Rule Systems for Runtime Verification A short tutorial*

Howard Barringer[1], Klaus Havelund[2], David Rydeheard[1], and Alex Groce[3]

[1] School of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL, UK
{Howard.Barringer,David.Rydeheard}@manchester.ac.uk

[2] Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
Klaus.Havelund@jpl.nasa.gov

[3] School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, USA
Alex.Groce@eecs.oregonstate.edu

Abstract. In this tutorial, we introduce two rule-based systems for on and off-line trace analysis, RULER and LOGSCOPE. RULER is a conditional rule-based system, which has a simple and easily implemented algorithm for effective runtime verification, and into which one can compile a wide range of temporal logics and other specification formalisms used for runtime verification. Specifications can be parameterized with data, or even with specifications, allowing for temporal logic combinators to be defined. We outline a number of simple syntactic extensions of core RULER that can lead to further conciseness of specification but still enabling easy and efficient implementation. RuleR is implemented in Java and we will demonstrate its ease of use in monitoring Java programs. LOGSCOPE is a derivation of RULER adding a simple very user-friendly temporal logic. It was developed in Python, specifically for supporting testing of spacecraft flight software for NASA's next 2011 Mars mission MSL (Mars Science Laboratory). The system has been applied by test engineers to analysis of log files generated by running the flight software. Detailed logging is already part of the system design approach, and hence there is no added instrumentation overhead caused by this approach. While post-mortem log analysis prevents the autonomous reaction to problems possible with traditional runtime verification, it provides a powerful tool for test automation. A new system is being developed that integrates features from both RULER and LOGSCOPE.

Keywords Runtime verification, rule systems, temporal logic, code instrumentation, log file analysis, Java, AspectJ, Python.

* Part of the research described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

1 Introduction

This brief tutorial introduces the reader to two related rule-based systems for runtime monitoring via a number of simple examples. The first is RULER: a system that started as a low-level rule system into which one can compile different temporal specification logics for efficient trace conformance checking but then assumed a life of its own as a specification logic. The second is LOGSCOPE: a derivation of a subset of RULER that includes a simple, user-friendly, higher-level temporal pattern language, and illustrates the idea behind RULER as a target of translation from a form of temporal logic. The presentation assumes some familiarity with the basic notions of runtime monitoring/verification. Section 2 introduces the basic ideas underlying RULER and its associated trace conformance checking algorithm. Section 3 starts the brief tour of RULER beginning with the RV world's equivalent of "Hello World" specifications, putting them to use to monitor Java programs using AspectJ for instrumentation, and then gives a glimpse of one of the more powerful monitor combinator features. Section 4 introduces the ideas underlying LogScope and its higher-level temporal pattern language and then in Section 5 we explore the log file analysis system through a case study. Section 6 concludes with a brief review of our approaches and comments on future work.

2 Underlying principles of RULER

2.1 A little history

In the beginning, there was linear-time propositional temporal logic, PTL, and the stage for verification of behavioural properties of concurrent programs was set [17]. But then there were concerns voiced over the lack of expressiveness for real system specification, in particular not being able to express classes of regular properties, such as an event occurs on every even moment. And so PTL begat the family of extended temporal logics, including for example, Wolper's ETL [21], the fixed point temporal calculus νTL [3], fixed point calculi extended with chop [14], etc. Even though model checking was gaining much ground, these richer, more expressive, temporal logics raised significant challenges for such automated verification techniques. However, there was still much interest in using richer logics as a basis for formal specification. Techniques for "executing" temporal specifications were developed, for example, Moskowski's Tempura based on Interval Temporal Logic [15, 16]. With a small change in view of logic, from the declarative to the imperative, the fixed point temporal logics gave rise to METATEM [4], in which one might program directly with (recursive) temporal rules of the form "declarative past implies imperative future". The interpretation mechanism for METATEM was basically as follows. Given the current state of computation, i.e. the execution history of assignments to variables, determine which of the rules whose antecedent (declarative past formulas) conditions hold true, and then use their associated consequents (imperative future formulas) to build the next computation state. A fundamental property of these logics, the *separation property* [12], enables one to separate the conjunction of future time consequent formulas into atomic facts that have to hold now and pure future time formulas representing obligations for the future. Thus one could build an execution

trace that conformed to future time obligations, subject to the not so insignificant issue of non-determinism and looping that is potentially present.

The techniques underlying METATEM's evaluation had a significant influence on our development of runtime verification temporal logics. The plethora of different trace languages being used for property specification in runtime verification, e.g. future-time linear temporal logics, past-time temporal logics, extended regular expressions, interval logics, etc., led us to the development of the general purpose, rule-based, temporal system, EAGLE [5], which presents a natural rule/equation based language for defining, and even programming, monitors for complex temporal behavioural patterns. Here are a few examples.

$$\textbf{max}\ \texttt{Always}(\textbf{Form}\ F) = F \wedge \bigcirc \texttt{Always}(F)$$

$$\textbf{min}\ \texttt{Match}(\textbf{Form}\ C, \textbf{Form}\ R) = (C \cdot \texttt{Match}(C,R) \cdot R \cdot \texttt{Match}(C,R)) \vee \texttt{Empty}()$$

$$\textbf{min}\ \texttt{HappensBefore}(\textbf{Form}\ F, \textbf{double}\ u) = \mathit{clock} < u \wedge (F \vee (\neg F \wedge \bigcirc \texttt{HappensBefore}(F,u)))$$

Elsewhere, e.g. [8], we have argued that whilst EAGLE is elegant and expressively rich, there is a potentially high computation cost through (i) the potential non-determinism in specification and (ii) the symbolic manipulation that was required in the evaluation algorithm. As an example of (i), the concatenation operator in the `Match` predicate definition above is non-deterministic and requires considerable care in use. The temporal predicate $\texttt{Match}(\mathit{call}, \mathit{return})$ for formulas *call* and *return*, specifies the behaviour that every *call* has a matching *return*. In order to achieve this expected temporal behaviour pattern, the formulas passed to `Match` should specify single state sequences. If that is not the case, the concatenation operator may choose an arbitrary cut point, and therefore skip unmatched *C*s or *R*s in order to give a positive result. Regarding point (ii) above, we should comment that this may possibly just reflect our inability to get a good implementation. Rather than trying to improve EAGLE's implementation, we developed a lower-level rule-based system, RULER.

