RuleR: A Tutorial Guide
DRAFT: Version 0.11

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

Klaus Havelund
NASA's Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
EMAIL: Klaus.Havelund@jpl.nasa.gov

June 30, 2008

Abstract

RuleR was introduced as a primitive conditional rule-based system, which we claim can be efficiently implemented for run-time checking, and into which one can compile various temporal logics used for run-time verification. RuleR was created in the spirit of Eagle but has a considerably easier semantics. Although our original goals positioned RuleR as a low-level target rule-system for easy interpretation, following a prototype Java implementation of RuleR incorporating data and rule parameters, we have since enhanced RuleR to support its direct use as a monitor specification system. This tutorial takes the reader on a tour through the current system, its enhancements over "core" RuleR and examples of its use in monitoring Java programs instrumented using AspectJ. We recommend, however, that the paper [2] is read first.

RuleR remains an experimental system with an implementation that evolves freely with the whim of its originators. Not all of its features (including bugs) are documented.

Keywords

Run-time verification, rule systems, temporal logic, grammars, Java, AspectJ.

1 Introduction

This guide introduces the reader to RuleR via a number of simple examples. It assumes familiarity with the basic notions of run-time monitoring/verification and, importantly, that the reader has read, or at least scanned, the paper [2]. Section 2 starts the tour with some simple propositional RuleR examples and then shows how RuleR can be used in conjunction with Java/AspectJ. Section 3 then describes the first of the rule optimizations in which rules with different default persistencies are introduced. Section 4 gives examples of parameterized RuleR further extensions/rule optimizations. Section 5 indicates how real-time, time-stamped observation events are introduced and used. Section 6 describes a host of other rule extensions, such as assert, watch, output rules, and monitor expressions. Finally, Section 7 addresses how RuleR can be used to drive action in the monitored application.

2 Basic Propositional RuleR Examples

2.1 Example 1: a simple safety property

Our first example is to write a monitor that requires an observation, denoted by the identifier a, to always be present. In terms of temporal logic, the monitor encodes the formula □a. Figure 1 contains the text that would be input to RuleR. It starts with a definition of a rule system schema, introduced by the keyword **ruler**, named Always. This particular rule system contains:

- a definition of the rule identifiers, introduced by the keyword **ruleIDs**
- a definition of the observation identifiers, introduced by the keyword **observes**
- a definition of the rules, introduced by **rules**
- a definition of which rule identifiers are initially active, introduced by the keyword **initials**

```
ruler Always {
    ruleIDs { Ra, Rna }
    observes { a }
    rules {
        Ra: a -> Ra, Rna;
        Rna: !a -> Fail;
    }
    initials { Ra, Rna }
}

monitor {
    uses { A: Always }
    run A .
}
```

Figure 1: A RuleR monitor for $\Box a$

There are two rule identifiers, Ra and Rna. The rule denoted by Ra is a simple conditional, such that when active, checks whether the observation a is present and, if so, makes the rules Ra and Rna active for the next monitoring step. This rule thus perpetuates itself provided the observation a is present. Initially, both Ra and Rna are made active. The rule denoted by Rna checks on the presence of !a, i.e. the negation of observation a; this holds if either the observation !a is present in the observation state, or the observation a is not present in the observation state. If the rule Rna fires, i.e. its condition evaluates to true, then the consequent **Fail**, which corresponds to false, occurs. This will cause the monitoring to fail and a failure result passed back to the system running the monitor (this will be explained more fully after the next example).

The input to the RuleR system also contains a definition of a monitor, introduced by the keyword **monitor**. This monitor definition has two parts: an instantiation of a rule system schema, introduced by the keyword **uses**; and a monitor expression that is to be run, introduced by the keyword **run**. The former simply creates an instance of the schema, in this case Always, whose names, apart from observations, are prefixed by the instance name, in this case A. This thus allows uniquely named instances of schema definitions. The run expression here is simply the name of the instantiated schema to be run as a monitor.

It should be clear that the rules Ra and Rna remain active as long as the observation a keeps occurring. As soon as the observation a doesn't occur, the rule names are no longer made active and the monitor is forced to emit a failure signal.

