An Overview of MarQ

Giles Reger

University of Manchester, Manchester, UK

Abstract. MarQ is a runtime monitoring tool for specifications written as quantified event automata, an expressive automata-based specification language based on the notion of parametric trace slicing. MarQ has performed well in the runtime verification competition and implements advanced indexing and redundancy elimination techniques. This overview describes the basic structure and functionality provided by MarQ and gives a brief description of how to use the tool.

1 Introduction

Runtime monitoring [3, 7] is the process of checking whether an execution trace produced by a running system satisfies a given specification. This paper gives an overview of the MarQ tool [12] for monitoring specifications written as quantified event automata (QEA) [1, 9, 6]. QEA is an expressive formalism for *parametric* properties i.e. those concerned with events parameterised by data.

MarQ is available from

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

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

This overview briefly describes the QEA formalism (Sec. 2), how to write and use these to monitor log files and Java programs using MarQ (Sec. 3) and its performance (Sec. 4). It concludes with remarks about its future (Sec. 5).

2 Quantified Event Automata

Quantified event automata [1] combine a logical notion of quantification with a form of extended finite state machine. To demonstrate the expressiveness of this formalism, Figure 1 gives three (simple) example QEA specifications for the following properties:

1. *SafeIterator.* An iterator created from a collection of size $size$ should only be iterated at most $size$ times.
2. *SafeMapIterator.* There should not be a map m, collection c and iterator i such that c is created from m, i is created from c, m is updated and then i is used. This demonstrates the use of multiple quantifiers.
3. *PublisherSubscriber.* For every publisher there exists a subscriber that receives all of that publisher's messages. This demonstrates how alternating quantification can be used to concisely capture a complex property about related objects.

See related publications [1, 9, 6] for further examples and a full description of their semantics. Note that QEA have a (may valued) finite-trace semantics so liveness properties (like *PublisherSubscriber*) are implicitly bounded by an end of trace event.

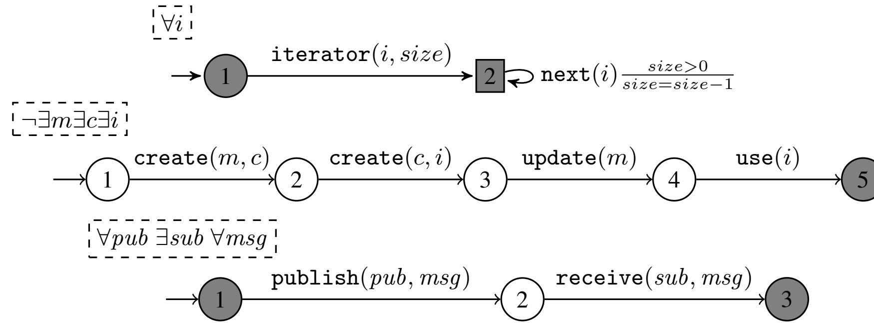

Fig. 1. Example quantified event automata.

3 Using MARQ

Here we briefly describe how to use MARQ. These examples (and others) are available online. We describe how to construct QEAs and their corresponding monitor objects and then how to use these objects to monitor log files and Java programs.

3.1 Creating QEAs and monitors

Currently MARQ provides a builder API for constructing QEA properties. Event names are specified as integers and there is a library of predefined guards and assignments that can be used in transitions. Below is an example of how the *SafeIterator* QEA can be constructed in this way. Sec. 5 discusses future plans to improve this.

```
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();
```

Here there are two event names (which must be consecutive positive integers starting from 1) and two variables, the quantified variable i (which must be a negative integer) and the free variable $size$ (which must be a positive integer). Two states are used (again positive integers) with 1 being the implicit start state.

Once we have constructed a QEA we create a monitor object by a call to the `MonitorFactory`. This will inspect the structure of the QEA and produce an optimised monitor object. Optionally, we can also specify garbage and restart modes on monitor creation (some of these are still experimental).

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

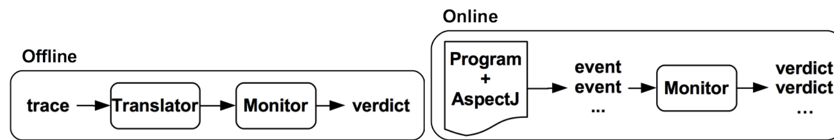

Fig. 2. Two different monitoring modes.

The garbage mode indicates how the monitor should handle references to monitored objects e.g. should weak references be used. By default the garbage mode is off, which is optimal for offline monitoring. The restart mode tells the monitor what should be done with a binding that fails the specification. For example, the `REMOVE` value here allows a *signal-and-continue* approach to monitoring safety properties.

3.2 Monitoring a trace offline

To monitor a trace we construct an appropriate `FileMonitor` (which reads in the trace) and call `monitor()` to produce a verdict. As illustrated in Fig. 2, offline monitoring of traces makes use of an optional Translator object to produce events in the form expected by the monitor constructed above. This allows parameters to be parsed as integers, reordered or filtered.