2.2 RULER rule systems and evaluation

The core of a RULER rule system is a collection of named rules. A rule is formed from a condition part (antecedent) and a body part (consequent). Unlike the rules in METATEM, the antecedent and consequent are restricted to be non-temporal. The rule's condition may be a conjunctive set of state expressions, whereas the body is a disjunctive set of conjunctive sets of state expressions. In the example below, `Start` is a rule, which, if active itself, will either activate the rule `Track` with argument `f` when an `openFile` observation holds for some object `f` or, if such an observation doesn't occur, will deactivate itself.

```
Start: openFile(f:obj) -> Track(f)
```

The simplest form of state expression is an observation, e.g. `openFile(f:obj)`, or a positive or negative occurrence of a rule, e.g. `Track(f)`. We will show in Section 3

some of the more complex expressions that can be used. At the basic level, rules in RULER have no default persistence, an active rule is just a single shot rule. A rule gets activated for the next evaluation step, then gets used and automatically deactivated. Whilst this mechanism is good for encoding grammars and temporal logics, it may not appear so natural for user-level rule specification where one might expect a rule's activation to persist until it has been used. We call this the *state* view in that a system remains in a particular state until some transition can be taken to move the system into another state. One could also have adopted a view that rules, once active should persist for ever, or at least until they are forcibly switched off. We chose the single-shot view purely because it is easy to translate other logics into this form (and that was an original goal for RULER, and the other notions of persistence can easily be encoded using the single-shot rules).

Given a collection of named rules and some initial conditions, a trace of input observations can be checked for conformance against the rule system as outlined below in Figure 1. For ease, we call a set of rule literals and observation literals a rule activation state and hence a frontier is a set of rule activation states. The frontier represents a choice of possible states. Overall, all the traces allowed by the set of rules are explored, in a breadth-first fashion, against the given trace of sets of observations. In a single monitoring step, the algorithm computes a new frontier (a set of sets of observation obligations and rule activations) according to the given input observations and the current frontier of states. The initial frontier is defined by the specified initial conditions. The step computation is repeated until either the monitoring input is exhausted or a conflict between the constraints of the rule system and the input has been determined. The breadth-first exploration of traces allowed by the rule system is undertaken in order to avoid backtracking when a conflict between input observations and rule system obligations occurs.

```
1: form an initial frontier of rule activation states
2: WHILE input observations exist DO
3:   obtain the next set of observations
4:   add the observations into each of the the frontier's states
5:   report failure if there's no consistent resultant state
6:   FOREACH of the current and consistent resultant states,
7:     use all active rules to form a successor set of activation states
8:   create the next frontier of rule activation states
          by taking the union of all successor sets
9: OD
10: yield success iff last frontier has an acceptable final state
```

Fig. 1. The basic monitoring algorithm

As a quick demonstration of the evaluation mechanism, consider the following four rules that, if run with the rules `Start`, `Close` and `Continue` initially active, will check that only opened files are closed.

```
Start: openFile(f:obj) -> Track(f);
Close: closeFile(f:obj), !Track(f) ->
           print("Error: closing unopened file" + f);

// to ensure persistence of the rules
Track(f:obj): !closeFile(f) -> Track(f);
Continue: -> Start, Close, Continue;
```

Before we proceed with an example evaluation of these rules, we must first explain a little of the notation used above. The first rule has name `Start`, an antecedent as the observation `openFile(f:`**obj**`)` and with its consequent as the rule `Track(f)`. The argument `f:`**obj** to `openFile` defines `f` as a formal argument name of type **obj**. During rule evaluation, the variable `f` will be bound to all values of type **obj**, say `fv`, that will cause `openFile(fv)` to match an input observation. The consequent `Track(f)`, on the other hand, uses the bound value for the variable `f`. In other words, the variable `f` in `openFile(f:`**obj**`)` is free whereas the variable `f` in `Track(f)` is a bound occurrence (bound by the matching of `openFile`). The name of the third rule `Track` also has a formal argument `f` of type **obj**; the value of the variable `f` in the observation expression `!closeFile(f)` is bound to the defining occurrence given in the rule name. The exclamation mark `!` denotes negation.

Let us assume the following trace of input observations, where `f1`, `f2` and `f3` denote different file object values.

```
{ openFile(f1) }, { openFile(f2),closeFile(f1) }, { closeFile(f3) }
```

Initially, the following frontier of active rule expressions is created.

```
{ { Start, Close, Continue } }
```

There is an input observation, hence by line 3 of the algorithm `openFile(f1)` is added across the states of the frontier, yielding the following.

```
{ { Start, Close, Continue, openFile(f1) } }
```

There is no logical inconsistency in this frontier and the loop at line 6 of Figure 1 generates the consequences of all the active rules. The rule `Start` is active and as `openFile(f:`**obj**`)` unifies with `openFile(f1)`, creating a binding of the variable `f` to the file object value `f1`, the rule consequent, i.e. `Track(f1)`, is added into the "next" set of states. The rule `Close` is active, however, its condition doesn't hold as there is no `closeFile` observation, and hence contributes nothing to the successor set of states. The rule `Track` for some file object value is not active and hence plays no role at this stage. The rule `Continue` is active and as its antecedent condition is empty will cause the rule's consequent to be added to the successor set of states. These rules thus generate the successor set of states.

```
{ { Track(f1), Start, Close, Continue } }
```

As there was only one state in the frontier initially, the above set of states becomes the next frontier.

The process now repeats with the next set of input observations, i.e.

```
{ openFile(f2), closeFile(f1) }
```

and line 4 of the algorithm thus generates the frontier.

```
{ { Track(f1), Start, Close, Continue,
                        openFile(f2), closeFile(f1) } }
```

The rule application part of the algorithm will now generate the next frontier, given below.

```
{ { Track(f2), Start, Close, Continue } }
```

Note that the rule expression `Track(f1)` is not present; there was a `closeFile(f1)` observation and hence the condition `!closeFile(f1)` fails to hold with the result that the consequent of the rule, which is itself, does not included in the successor set of states. On the other hand, `Track(f2)` does appear as a consequence on the `Start` rule.

For the final set of input observations, i.e. { `closeFile(f3)` }, the system generates

```
{ { print("Error: closing unopened file" + f3), Track(f2), Start,
      Close, Continue } }
```

as the next frontier. The condition of the rule `Close` was satisfied this time and hence gave rise to the special predicate **print** being added to the frontier, which then gets treated as an actual print statement by the RULER interpreter before the next input is read — this is just one way of reporting failures. Notice that the system continues to track the opened file `f2`.

If one assumes that this is the end of the observation trace, the system will actually check to see whether there are any rules that have been explicitly forbidden from terminal states. In this case, one might well have indicated that any `Track` rule should not be present since this indicates the occurrence of a file still being open.