2.2 Example 2: a simple liveness property

Our next example is to write a monitor that requires an observation, denoted by the identifier b, to eventually occur during a monitoring run. The rule system schema definition Eventually in Figure 2 has

```
ruler Eventually {
    ruleIDs { Rb, Rnb }
    observes { b }
    rules {
        Rnb: !b -> Rb, Rnb;
        Rb: b -> Ok;
    }
    initials { Rb, Rnb }
    forbidden { Rb }
}

monitor {
    uses { E: Eventually }
    run E .
}
```

Figure 2: A RuleR monitor for $\Diamond b$

a very similar form to Always of Figure 1 except that it also uses the keyword **forbidden** that defines a set of rule identifiers that are *not* allowed to exist in the rule activation state at the end of monitoring. In the given example, the rule Rb is used to check for the presence of the desired observation. If it exists, the consequent of the rule, **Ok**, means that no obligations are placed on the next monitoring state (formally, it corresponds to true). The rule Rnb, on the other hand, perpetuates both itself and rule Rb if the observation is absent or its negation is present. If at the end of monitoring, the rule Rb is still present in all of the possible rule activation states[1], then clearly the observation b hasn't occurred and, hence, monitoring should fail. For this (deterministic) example, as soon as rule Rb fires, the rule activation state collapses and the monitor is able to emit a success signal.

2.3 Example 3: A regular pattern

Suppose we wish to monitor the pattern that should ever an observation g occur then it is immediately followed by an observation sequence with a prefix that matches the regular expression (ab)∗c. Figure 3 defines a suitable rule system encoded in the primitive, core, RuleR system. The rules Rg and Rng perpetuate a check for the observation g. Whenever g is detected, then the rules Sa, Sc and Snac are activated. The first two of these check for an a or a c observation. The other rule, Snac, will cause failure if neither an a nor a c occur. If rule Sa fires, then rules Sb and Snb ensure that a b immediately follows (with failure if it doesn't). Successful firing of rule Sb then repeats the regular pattern by reactivating rules Sa, Sc and Snac. Notice that the rule system schema also forbids the presence of rules Sa, Sb and Sc, which means that the regular pattern (ab)∗c must be completed before termination of the monitoring.

Exercise 1 *Modify the rule system REPattern so that the trigger observation g pattern is only sought again after successful matching of the pattern (ab)∗c rather than on every monitoring step.*

2.4 Hooking up RuleR to Java/AspectJ

The current prototype of RuleR is coded in Java. A simple interface exists 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,
```

[1] although there will be only one possible rule activation state for this example, recall that a breadth first exploration is undertaken by the monitor, covering possible choices in a rule's consequent

```
ruler REPattern {
    ruleIDs { Rg, Rng, Sa, Sb, Sc, Snb, Snac }
    observes { g, a, b, c }
    rules {
        Rg: g -> Sa, Sc, Snac, Rg, Rng;
        Rng: !g -> Rg, Rng;

        Sa: a -> Sb, Snb;
        Sc: c -> Ok;
        Snac: !a, !c -> Fail;

        Sb: b -> Sa, Sc, Snac;
        Snb: !b -> Fail;
    }
    initials { Rg, Rng }
    forbidden { Sa, Sb, Sc }
}

monitor {
    uses { RE: REPattern }
    run RE .
}
```

Figure 3: A RuleR monitor for $\Box(g \rightarrow (ab) * c)$

```
                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 described yet, see Section **??**) 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, see section 5).

The first argument of the dispatch method is a string representing the name of the observation event used in the monitor. The second argument provides the list of arguments that are to be associated with the event — see section 4. An alternative 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}
```

1. The status Signal.TRUE means that all the constraints imposed by the monitor have now been satisfied — there are no further monitoring rules active.

2. 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, e.g. in our first example monitoring that a should occur on every monitoring step.

3. 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, as in our second example monitoring that b should eventually occur.

4. The status Signal.FALSE means that the monitoring constraints have definitely been falsified and no further input can change the status.

5. The status Signal.UNKNOWN means that the monitor has been unable to resolve the status of monitoring against one of the above values. This will typically arise when two constraints are conjoined, for example one which is Signal.STILL_TRUE and the other is Signal.STILL_FALSE, see Section 6.6.4.

2.4.1 Using from AspectJ instrumentation

Here we exemplify using the above interface from within an AspectJ instrumentation of a Java application. The Java application will be the following program that we wish to satisfy the monitor defined by the rule system REPattern.

