: considering properties with multiple quantifications and automatically translating properties to remove features where possible. It will be possible to use MARQ to continue research in specification mining for QEA [22].

References

1. `http://rv2014.imag.fr/monitoring-competition/results`.
2. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40:345–364, October 2005.
3. P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. *SIGPLAN Not.*, 42(10):589–608, Oct. 2007.
4. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, pages 68–84, 2012.
5. E. Bartocci, B. Bonakdarpour, and Y. Falcone. First international competition on software for runtime verification. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 1–9, 2014.

6. D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu. Monpoly: Monitoring usage-control policies. In S. Khurshid and K. Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 360–364. Springer Berlin Heidelberg, 2012.

7. A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proceedings of the 7th international conference on Runtime verification*, RV'07, pages 126–138, Berlin, Heidelberg, 2007. Springer-Verlag.

8. F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261, 2009.

9. C. Colombo, G. J. Pace, and G. Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 33–37, Washington, DC, USA, 2009. IEEE Computer Society.

10. H. Cuenca Cruz. Optimisation techniques for runtime verification. Master's thesis, University of Manchester, 2014.

11. B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: Runtime monitoring of synchronous systems. *2013 20th International Symposium on Temporal Representation and Reasoning*, 0:166–174, 2005.

12. N. Decker, M. Leucker, and D. Thoma. junitrv–adding runtime verification to junit. In G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 459–464. Springer Berlin Heidelberg, 2013.

13. N. Decker, M. Leucker, and D. Thoma. Monitoring modulo theories. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014.*, pages 341–356, 2014.

14. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In M. Broy and D. Peled, editors, *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems, to appear*. IOS Press, 2013.

15. D. Jin, P. O. Meredith, D. Griffith, and G. Rosu. Garbage collection for monitoring parametric properties. *SIGPLAN Not.*, 46(6):415–424, June 2011.

16. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

17. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, may/june 2008.

18. P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the mop runtime verification framework. *J Software Tools for Technology Transfer*, pages 1–41, 2011.

19. S. Navabpour, Y. Joshi, W. Wu, S. Berkovich, R. Medhat, B. Bonakdarpour, and S. Fischmeister. Rithm: A tool for enabling time-triggered runtime verification for c programs. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 603–606, New York, NY, USA, 2013. ACM.

20. R. Purandare, M. B. Dwyer, and S. Elbaum. Monitoring finite state properties: Algorithmic approaches and their relative strengths. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 381–395, Berlin, Heidelberg, 2012. Springer-Verlag.

21. G. Reger. *Automata Based Monitoring and Mining of Execution Traces*. PhD thesis, University of Manchester, 2014.

22. G. Reger, H. Barringer, and D. Rydeheard. A pattern-based approach to parametric specification mining. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, November 2013. To appear.

23. G. Roşu and F. Chen. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science*, 8(1):1–47, 2012. Short version presented at TACAS 2009.