3 RuleR by example

RULER is a Java-based program that implements the monitoring of systems using specifications presented as rule systems. In this section, we focus on practical issues by developing a few simple monitors in RULER, showing how it can easily be used to monitor Java programs at runtime, using AspectJ to instrument the monitored program and invoke the RULER monitor. RULER is very much an experimental system, providing a basis for us to try out different specification language concepts and monitoring features. As such, it is not a stable system, nor a publically released system (though that will change). From its very simple beginning, which implemented only the propositional rule system presented in [6] into which one might compile different temporal logical specifications, RULER has become a strongly typed, almost stream-functional-like, system for user-level programming and combining trace monitors.

3.1 Example 1: a simple response property

We begin our brief tour by writing a monitor to check the validity of responses given in answer to a simple arithmetic quiz. Our system is to observe and monitor a sequence of *question* and *answer* events. To keep matters simple, the question event has two integer arguments, and the answer event has a single integer argument. The property we require of the sequence, in terms of temporal logic is as follows.

$$\Box \forall x, y : int \cdot question(x,y) \Rightarrow \bigcirc((\neg \exists u, v, z : int \cdot answer(z) \vee question(u,v))\, \mathcal{U}\, answer(x+y))$$

This (first order) temporal formula expresses the constraint that every question is followed by a correct answer event summing the two arguments of the question, and there should also be no intervening question or answer event between the question and correct answer. In terms of RULER, we can specify the required behaviour via the following rule system.

```
ruler SumCheck{
  observes question(int, int), answer(int);

  state Check{
    question(x:int, y:int) -> Response(x+y);
  }
  state Response(required:int){
    answer(z:int)
      {: z != required ->
           print("Wrong answer! Expected "
                    + required + " but given " + z),
           Check;
         default -> Check;
      :}
    question(x:int, y:int) ->
      print("Unexpected question! Previous one unanswered"),
      Response(required);
  }
  initials Check;
  forbidden Response;
}
```

The rule system is named `SumCheck`. It defines, using the **observes** keyword, that `question` is an observation event with two integer arguments and that `answer` takes a single integer argument. Two rules are then specified. Both are introduced with the keyword **state** and indicates that the rules use *state persistence*, namely, once the rule is active, it will remain active until it is successfully used. RULER has two other persistence attributes for rules, *always persistence* and the underlying *single shot persistence*, introduced by the keywords **always** and **step**. We have found that state persistence is most common when users write RULER specifications directly, which undoubtedly

reflects a state machine oriented view of writing specifications, and so RULER will actually assume state persistence by default, reducing the amount that needs to be written. For clarity, we will maintain its use in the specifications we present in this short overview.

The first rule above is named `Check`. Its body, enclosed by the parentheses `{ }`, comprises just a single rule, which will activate the `Response` rule as soon as a `question` observation occurs. Of course, note that when such happens, the state persistence means that the `Check` rule is deactivated. On the other hand, if a `question` observation doesn't occur, then the state persistence of `Check` will keep the rule active for the next monitoring step.

The second rule defines the `Response` rule. It is supplied with the integer value that is expected to be given in the first occurence of an `answer` that follows activation of the `Response` rule. The rule has two parts, the first of which is what we've termed a factored rule; its precondition `answer(z:`**`int`**`)` has been factored out of the subsequent two sub-rules. Following the precondition, there are two rules enclosed within the parentheses `{: :}`. Such parentheses indicate that the enclosed rules are to be evaluated in the given serial order. This is not the usual interpretation of multiple rules in RULER, which are evaluated, effectively, in parallel. The condition part of the first sub-rule, `z != required`, checks for an invalid response. If the condition holds, the special **print** event is activated, together with the `Check` rule in order to continue checking answers against questions. If the condition of the first sub-rule doesn't hold, then the next rule in the list is attempted. That particular rule has **default** as its antecedent, which means it will always be able to be used; the sub-rule's consequent just restarts the checking process by activating the `Check` rule. The second part of the `Response` rule handles an occurrence of an undesired `question` event.

The last two elements of the rule system define which rules are initially active, the line with the keyword **initials**, and which rules are not allowed to be active at the end of monitoring a finite sequence of observations. We have expressly forbidden `Response` since its activity represents an unanswered question.

Finally, we have to create a monitor based on the `SumCheck` rule system.

```
monitor{
  uses SC: SumCheck;
  run SC .
}
```

The text of both the rule system `SumCheck` and the monitor definition are then the *specification* input to the RULER system. The **monitor** definition creates an instance of the `SumCheck` rule system, which is really a rule system schema, names it `SC`, and then runs it. Later we will expose one of the other ways of creating monitor expressions.

3.2 Hooking RULER up to Java via AspectJ

The current prototype of RULER provides a simple Java interface to enable its direct use from other Java applications. The Java class `RuleR.java` provides a constructor for the creation of a RULER monitor, a method for dispatching a single event to the monitor, and a method for dispatching an "end of input stream" event to the monitor.

```
public RuleR(String fileName, boolean timing){ ... }

public Signal dispatch(String eventName, Object[] argList){ ... }

public Signal dispatch(String eventName){ ... }

public Signal dispatchEnd(){ ... }
```

The first argument of the constructor provides the basename for the input file (the constructor adds a ".ruler" extension) containing the rule system schema definitions and monitor definition, and for the output file (the constructor adds a ".output" extension). The RULER monitor will send (the final) monitoring status and output events (not yet described) to the output file. The second argument specifies whether events dispatched to the monitor should be accompanied by a real-time stamp event (true for timing on).

The first argument of the first dispatch method is the string that represents the name of the observation event. The second argument provides the list of arguments that are to be associated with the event. A second version is supplied for when there are no arguments.

All the dispatch methods return a five-valued status result of the following type.

```
public enum Signal {TRUE, STILL_TRUE,
                    STILL_FALSE, FALSE,
                    UNKNOWN}
```

The status `Signal.TRUE` means that all the constraints imposed by the monitor have now been satisfied and there are no further monitoring rules active. Whereas, the status `Signal.STILL_TRUE` means that no monitoring constraints have yet been falsified, however, there are still monitoring rules active. This status condition will arise, typically, during the monitoring of a safety property. The status `Signal.STILL_FALSE` means that the monitoring constraints have not yet been satisfied, but further input may indeed do so. This status condition will arise, typically, during the monitoring of a liveness property when one is waiting for some eventuality to occur. The status `Signal.FALSE` means that the monitoring constraints have definitely been falsified and no further input can change the status. The status `Signal.UNKNOWN` means that the monitor has been unable to resolve the status of monitoring against one of the above values. This may arise when the results of two monitors are composed in parallel, for example when one yields `Signal.STILL_TRUE` and the other yields `Signal.STILL_FALSE`.