```
public class ExampleThree {

    static void a(){
        System.out.println("a called");
    }

    static void b(){
        System.out.println("b called");
    }

    static void c(){
        System.out.println("c called");
    }

    static void g(){
        System.out.println("g called");
    }

    static void end() {
        System.out.println("end called");
    }

    public static void main(String[] args) {

        a();
        b();
        c();
        g(); a(); b(); a(); b(); c();
        a();
        g(); a(); c();
        b();

        end();
    }
}
```

The aspect REPattern defines an instance of a RuleR monitor, using the constructor mentioned above, from the file "src\examples\REPattern.ruler" which contains the example ruler definitions in Figure 3. Four pointcuts are defined corresponding to calls of the methods a(), b(), c() and g() in the application ExampleThree.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 rules.RuleR;
```

```
import rules.RuleSystem.Signal;

public aspect REPattern {

  RuleR ruler =
    new RuleR("src\\examples\\REPattern", false);

  pointcut a() : call(void a());
  pointcut b() : call(void b());
  pointcut c() : call(void c());
  pointcut g() : call(void g());

  before() : a()
    if (ruler.dispatch("a") == Signal.FALSE){
      System.err.println("Symbol a incorrect");
      System.exit(0);
    }
  }
  before() : b(){
    if (ruler.dispatch("b") == Signal.FALSE){
      System.err.println("Symbol b incorrect");
      System.exit(0);
    }
  }
  before() : c(){
    if (ruler.dispatch("c") == Signal.FALSE){
      System.err.println("Symbol c incorrect");
      System.exit(0);
    }
  }
  before() : g(){
    if (ruler.dispatch("g") == Signal.FALSE){
      System.err.println("Symbol g incorrect");
      System.exit(0);
    }
  }
  before() : call(void end()){
    ruler.dispatchEnd();
  }
}
```

Running the main Java application ExampleThree.java as a Java/AspectJ combination produces the following console output. The first line reports that the monitor RE.REPattern is active. Then the main application output occurs until the monitoring fails when a c event occurs when a b event was expected.

```
Rule system RE.REPattern
a called
b called
c called
g called
a called
b called
a called
b called
c called
a called
g called
a called
```

```
Symbol c incorrect
```

3 Rule Improvements: rule optimizations - I

RuleR was designed initially as a low-level system into which one can compile other monitoring logics and descriptions for efficient interpretation. Indeed, we have given examples of translations from past and future time temporal logics, from regular, context free and context sensitive languages. However, from experimentation with the first prototype, it was decided to turn RuleR into more of a user-level system, provided there was no increase in run-time interpretation cost. We will use the term "core" RuleR to refer to the original low-level presentation of rule systems. In this section of the tutorial, we introduce the first of these additional non-core features.

3.1 On rule types

The interpretation given to rules in the core language is such that by default they are deactivated for the next monitoring step; the rules are, in a sense, single-shot. Hence, in order to ensure a rule, say R persists, some rule must be used to generate R on each step. The rule system REPattern in Figure 3 has several examples of this re-generation. Whilst this is straightforward for a "compiler" to generate, it becomes tedious for a user "programming" monitors directly in RuleR. Three types of rules were then introduced, each with a different notion of default persistence.

1. Rules categorised as always are, once active, always active, as the name implies. Such rules can be forcibly de-activated, however, by some other rule obligating the next monitoring step with its negation.

2. Rules categorised as state, once active, remain active until it is fired. A rule is fired when it is active and its condition evaluates to true. Again, state rules can be forcibly deactivated before they get fired.

3. Rules categorised as step have the basic "core" rule interpretation.

We re-present the REPattern rule system of Figure 3 in Figure 4 below. First of all note that the rules for reacting to the presence (and absence) of a g observation are now captured by the single **always** rule G. This rule is made active initially and, since there is no other rule that cancels it, will remain active for the duration of the monitoring.

The three rules Sa, Sc and Snac are replaced by the single **state** rule S. However, another extension has also been made that enables a rule name refer to a collection of rule bodies, as opposed to just one as in core RuleR. The new rule S has three rule bodies (in this particular case, corresponding to the bodies of the rules Sa, Sc and Snac). Clearly this reduces the amount of RuleR text that a user has to write. Rule S will count as being fired when any of its bodies are fired.

Similarly, the two rules Sb and Snb are replaced by the **state** rule T, which again is a multi-bodied rule.

4 Handling Parameters

In order for RuleR to maintain its simplistic "core" rule form but have sufficient expressive power, rule and data parameters were introduced. In this section, we explore these features and then some further syntactic extensions.