MARQ accepts trace files in the formats specified by the runtime verification competition [4]. Therefore, any system that can be intrusmented to produce such traces can be monitored offline. The following code can be used to construct a monitor for a CSV trace for the *SafeIterator* property. The translator object will parse the size parameter as an integer and other parameters as (interned) strings (objects with a notion of equality).

```
String trace = ``trace_dir/trace.csv'';
QEA qea = builder.make(); //see above
OfflineTranslator translator = TranslatorFactory.makeParsingTranslator(
                               event("iterator",param(0,OBJ),param(1,INT)),
                               event("next",param(0,OBJ)));
CSVFileMonitor m = new CSVFileMonitor(trace_name, qea, translator);
Verdict v = m.monitor();
```

3.3 Monitoring online via AspectJ

For monitoring Java programs MARQ is designed to be used with ASPECTJ i.e. using a *pointcut* for each event and submitting the necessary information directly to the monitor object as in the following extract. For other examples of how instrumentation and monitoring using ASPECTJ can be achieved see the online examples and [12].

```
after (Collection c) returning (Iterator i) :
    call(Iterator Collection+.iterator()) && target(c) {
    synchronized(monitor){ check(monitor.step(ITERATOR,i,c.size())); }
}
before(Iterator i) : call(* Iterator.next()) && target(i) {
    synchronized(monitor){ check(monitor.step(NEXT,i)); }
}
private void check(Verdict verdict){
    if(verdict==Verdict.FAILURE){ <report error here> }
}
```

4 Performance

We briefly discuss the performance of MarQ, see [9, 12] for experiments.

Implementation. MarQ has a number of features related to efficiency:

- *Structural specialisation.* MarQ analyses the QEA and constructs a monitoring algorithm suited to its structure. For example, particular indexing mechanisms can be employed. This is an ongoing area of research.
- *Symbol-based indexing.* Whilst other tools for parametric trace slicing use value-based indexing to lookup monitoring state, MarQ uses a symbol-based technique inspired by discrimination trees from automated reasoning.
- *Redundancy elimination.* MarQ analyses the QEA to determine which states are *redundant* and eagerly discards redundant information during monitoring.
- *Garbage removal.* As mentioned earlier, MarQ can be configured to weakly reference monitored objects and remove these from indexing structures when they become garbage. It is an ongoing area of research to extend these ideas to offline monitoring.

See [12] for further details.

Competitions. MarQ performed well in the 2014, 2015 and 2016 iterations of the runtime verification competition. It came joint first in the Java division in 2014[1] with JavaMOP [8] and in 2015[2] and 2016 [13] it came second to Mufin [2] (which is very efficient on certain forms of *connected* properties). In 2014 and 2016 it came first in the Offline division and in 2015 it came second to LogFire [5] (although performed better on benchmarks jointly entered).

5 Conclusion

MarQ is an efficient tool for parametric runtime verification of QEA. The development of MarQ is an ongoing project and the tool will continue to be updated and improved. The current planned areas for improvement are as follows:

- Improve the current method for defining QEA to remove the dependency on arbitrary details such as quantified variables being negative integers. Furthermore, providing a more general purpose method for defining guards and assignments rather than the current pre-defined library.
- Implement alternative front-end specification languages that compile into QEA. For example, a form of first-order temporal logic [14].
- Incorporate methods for explaining violations in terms of edits to the trace [10].
- Explore integration with specification mining techniques [11].

Please contact the author with comments or suggestions.

[1] See http://rv2014.imag.fr/monitoring-competition/results.html
[2] See https://www.cost-arvi.eu/?page_id=664

References

1. 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.
2. N. Decker, J. Harder, T. Scheffel, M. Schmitz, and D. Thoma. Tacas 2016. pages 868–884, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
3. 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.
4. Y. Falcone, D. Nickovic, G. Reger, and D. Thoma. Second international competition on runtime verification CRV 2015. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, pages 405–422, 2015.
5. K. Havelund. Rule-based Runtime Verification Revisited. *International Journal on Software Tools for Technology Transfer (STTT)*, 2014.
6. K. Havelund and G. Reger. Specification of parametric monitors - quantified event automata versus rule systems. In *Formal Modeling and Verification of Cyber-Physical Systems*, 2015.
7. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, may/june 2008.
8. 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.
9. G. Reger. *Automata Based Monitoring and Mining of Execution Traces*. PhD thesis, University of Manchester, 2014.
10. G. Reger. Suggesting edits to explain failing traces. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, pages 287–293, 2015.
11. 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.
12. G. Reger, H. C. Cruz, and D. Rydeheard. MarQ: monitoring at runtime with QEA. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, 2015.
13. G. Reger, S. Hallé, and Y. Falcone. Third international competition on runtime verification CRV 2016. In *Runtime Verification - 16th International Conference, RV 2016. Proceedings*, page To Appear, 2016.
14. G. Reger and D. Rydeheard. From first-order temporal logic to parametric trace slicing. In E. Bartocci and R. Majumdar, editors, *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings*, pages 216–232. Springer International Publishing, 2015.