Using AspectJ for instrumentation We now show how the above interface can be used from within an AspectJ instrumentation of a Java application. The application will be the following program that we wish to monitor using the rule system `SumCheck` given above.

```
public class SumCheck{

  static void question (int x, int y){ /* what ever */ }
```

```
  static void answer (int x){ /* what ever */ }

  static void end() { /* just to indicate the end */ }

  public static void main(String[] args) {
    question(1,1); answer(2);
    question(2,3); answer(5);
    question(4,5); answer(9);
    question(1,0); answer(10);
    question(2,1); answer(3);
    end();
  }
}
```

The aspect `SumCheckInst` defines an instance of a RULER monitor, using the constructor mentioned above, from the file "src/examples/SumCheck.ruler" which contains the example ruler definitions from above. Two pointcuts are defined corresponding to calls of the methods `question(...)` and `answer(...)` in the application `SumCheck.java`. **before**-advices define the instrumentation code, each one calling the `dispatch` method of the RULER monitor instance `ruler`. In this example, if the dispatch method returns a `Signal.FALSE` status, an appropriate error message is printed on `System.err` and the system is terminated cleanly. The `dispatchEnd` method is invoked before the main application calls its `end()` method.

```
import ruler.*;

public aspect SumCheckInst{
  RuleR ruler = new RuleR("src/examples/SumCheck", false);

  pointcut question(int x, int y) :
    call(static void question(int, int)) && args(x, y);
  pointcut answer(int z) :
    call(static void answer(int)) && args(z);

  before(int x, int y) : question(x, y){
    ruler.dispatch("question", new Object[]{x,y});
  }
  before(int z) : answer(z){
    ruler.dispatch("answer", new Object[]{z});
  }
  before() : call(void end()){
    ruler.dispatchEnd();
  }
}
```

Running `SumCheck` as a Java/AspectJ application then results in the following output.

```
Rule system SC.SumCheck running... Step = 0
```

```
On monitoring step 8: Wrong answer! Expected 1 but given 10
End of monitoring on step 11: status of SC.SumCheck is still_true
```

3.3 A Java API example

In the Java collections framework, the `Iterator` interface provides three methods to support iteration over a collection, `hasNext`, `next` and `remove`. The method detail from the Sun Java 1.6 documentation [20] for the `remove` method states the following.

> Removes from the underlying collection the last element returned by the iterator (optional operation). This method can be called only once per call to next. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.
>
> **Throws:**
>
> UnsupportedOperationException - if the remove operation is not supported by this Iterator.
>
> IllegalStateException - if the next method has not yet been called, or the remove method has already been called after the last call to the next method.

Furthermore, when an iteration has no further elements a call of the `next` method will raise the NoSuchElementException.

We will write a RULER monitor that pre-empts erroneous calls of the iterator methods. We will use the following AspectJ instrumentation code to dispatch an event on each and every occurrence of any iterator method call. In the current version of RULER the `dispatch` method returns only the status value (`Signal`); other pertinent information, such as rules used, error locations, etc. will be included in subsequent versions. We therefore currently have to include in the AspectJ instrumentation knowledge about errors, such as the error messages passed to the `Check` method called from the **before**-advices.

```
public aspect iteratormonitor{

 RuleR ruler = new RuleR("src/examples/safeIteration", false);

 static void check(Signal s, String message){
   if (s==Signal.FALSE){
     System.err.println("Failure on " + message);
     System.exit(0);
   }
 }

 pointcut hasNext(Iterator i) :
   call(* java.util.Iterator+.hasNext()) && target(i);
 pointcut next(Iterator i) :
   call(* java.util.Iterator+.next()) && target(i);
```

```
  pointcut remove(Iterator i) :
    call(* java.util.Iterator+.remove()) && target(i);

  before(Iterator i): hasNext(i){
    ruler.dispatch("hasNext", new Object[]{i});
  }
  before(Iterator i): next(i){
    check(ruler.dispatch("next", new Object[]{i}),
                    "No preceding call of hasNext");
  }
  before(Iterator i): remove(i){
    check(ruler.dispatch("remove", new Object[]{i}),
                    "Must call next before remove on iterator");
  }
}
```

Below we present a RULER monitor which observes events issued immediately before calls to the iterator methods `hasNext`, `next` and `remove`. The events are supplied with the actual iterator object so that tracking of multiple uses of iterators can take place at the same time. RULER handles object references as weak references so that the monitoring is transparent as far as garbage collection in the original application is concerned.

```
ruler SafeIteratorCheck{
  observes  hasNext(obj), next(obj), remove(obj);

  always Start{
    hasNext(i:obj) -> Next(i);
  }
  state Next(i:obj){
    next(i) -> Remove(i);
  }
  state Remove(i:obj){
    remove(i) -> Ok;
  }

  assert Start, Next, Remove;
  initials  Start;
}
```

Of the three rules defined in the above RULER specification, the `Start` rule is of always activation persistence. Whenever a `hasNext` event occurs for some iterator `i`, the state rule `Next` is activated to track uses of the next and remove methods on that specific iterator. If an associated `next` event occurs, then the state rule `Remove` is activated. The latter rule simply checks for an associated `remove` or `hasNext` event, taking no action (the **Ok** keyword) other than satisfying the rule and deactivating itself if either event occurs. So how does this system not accept traces with two `remove` events with

no intervening `next` event for the same iterator, or indeed, two `next` events with no intervening `hasNext` event? There is one further rule defined by the line starting with the keyword **assert**. It specifies that on every monitoring step one of the listing rule names must be applied successfully; if there is a monitoring step in which none of the rules applies then the dispatch method returns a status signal of `FALSE`. Suppose, therefore, the monitoring has rules `Start` and `Remove(i)` active for some iterator `i`, which means a `remove` event for iterator `i` is allowed. The `Remove(i)` rule will disappear as soon as `remove` event occurs, in which case only the `Start` rule remains active. If another `remove(i)` were to occur, no rule would be successfully used, breaking the constraint imposed by the **assert** directive. There are of course several different ways to specify the desired behaviour and this one was chosen simply to highlight the use of the **assert** directive.