4.1 A simple counting example

The monitor specified in Figure 5 recognizes patterns (a+ b+ c+)+. There is nothing new in this monitor apart from the inclusion of the observation a as an initial requirement.

The exercise now is to modify the above monitor so that it accepts the context sensitive patterns $(a^n b^{2n} c^{3n})+$. The scheme employed in Figure 6 is to count the number of a observations, then once a b observation is made start counting down from twice the number, but remembering the number of c observations that will be required. The **state** rule A header is supplied with a formal argument x^2

```
ruler REPatternV2 {
    ruleIDs { G, S, T }
    observes { g, a, b, c }
    rules {

        always G {
            g -> S;
        }

        state S {
            a -> T;
            c -> Ok;
            !a, !c -> Fail;
        }

        state T {
            b -> S;
            !b -> Fail;
        }
    }
    initials { G }
    forbidden { S, T }
}

monitor {
    uses { RE: REPatternV2 }
    run RE .
}
```

Figure 4: A revised RuleR monitor for $\Box(g \rightarrow (ab) * c)$

Initially, the rule Terminal is activated. It requires an a observation to be met, failing otherwise, and then activates rule A with an actual argument of 1. The **state** rule A then counts the number of as met (note, it has to regenerate itself, since it is a **state** rule). If a b is encountered, then the monitor activates rule B with two arguments, the first being the count of bs that remain to accepted, and the number of cs that should follow on from the bs. The rule bodies of **state** rule B use relational expressions in the condition parts in order to determine the correct switching point.

Exercise 2 *Modify the rule schema definition of Figure 6 to recognise patterns of events of the form* $(a^{3n}b^{2n}c^n)+$.

4.2 Handling data in observations

The previous example shows how rules may be simply parameterized. Our next example uses data derived from an observation to drive the monitoring pattern. The example is a modification of that in Figure 3 where the observation g occurs with an integer argument used to specify how many blocks of a followed by b are to be seen, before the final c observation. Mixing temporal logic and regular expression notation, we require a monitor for $\forall n.\Box(g(n) \Rightarrow \bigcirc((ab)^n c))$. This is presented in Figure 7. The condition of the rule body of **state** rule G is now g(x). The identifier x is parsed as a binding occurrence. The condition will match against an observation g(10), for example, and bind the variable x to the integer value 10. In such a situation, the rule S in the consequent of G will be passed the value 10. The rule S itself has more cases to handle now in order to ensure that only the correct number of iterations of

[2] In the prototype version of RuleR no type information is supplied. It is assumed that arithmetic is undertaken on integers. However, in addition to integers (or rather ints), actual arguments may be Java Object references, or even "calls" of RuleR rules.

```
ruler ABCmonitor {
    ruleIDs { A, B, C }
    observes { a, b, c }
    rules {
        state A {
            b -> B;
            c -> Fail;
        }
        state B {
            c -> C;
            a -> Fail;
        }
        state C {
            a -> A;
            b -> Fail;
        }
    }
    initials { a, A }
    forbidden { A, B }
}
monitor {
    uses { ABC: ABCmonitor }
    run ABC .
}
```

Figure 5: Monitoring $(a + b + c+)+$

the block a b occur. In the next subsection, a further syntactic rule optimization will be presented, a kind of rule condition factorization. Before then, we present below the AspectJ instrumentation for this example to show how the parameter for observation g is extracted and passed in the dispatch method.

```
import rules.RuleR;
import rules.RuleSystem.Signal;

public aspect REPatternV3 {
  RuleR ruler =
    new RuleR("src\\examples\\REPatternV3", false);

  pointcut a() : call(void a());
  pointcut b() : call(void b());
  pointcut c() : call(void c());
  pointcut g(int x) : call(void g(int)) && args(x);

  before() : a() {
    if (ruler.dispatch("a") == Signal.FALSE){
      System.err.println("Symbol a incorrect");
      System.exit(0);
    }
  }
  before() : b() {
    if (ruler.dispatch("b") == Signal.FALSE){
      System.err.println("Symbol b incorrect");
      System.exit(0);
    }
  }
```

```
ruler CountingABC {
    ruleIDs { A, B, C, Terminal }
    observes { a, b, c }
    rules {
        state A(x) {
            a -> A(x+1);
            b -> B(2*x-1, 3*x);
            c -> Fail;
        }
        state B(x, y) {
            x>0, b -> B(x-1, y);
            x=0, c -> C(y-1);
            x!=0, c -> Fail;
            a -> Fail;
        }
        state C(y) {
            y>1, c -> C(y-1);
            y=1, c -> Terminal;
            a -> Fail;
            b -> Fail;
        }
        state Terminal {
            a -> A(1);
            !a -> Fail;
        }
    }
    initials { Terminal }
    forbidden { A, B, C }
}
monitor {
    uses { ABC: CountingABC }
    run ABC .
}
```