The Java code fragment below contains a deliberate coding error, one that could easily occur in practice. `myIt` iterates over a list of numbers. Each list element is checked against another array of integers to see whether it is to be removed from the list (whether value is a multiple of an element of the removes array). Unfortunately, the programmer forgot to stop the inner loop after removing an undersirable value.

```
  Iterator<Integer> myIt = myList.iterator();

  while (myIt.hasNext()){
    int value = myIt.next();
    for (int i = 0; i<removes.length; i++){
      if (isMultiple(value,removes[i])){
        myIt.remove();
//      break;
      }
    }
  }
```

To give an indication of the cost of this on-line monitoring, the corrected example, obtained by uncommenting the `break` statement, was timed for a list of 50000 integers with a filtration array of size 5. Without any monitoring, the loop execution took approximately 0.75 seconds and 5.37 seconds with on-line monitoring. The AspectJ intrumentation alone doubled the execution time, indicating that the RULER monitoring had close to 4-fold cost. This may seem high, however, one should note that the amount of computation undertaken in the monitored application between the issuance of consecutive monitoring events is very small.

3.4 Monitor Chaining

An interesting experimental feature of RULER is its monitor combinator capability. Here we exemplify just one form, namely *chaining*. RULER monitors can be viewed as event stream transducers, transforming an input event stream into an output event stream. So far, we have illustrated rule systems that transformed the input stream of observation events to an output stream of status values. However, RULER enables users

to define their own output events. Furthermore, for the previous examples, the output stream of the monitor has been implicitly directed to a file. However, chaining allows the output stream of one monitor to be the input stream of another. In effect, a monitor can be created that monitors the output of a preceding monitor. Indeed, this can be viewed as a helpful design strategy for complex monitoring situations; it is supporting a form of abstraction. The example given below also illustrates how close RULER is to a stream-based functional programming language. This, of course, raises important questions about the differences between specification languages and programming languages.

Consider a Java application that uses (recursive) tree structures for storage and retrieval of information. A RULER monitor is required to analyse the use of these structures in terms of (some form of) efficiency of search. The tree structure class provides a recursively defined method, `find`, for search. We assume that the application is instrumented to report calls to, and returns from, the method `find`. We define two monitors `Trace` and `StatsGatherer`. The first will determine the maximum depth reached in the associated structure to find an item and send the ratio of maximum depth to size of tree to a second monitor that will collect and analyse statistics of use of the structure. The second monitor will make reports when undesirable performance occurs.

```
ruler Trace(traceOut:obs){
  observes call(obj,int), return(obj,int);
  state Top{
    call(tree:obj,size:int) -> Stack(tree, 1, 1, size);
  }
  state Stack(tree:obj, level:int, max:int, size:int){
    call(x:obj, y:int) -> Stack(tree, level+1, max+1, size);
    return(x:obj, b:int)
      {: level==1
         {: b==1 -> traceOut(tree, 1.0*max/size), Top;
            default -> Top;
         :}
       default -> Stack(tree, level-1, max, size);
      :}
  }
  initials Top;
  outputs traceOut;
}
```

The `Trace` monitor observes `call` and `return` events. The first argument of both event types is the tree object being searched. The second argument of a `call` event is the size of the tree structure, whereas the second argument of the `return` event gives the return value of the associated find method (1 means item found, 0 not found). Given an initial `call` event, the subsequent `call`s and `return`s relate to recursive invocations of the instrumented application's search method. The `Trace` monitor thus matches `call`s and `return`s and keeps track of the maximum depth reached. When a top-level `return` event occurs, a `traceOut` event is output if the search call was successful. The `traceOut` event has two items of data, the tree reference and the ratio of maximum depth to the size of the tree.

```
ruler StatsGatherer(traceIn:obs, G:double, B:double){
  locals  report(obj,int, int);
  always Start{
    traceIn(t:obj, ratio:double), !Track(t,x:int,y:int)
      {: ratio < G -> Track(t,1,0);
         ratio > B -> Track(t,0,1);
         default   -> Track(t,0,0);
      :}
  }
  state Track(tree:obj, Gs:int, Bs:int){
    traceIn(tree, ratio:double)
      {: ratio < G -> Track(tree, Gs+1, Bs);
         ratio > B -> Track(tree, Gs, Bs+1);
         default   -> Track(tree, Gs, Bs);
      :}
    (Bs != 0) && (Gs != 0 ) && (2*Bs > 3*Gs)
                      -> report(tree, Bs, Gs);
  }
  initials  Start;
  outputs  report;
}
```

The events output by the `Trace` rule system schema are to be consumed by a monitor based on the rule system `StatsGatherer` given above. The schema has two arguments `G` and `B` of type **double** which are used to categorize the ratio of depth of search over size of tree as good, ok and bad performance. The monitor is designed to report when twice the number of bad searches exceeds three times the number of good searches. `StatsGatherer`'s initial rule has `always` activation persistence but it only starts tracking results for a particular tree if it is not tracking that tree already. The variables `t`, `ratio` are bound by matching `traceIn(t,ratio)` against a (grounded) observation `traceIn(...)`. Note that `traceIn` is a RULER schema argument and will be replaced by the actual name of the event on instantiation of the schema. Thus the variable `t` appearing in the expression `!Track(t, x:int, y:int)` is already bound, however, the other two variables will be bound through attempted matches against an appropriate observation event. The second rule body of the `state` rule `Track` outputs a report when the desired performance criteria are not met. Again, instead of simply reporting the situation, one would want, at that stage, to instigate some change in the application structure to improve the desired performance, e.g. re-shaping the tree, for which, undoubtedly, rather more data would need to be gathered.

Finally, the monitor to be run is constructed by chaining together an instance of the `Trace` rule system schema with an instance of the `StatsGatherer` rule system schema, the latter requiring its parameters `G` and `B` to be instantiated with actual values (`0.33` and `0.5`).

```
monitor{
  uses T: Trace, S: StatsGatherer;
  locals  info( obj, double);
```

```
  run (T(info) >> S(info, 0.33, 0.5)) .
}
```

3.5 Other features

RULER includes a number of other features to simplify and shorten the writing of monitoring specifications. These include: non-positional parameter naming; the notion of "success" rule sets, almost dual to the notion of "forbidden" rule sets; conditional, looping and parallel combinations of monitors; set and list types and associated operations; and so on. The RULER tutorial [7] documents the majority of these, however, RULER is an experimental system and it changes at the whim of its authors. As briefly mentioned before, the expansion of these user-oriented features brings our specification language very close to a stream-functional programming language, enabling generic, compact, but sometimes obscure, specifications. Some masters-level student-driven case studies with RULER have been undertaken in the areas of (i) off-line monitoring of logs from an on-line assessment system, (ii) active on-line, or supervisory, monitoring in which monitoring results trigger evolutionary action in the application system, and (iii) in providing fault explanation. These small studies indicate that the underlying approach with RULER holds promise for specialised applications, as has been developed with the LOGSCOPE system that we describe in the following section.

4 The LOGSCOPE system

4.1 A little history

The LOGSCOPE monitoring system was developed in Python after the RULER system in order to specifically support testing of NASA's next Mars Rover, the *Mars Science Laboratory* (MSL), to be launched in 2011. MSL is a compact car size rover, developed at the Jet Propulsion Laboratory. It is programmed in more than 3 million lines of code (including auto-generated code), more than the combined code onboard all previous Mars missions together. This trend in software growth is expected to continue also for future Mars missions, making it a challenge to apply traditional formal methods. The system is highly multi-threaded with approximately 160 threads. It is programmed in C by a team of 30+ programmers. The system is tested by a team of 10+ testers. Testing is complicated by the fact that the programming team has little time for activities not directly related to development of new software, and hence cannot be disturbed too much by a testing effort. This prevents for example an approach based on new test-specific code instrumentation, be it automated or not. In addition, the system is difficult to execute due to its tight connection with hardware. This makes test-input generation and multiple automated re-runs a challenging task.

However, to our advantage and independently of the effort described in this paper, the MSL flight software produces rich logs, which are stored in SQL databases. A log is a collection of time-stamped events, where an event is a mapping from fields to values (a record). Test scripts are written in Python to test these events. A script typically consists of a sequence of sub-tests, where a sub-test consists of submitting a command to

the rover, and then checking that as a consequence certain events happen or do not happen thereafter. Checking the occurrence (or non-occurrence) of events is done through various API calls. These calls also check the values of various arguments to the events, which have to co-relate in certain ways. Attempts have been made to run the Python test scripts concurrently with the MSL flight software, hence checking the occurrence of logging events on-the-fly as they happen. However, this online approach turned out to be problematic due to the fact that events are not necessarily observed in the order generated due to delays in the system.

It was therefore decided instead to perform off-line analysis of the logs stored in the SQL databases. It was furthermore perceived advantageous to construct a specification language for writing properties about these logs. At this point a decision could have been made to apply RULER. However, two constraints caused us to re-develop a variation of RULER in Python. First, engineers seemed to experiment with an informal notion of temporal logic in their comments to the Python code, explaining what the code was supposed to do. We concluded that it might be useful to support a notation close to this, a notation that is not directly supported by RULER. Second, it should be possible to integrate the specification framework with Python in a painless manner, supporting a mixture of Python and the domain specific specification language, and minimizing the number of programming languages involved. It should be stated that it would have been perfectly possible to map the temporal logic to RULER.

This effort led to the development of LOGSCOPE. LOGSCOPE is a Python program that supports analysis of logs for testing purposes. The tool in principle takes as input a log and a specification of expectations wrt. the format of the log, and produces as output a report on violations of the specification. A log is a Python sequence containing Python dictionaries (maps from fields to values) as events.

LOGSCOPE supports an automaton language that conceptually forms a subset of the RULER language. The automaton language has also been influenced by the graphical RCAT state machine language [18] and by the textual state machine language RMOR [11] for monitoring C programs. LOGSCOPE furthermore adds a temporal logic with sequencing, and translates it into the automaton subset. Several systems exist supporting different logics, such as past time temporal logic [13], future time temporal logic [19], regular expressions [1], and state charts [10]. The MOP system [9] implements a series of such traditional logics as separate plugins, but within the same framework. Data parameterization is in MOP handled separately from the interpretation of the logics. This is in contrast to RULER and LOGSCOPE, where data parameterization is an integral part of the logic. Andrews and Zhang [2] offer a parameterized state machine framework similar to LOGSCOPE's parameterized state machines. Their state machines are compiled into Prolog.

4.2 Overview of LOGSCOPE

The LOGSCOPE specification language consists of two sub-languages: (i) a higher-level *pattern language*, much resembling a temporal logic, and (ii) a lower-level, but more expressive, RULER-like automaton language. Patterns are automatically translated to automata. It is the intention that the user should mostly write patterns, but automata can become necessary in cases where the extra expressive power is needed. The pattern

specification language can be characterized by the following incomplete grammar (the non-terminal *event* is not defined):

⟨*pattern*⟩ → **pattern** ⟨*NAME*⟩ `":"` ⟨*event*⟩ `"=>"` ⟨*consequence*⟩

⟨*consequence*⟩ →
⟨*event*⟩
| `"!"` ⟨*event*⟩
| `"["`⟨*consequence*$_1$, ... ,*consequence*$_n$⟩`"]"`
| `"{"`⟨*consequence*$_1$, ... ,*consequence*$_n$⟩`"}"`

A pattern has a name that can be referred to in violation reporting. On the occurrence of an event (left hand side of the => symbol), a consequence is expected to follow. A consequence can either be an event that should occur eventually, or the event should not (‘`!`’) occur eventually. Consequences can also be composed, ordered (`[ ... ]`) or unordered (`{ ... }`). An ordered sequence of consequences have to be full-filled in the order given, in contrast to unordered sequences. Examples below will illustrate these concepts as well as the details of events.

The automaton language is conceptually a simple subset of the RULER specification language. What is referred to as a *rule system* in RULER is in LOGSCOPE referred to as an *automaton*. Each individual rule of an automaton is attached to a source state, and can only be triggered by the occurrence of a single monitored event. The result of a rule can only be the conjunction of target states (no disjunction). States cannot be negated: a state is active until left, as in traditional state machines. States can be parameterized with data as in RULER. However, automata cannot be parameterized, and there are no operations defined on automata (such as chaining). The main difference from traditional state machines is the data parameterization of states; the fact that a transition can enter multiple target states with conjunctive semantics (they must all lead to success); the predicates and actions on labels; and the different forms of states as will be illustrated. LOGSCOPE's automaton language represents an interesting practically effective subset of RULER.

5 LOGSCOPE by example

5.1 Running LOGSCOPE on a log

A log is a sequence of events. LOGSCOPE more specifically analyzes a Python sequence of events, where an event is assumed to be a Python dictionary: a mapping from field names (strings) to values (in any of Python's data formats). A special field named `"OBJ_TYPE"` must be defined in all events, and must be mapped to a string indicating what kind of event it concerns. In MSL five such kinds of events are considered:

- `COMMAND`: commands issued to the spacecraft (input to the system).
- `PRODUCT`: science results produced by the software/hardware (output of the system).
- `EVR`: internal transitions (EVent Report).
- `CHANNEL`: samplings of the spacecraft state.