Figure 6: Monitoring $(a^n b^{2n} c^{3n})+$

```
  before() : c() {
    if (ruler.dispatch("c") == Signal.FALSE){
      System.err.println("Symbol c incorrect");
      System.exit(0);
    }
  }
  before(int x) : g(x) {
    if (ruler.dispatch("g", new Object[]{x}) == Signal.FALSE){
      System.err.println("Symbol g incorrect");
      System.exit(0);
    }
  }
  before() : call(void end()) {
    ruler.dispatchEnd();
  }
}
```

```
ruler REPatternV3 {
    ruleIDs { G, S, T }
    observes { g, a, b, c }
    rules {
        always G {
            g(x) -> S(x);
        }
        state S(x) {
            a, x>0 -> T(x);
            a, x<=0 -> Fail;
            c, x=0 -> Ok;
            c, x!=0 -> Fail;
            !a, !c -> Fail;
        }
        state T(x) {
            b -> S(x-1);
            !b -> Fail;
        }
    }
    initials { G }
    forbidden { S, T }
}
monitor {
    uses { RE: REPatternV3 }
    run RE .
}
```

Figure 7: A monitor for $\forall n.\Box(g(n) \multimap ((ab)^n c))$

4.3 Condition factorization: rule optimization II

The conditions of the rule bodies of rule S in Figure 7 have common terms, for example, the observation a in the first two rule bodies, the c in the next two. Furthermore, the arithmetic conditions in the first two are complementary, similarly for the second pair (with observation c). In a certain sense, the rules are encoding conditionals. Extracting common factors will clearly be more efficient for interpretation and whilst one could leave this to a rule optimizer to do we've introduced a factored rule form that enables the user to program this directly. In doing so, we also introduced an execution order for the branches from the factor (the rule bodies enclosed by {: and :}). Unlike "core" rules, only one branch of a factored rule may be fired. This provides another small optimization. We present the **state** rule S using factored rule bodies.

```
state S(x) {
    a {: x>0 -> T(x);
          -> Fail;
      :}
    c {: x=0 -> Ok;
          -> Fail;
      :}
    !a, !c -> Fail;
}
```

The new version contains two factored rule bodies, one starting with the common condition a and the other with c. Note that now it is not necessary to include the negation of x>0, respectively x=0, since this part could only be fired if the preceding branches fail to fire.

```
ruler REPatternV4 {
    ruleIDs { G, S, T }
    observes { g, a, b, c }
    rules {
        always G {
            g(x) -> S(x);
        }
        state S(x) {
            a {: x>0 -> T(x);
                 -> Fail;
              :}
            c {: x=0 -> Ok;
                 -> Fail;
              :}
            !a, !c -> Fail;
        }
        state T(x) {
            b -> S(x-1);
            !b -> Fail;
        }
    }
    initials { G }
    forbidden { S, T }
}
monitor {
    uses { RE: REPatternV4 }
    run RE .
}
```

Figure 8: A monitor for $\forall n.\Box(g(n) \multimap ((ab)^n c))$: with condition factorization

4.4 Using and parameterizing rule system schemas

Currently, an input file for RuleR may contain several rule system schema definitions. Furthermore, a rule system schema may depend upon, i.e. use, a previously defined schema. In particular, we have extended core rule system schema to include definitions such as below.

```
ruler Comb {
    ...
    uses { A: Always, E: Eventually }
    ...
}
```

The **uses** definition effectively imports named instances of the specified rule system schemas into the currently specified rule system schema. Assume that the schemas Always and Eventually are as defined in Figures 1 and 2, then the rule system schema Comb will have rules named A.Ra, A.Rna, E.Rb and E.Rnb in addition to any that are explicitly introduced by its **ruleIDs** definition. The imported rules are then available for direct use in the rule definitions of Comb. In the example below, the initially activated **always** rule R activates the rules Ra and Rna of the A instance of Always whenever a g observation is made.

```
ruler Comb {
    uses { A: Always, E: Eventually }
    ruleIDs { R }
    observes { g, h }
    rules {
      always R {
```

```
            g -> A.Ra, A.Rna;
            h -> E.Rb, E.Rnb;
        }
    }
    initials { R }
}
```