– CHANGE: delta changes to the spacecraft state.

We shall as an example consider the following simplified script file creating a log of 5 events, and then calling LOGSCOPE to analyze it against the specification in a file "specs/msl-test":

```
import logscope
log =
  [
   {"OBJ_TYPE" : "COMMAND", "Type" : "FSW",
     "Stem" : "PIC_4", "Number" : 231, "Bit" : 1, "Size" : 2000},
   {"OBJ_TYPE" : "EVR", "Dispatch" : "PIC_4", "Number" : 231},
   {"OBJ_TYPE" : "CHANNEL", "DataNumber" : 5},
   {"OBJ_TYPE" : "EVR", "Success" : "PIC_4", "Number" : 231},
   {"OBJ_TYPE" : "PRODUCT", "ImageSize" : 1200}
  ]
logscope.monitor(log,"specs/msl-test")
```

Note that normally the log will be produced by the running system to be monitored. The log corresponds to a command "PIC_4" (take a picture) being fired, followed by a dispatch of that command, then a channel observation, then a success of the command, and finally a data product. Commands in the log are numbered consecutively for identification, and related events are given the same number as the command causing them. In the subsequent sub-sections we shall compose the specification in the file "specs/msl-test".

5.2 Simple properties

Assume we want to express the property: R_1*: "Whenever a flight software command is issued, then eventually an EVR should indicate success of that command"*. Our log satisfies this specification since event number 1 (the command) is matched by event number 4, the success. The following LOGSCOPE pattern formalizes this requirement:

```
pattern P1:
  COMMAND{Type:"FSW", Stem:x, Number:y} =>
    EVR{Success:x, Number:y}
```

This pattern states that *if* a flight software ("FSW") command is observed in the log with the Stem (name) field having some unknown value x, and the Number field having some unknown value y; *then* later in that log, an EVR should occur with a Success field having x as value and a Number field having y as value. In between the {...} brackets occur zero, one or more constraints, each consisting of a field name (without quotes), and a range specification. We saw two forms of range specifiations: the string "FSW" for the field Type and the names x and y for the other fields. A string constant represents a concrete constraint: the field in the event has to match this value exactly (by Python equality ==). One can also provide an integer as such a concrete range constraint. A name (x and y in this case) has one of two meanings: (i) either the name has not been

bound before, and it is bound to the value of the field in the current event, or (ii) it has been bound before, and it functions as a constraint: the field now has to have the value the name was bound to by the previously binding event.

A consequence can also be the negation ('!') of an event. Suppose we want to state the following property: *R_2: "Whenever a flight software command is issued, then thereafter no EVR indicating failure of that command should occur"*. This can be expressed by the following pattern, also satisfied by our log:

```
pattern P2:
  COMMAND{Type:"FSW", Stem:x, Number:y} =>
    !EVR{Failure:x, Number:y}
```

5.3 Composite properties

As an example, consider the following requirement: *R_3: "Whenever a flight software command is issued, there should follow a dispatch of that command, and then exactly one successful execution. There should be no dispatch failure before the dispatch and no failure between dispatch and success"*. This property can be stated as follows.

```
pattern P3 :
  COMMAND{Type:"FSW", Stem:x, Number:y} =>
    [
     !EVR{DispatchFailure:x, Number:y},
      EVR{Dispatch:x, Number:y},
     !EVR{Failure:x, Number:y},
      EVR{Success:x, Number:y},
     !EVR{Success:x, Number:y}
    ]
```

The consequence consists of a sequence (in square brackets `[...]`) of (sub) consequences: events and negations of events. The ordering means that the dispatch should occur before the success, and the negations state what should *not* happen in between the non-negated events.

It is also possible to indicate an un-ordered arrangement of events. For example, suppose we are not concerned about the order in which events occur, except that after a success there should not follow another success. This can be formulated as follows:

```
pattern P4 :
  COMMAND{Type:"FSW", Stem:x, Number:y} =>
    {
      EVR{Dispatch:x, Number:y},
     [EVR{Success:x, Number:y},!EVR{Success:x, Number:y}],
     !EVR{DispatchFailure:x, Number:y},
     !EVR{Failure:x, Number:y}
    }
```

The curly brackets `{ ... }` indicate an un-ordered collection of consequences. The fact that they are un-ordered means that the non-negated events can occur in any order, and negations have to hold at all time after the triggering command. However, nested inside the `{ ... }` construct we have an ordered sequence `[ ... ]` expressing that after one success should not follow another success.

5.4 Event predicates and actions

Events can be associated with predicates and actions. Predicates perform more sophisticated checks on values of bound variables and consequently restrict the matching. Actions are executed with side effects on a global state when events and their predicates match. Event predicates as well as actions may refer to user-introduced Python code. The following example formalizes the following requirement: *R_4: "After a picture command (commands with names starting with* `"PIC"`*) should follow, before the occurrence of the next flight software command, a channel reading of the* `DataNumber` *bitvector (integer) variable where bit 0 is the value of the command's* `Bit` *field, and subsequently should follow exactly one image product, and it should be of a size less than the command's* `Size` *field".*

```
{:
  def bit(p,n):
    return (int(n) >> int(p)) & 1
:}

pattern P5 :
  COMMAND{Type:"FSW", Stem:x, Bit:y, Size:z}
    where {: x.startswith("PIC") :}  =>
  [
    CHANNEL{DataNumber:d} where {: bit(0,d) == y :},
    PRODUCT{ImageSize:s} do {: assert s < z :},
   !PRODUCT{}
  ] upto COMMAND{Type:"FSW"}
```

The specification starts with a definition in Python of the function `bit(p,n)`, returning the bit in position `p` (counted from the right) of the number `n`. The Python code must be enclosed with the symbols `{: ... :}` (and occur at the beginning of the specification file). An atomic predicate in the pattern definition can be an arbitrary Python expression, also delimited by the symbols `{: ... :}`, as in: `{: bit(0,d) == y :}`. Predicates can be composed using the traditional Boolean operators: **and**, **or**, **not**, and brackets `( ... )`. The **do** statement associated with the `PRODUCT` event expresses that when a product is observed, that Python statement is executed. In this case it simply executes an **assert** statement testing the size of the product. Note that this is different from an event of the form:

```
PRODUCT{Stem:x, ImageSize:s} where {: s < z :}
```

This latter event would cause the monitor to wait until a product was observed matching the condition, consequently ignoring any badly sized product before that.