The activation of the pairs of rules from the A and E instances of Always and Eventually correspond to the initial rule activations of those schema, respectively. Such use is, in our view, the norm. We have therefore allowed users to "call" a rule system instance directly in a rule consequent, as below.

```
ruler Comb {
    uses { A: Always, E: Eventually }
    ruleIDs { R }
    observes { g, h }
    rules{
      always R {
            g -> A; // A is replaced by A's initial frontier
            h -> E; // similarly for E
      }
    }
    initials { R }
}
```

4.4.1 Parameterizing rule system schemas

Rule system schemas may be parameterized. A list of formal argument names may be given in the RuleR system header; these are, at present, un-typed, but actual arguments may be data or rule names/instances or observation instances. In Figure 9 we have re-defined the monitor Comb so that the rule system instances A and E may be parameterized. Consider the Always schema definition. Its header has a formal argument y. That formal argument is then used as an actual argument to the rule R in the **initials** definition. The **always** rule R is defined with a formal argument x, which is used in the conditions of Rs rule bodies. The eventual chain of substitutions from the rule body in Comb results with the A instance of the rule system schema Always requiring that the observation a occurs for ever after a g observation. Similarly, after an h observation, sooner or later, a b observation must be seen.

Restrictions... The above example might lead one to try writing a rule such as

```
    state R {
        g -> A(E(b));
    }
```

in order to obligate that as soon as a g occurs then the observation b should be seen infinitely often. **Currently such parametrization is not possible**.

4.5 Super rules: rule optimization III

As has been found useful in other specification formalisms, the introduction of hierarchy, though not necessary, can simplify matters. We have therefore introduced the notion of rule hierarchy in RuleR in which a rule can be defined to be super to a collection of other rules (or, looking in the opposite direction, a rule may extend, or be a sub-rule of, another). When a sub-rule is activated, then all of its super-rules are also activated. Similarly, when a super-rule is activated in the absence of any sub-rule activation, then a default sub-rule is activated as well. The following example illustrates the scheme. We assume a policy on a file system where:

1. an OPEN action on a file may occur if it is not already opened;
2. an OPEN action must occur before an attempt to READ or WRITE to a file;