5.5 Translation to automata

Patterns are translated into the RULER-like automaton language. The following automaton is the result of translating the last introduced pattern `P5` above.

```
automaton P5 {
  always S1 {
    COMMAND{Type:"FSW", Stem:x, Bit:y, Size:z}
      where {: x.startswith("PIC") :} => S2(y,z)
  }

  hot state S2(y,z) {
    CHANNEL{DataNumber:d} where {: bit(0,d) == y :} => S3(z)
    COMMAND{Type:"FSW"} => error
  }

  hot state S3(z) {
    PRODUCT{ImageSize:s} do {: assert s < z :} => S4
    COMMAND{Type:"FSW"} => error
  }

  state S4 {
    PRODUCT{} => error
    COMMAND{Type:"FSW"} => done
  }
}
```

An automaton is expressed in terms of states and transitions between states triggered by events. Events are exactly as in patterns, including predicates and actions. Just as events can be parameterized with values as we have seen above, so can states, as in RULER, hence carrying values produced by incoming transitions. The automaton has four states: `S1`, `S2`, `S3` and `S4`. State `S1` has one exiting transition, labelled with the command event, and entering the parameterized state `S2(y,z)`. State `S1` is an **always** state, meaning that the state remains active after the detection of a command, in order to allow for detection of further commands. State `S2` is a **hot** state, meaning that this state must be left before the end of the log, otherwise an error is reported. The state has a transition to state `S3(z)` labelled with a channel event predicated with the bit-constraint on the `DataNumber`. The transition `COMMAND{} =>` **error** comes from the scoping **upto**-construct. It represents the property that the channel observation must be observed before the next flight software command. The other `COMMAND` transitions in states `S3` and `S4` also derive from the **upto**-construct. State `S4` is an un-parameterized normal state (not an **always** state and not a **hot** state) monitoring that no second product occurs before the next command.

6 Concluding Remarks

The two systems presented in this tutorial, RULER and LOGSCOPE, are based on the same underlying principle of a specification consisting of a set of rules, operating on a collection of states (disjuncts), each of which represents a possible path of success, and each being a collection of facts (conjuncts), where a fact is a named record of field-value pairs. In LOGSCOPE, disjunction is not supported and only one state is active at any point in time. While the RULER language focuses on defining a general and powerful rule-language, the LOGSCOPE language focuses on temporal logic, and its translation into an automaton-like subset of this general rule-language. RULER has grown considerably from the simple target language for interpreting different temporal logics, but its basic core preserves the simplicity of the original propositional system.

RULER has the flavour of a stream-functional programming language. This raises the question of the relationship between runtime verification languages and programming languages. For example, is a functional programming language better suited for runtime verification, or is it preferable to use the programming language of the application being monitored with some additional monitoring oriented features, in a sense creating an integrated specification and programming language? The work presented here explores the first choice (a functional language), leaving the alternative an open question.

In this paper RULER was applied to monitor Java programs online using aspects, whereas LOGSCOPE was defined for monitoring logs offline. However, RULER can just as well be used for log analysis, and supports both "infinite" trace online monitoring and finite trace offline monitoring. Similarly, since the monitoring algorithm in LOGSCOPE is equivalent to the one in RULER, LOGSCOPE can be used for online monitoring, although LOGSCOPE only makes two-valued verdicts about traces.

Current work in progress consists of defining a new system, RULER V2.0, which incorporates lessons learned from both these systems. Indeed, the users of LOGSCOPE have found the temporal logic pattern language intuitively easy and natural. One goal is to integrate a variant of the temporal logic in RULER V2.0. An important aspect of the new system will be optimization of the algorithms used, in particular, finding efficient event-driven rule indexing methods.

7 Acknowledgements

The first two authors would like to thank the Royal Academy of Engineering for the award of a Distinguished Visiting Fellowship, which enabled Klaus Havelund spend time at the School of Computer Science in Manchester in order to progress elements of the work presented in this paper.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*. ACM Press, 2005.

2. J. H. Andrews and Y. Zhang. General test result checking with log file analysis. *IEEE Transactions on Software Engineering*, 29(7):634–648, 2003.
3. B. Banieqbal and H. Barringer. Temporal logic with fixed points. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1987.
4. H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. MetateM: an introduction. *Formal Aspects of Computing*, 7(5):533–549, 1995.
5. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.
6. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. In *Proc. of the 7th International Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 111–125, Vancouver, Canada, 2007. Springer.
7. H. Barringer, D. Rydeheard, and K. Havelund. RuleR: A tutorial guide. Available at: `http://www.cs.man.ac.uk/~howard/LPA.html`, 2008.
8. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *Journal of Logic and Computation*, 2009. Advance Access published on November 21, 2008. doi:10.1093/logcom/exn076.
9. F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, 2007.
10. D. Drusinsky. *Modeling and Verification using UML Statecharts*. Elsevier, 2006. ISBN-13: 978-0-7506-7949-7, 400 pages.
11. K. Havelund. Runtime verification of C programs. In *Proc. of the 1st TestCom/FATES conference*, volume 5047 of *LNCS*, Tokyo, Japan, June 2008. Springer.
12. I. M. Hodkinson and M. Reynolds. Separation - past, present, and future. In S. N. Artëmov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, editors, *We Will Show Them! (2)*, pages 117–142. College Publications, 2005.
13. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a run-time assurance tool for Java. In *Proc. of the 1st International Workshop on Runtime Verification (RV'01)*, volume 55(2) of *ENTCS*. Elsevier, 2001.
14. M. Lange. Alternating context-free languages and linear time mu-calculus with sequential composition. *Electr. Notes Theor. Comput. Sci.*, 68(2), 2002.
15. B. Moszkowski. *Executing temporal logic programs*. Cambridge University Press, 1980.
16. B. C. Moszkowski and Z. Manna. Reasoning in interval temporal logic. In E. M. Clarke and D. Kozen, editors, *Logic of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 371–382. Springer, 1983.
17. A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.
18. M. Smith and K. Havelund. Requirements capture with RCAT. In *16th IEEE International Requirements Engineering Conference (RE'08)*, IEEE Computer Society, Barcelona, Spain, September 2008.
19. V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th International Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*. Elsevier, 2005.
20. Sun Microsystems, Inc. *Java Platform, Standard Edition 6, API Specification*, 2009. see: http://java.sun.com/javase/6/docs/api/.
21. P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56, 1983.