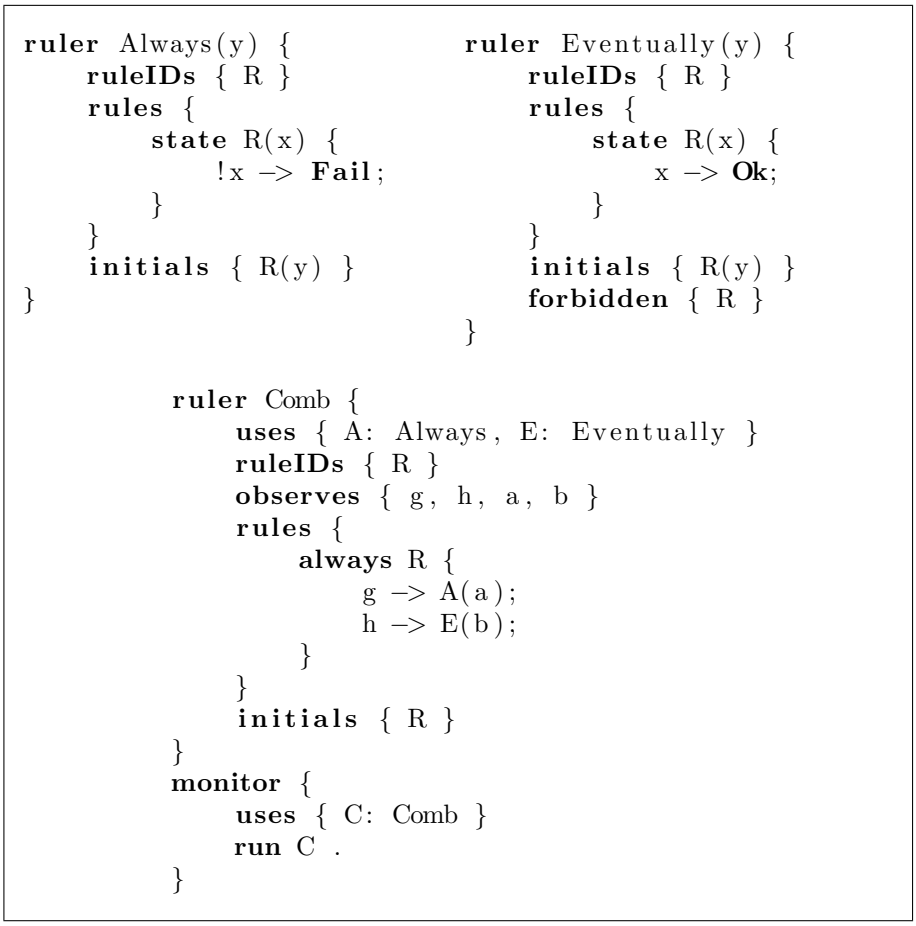

Figure 9: Parameterizing rule system schemas

3. once a file has been READ, then a WRITE action is forbidden, unless the file is closed and then reopened first;

4. a CLOSE action must occur on an opened file before a program terminates.

The rule system FileMonitor of Figure 10 uses a "super" rule named Opened; the list of rule names following the super keyword are the defined rule's sub-rules. To simplify the parsing of RuleR, the sub-rules must also specify their parent super-rule, which is given following the key word extends. Activation of any of Opened's sub-rules then automatically activates the super-rule Opened. Thus, an OPEN(f) observation on an unopened file f results in the monitor activating both the rule Unread(f) and the rule Opened(f) (for the appropriate file instance f). The Opened rule factors out the common rule bodies from its associated sub-rules. Clearly, there is a strong correlation between the notion of super-rule here and super-states in state chart formalisms.

Limitation... Super-rules are currently under development and hence, some features, such as giving priority of a sub-rule firing over a super-rule firing, are not implemented.

5 A real-time example

In all the examples we've considered so far, single events have stood as observations; indeed, the dispatch method used so far only takes a single event. However, the underlying RuleR technology, in fact, assumes that an observation (i.e. an input state) is a set of such events. We can thus use this feature to monitor, very easily, real-time events. We will assume that there is a time observation event that provides a time-stamp value to be associated with the other events occurring at that monitoring step.

```
ruler FileMonitor {
    ruleIDs { Start, Opened, Unread, Read, Modified }
    observes { OPEN, END, CLOSE, READ, WRITE }
    rules {
        always Start {
            OPEN(f), !Opened(f) -> Unread(f);
        }
        state Opened(f) super {Unread, Read, Modified} {
            END -> Fail;
            CLOSE(f) -> Ok;
        }
        state Unread(f) extends Opened {
            READ(f) -> Read(f);
            WRITE(f) -> Modified(f);
        }
        state Read(f) extends Opened {
            WRITE(f) -> Fail;
        }
        state Modifed(f) extends Opened {
            READ(f)  -> Read(f);
        }
    }
    initials { Start }
}
monitor {
   uses { FM: FileMonitor }
   run FM .
}
```

Figure 10: A simple file monitor using a super-rule

The time stamp value is the number of microseconds[3] since the creation of the RuleR monitor. For convenience, for RuleR linked to Java via AspectJ instrumentation, the RuleR constructor can switch on automated time-stamping, which then means that every dispatch method call automatically creates the time observation event. The example in Figure 11 illustrates how this can be used. The example concerns monitoring the behaviour of a set of automated doors in a building. There are openDoor, closeDoor and passDoor observation events. Once a door has been opened, it should be automatically closed within a certain time, unless something has passed through the door in the intervening period, which, in effect, resets the timer controlling how long the door remains open. Furthermore, only a limited number of doors may be open at any one time. An alarm should be raised as soon as either too many doors are open, or a door has not been closed within the required time limit.

The openDoor event is supplied with two values, one denoting the door and the second giving the limit on the time the door remains open (without passDoor activity). The initial rule Start carries two arguments, the number of doors currently open and the maximum number allowed to be open at any one time. The rule monitor's openDoor events. It is encoded using a factored rule body. The expected normal case, when the number of doors currently open is less than the maximum allowed, triggers the Opened state rule passing the door reference, its open time limit, and the time at which it was opened (obtained from the previous time event). This rule, also encoded using factored rule bodies, indeed a nested one, has three possibilities. The normal situation, when now$-$opentime $<$ timelimit, for closeDoor is to decrement the count of doors currently open, and for passDoor is to reset the time the door was opened to be the time associated with the current event. The abnormal situation, when time has run out, simply causes monitoring failure. Note that this particular failure may occur on *any* event; the rule is driven by the time stamp event. We will, in Section 6.4, indicate how an alarm might be

[3]A future version of RuleR, when long integers are implemented, will have the timing given in nanoseconds.

```
ruler DoorMonitor(maxopen) {
  ruleIDs { Start, Opened }
  observes { openDoor, closeDoor, passDoor, time }
  rules {
    state Start(x, max) {
      openDoor(door,timelimit), time(t)
        {: x<max -> Opened(door,timelimit,t), Start(x+1, max);
                 -> Fail;
        :}
      }
    state Opened(door, timelimit, opentime) {
      time(now)
        {: now-opentime < timelimit
            {: closeDoor(door), Start(x,max)
                                -> !Start(x,max), Start(x-1,max);
               passDoor(door) -> Opened(door, timelimit, now);
            :}
              -> Fail;
        :}
    }
  }
  initials { Start(0, maxopen) }
}
monitor {
    uses { D: DoorMonitor }
    run D(3) .
}
```

Figure 11: Handling real-time

raised, i.e. an application action provoked via RuleR.

As a small experiment, we ran the monitor against a simulation of door opening, passing and closing cycles, as encoded in the following Java main program.

```
public static void main (String [] args){

    CheckDoor me = new CheckDoor();

    Door doorA = me.new Door("A"),
         doorB = me.new Door("B"),
         doorC = me.new Door("C"),
         doorD = me.new Door("D");

    for (int n = 10000; n > 0; n=n-3){
        doorA.openDoor(10*n);
        doorB.openDoor(10*(n-1));
        doorC.openDoor(10*(n-2));
        doorC.passDoor();
        doorC.closeDoor();
        doorB.passDoor();
        doorB.closeDoor();
        doorA.closeDoor();
    }
    end();
}
```

The instrumentation in AspectJ was as follows

```
import rules.RuleR;
import rules.RuleSystem.Signal;

public aspect DoorMonitor {

  RuleR ruler =
    new RuleR("src\\examples\\doorMonitor", true);

  pointcut openDoor(CheckDoor.Door d, int limit) :
    call(* exp.CheckDoor.Door.openDoor(int)) && target(d) && args(limit);

  pointcut closeDoor(CheckDoor.Door d) :
    call(* exp.CheckDoor.Door.closeDoor()) && target(d);

  pointcut passDoor(CheckDoor.Door d) :
    call(* exp.CheckDoor.Door.passDoor()) && target(d);

  before(CheckDoor.Door d, int limit): openDoor(d, limit) {
    Signal s = ruler.dispatch("openDoor", new Object[]{d, limit});
    if (s==Signal.FALSE){
      System.err.println("Monitoring failed on openDoor for Door " + d.name);
      System.exit(0);
    }
  }
  before(CheckDoor.Door d): passDoor(d) {
    Signal s = ruler.dispatch("passDoor", new Object[]{d});
    if (s==Signal.FALSE){
      System.err.println("Monitoring failed on passDoor for Door " + d.name);
      System.exit(0);
    }
  }
  before(CheckDoor.Door d): closeDoor(d) {
    Signal s = ruler.dispatch("closeDoor", new Object[]{d});
    if (s==Signal.FALSE){
      System.err.println("Monitoring failed on closeDoor for Door "
                         + d.name + " for time limit " + d.limit);
      System.exit(0);
    }
  }
  before() : call(void end()) {
    if (ruler.dispatchEnd()==Signal.FALSE){
      System.err.println("Monitoring failed");
      System.exit(0);
    }
  }
}
```

Note that in order to obtain the time event to be associated with each dispatched openDoor, passDoor and closeDoor event, the third argument of the RuleR constructor is set true. Below, we provide an example of the monitoring output. The simulation provided decreasing time limits for the doors, starting at 100000 microseconds. The time limit became too small at 6.67 milliseconds. This gives a good indication of the overhead of the instrumentation and monitoring with this particular rule system since the application code cost was virtually nil.

```
Rule system D.DoorMonitor StepNo = 0
Rule system D.DoorMonitor StepNo = 5000
```

```
Rule system D.DoorMonitor StepNo = 10000
Rule system D.DoorMonitor StepNo = 15000
Rule system D.DoorMonitor StepNo = 20000
D.DoorMonitor's frontier has become vacuous: StepNo 24896
Monitoring failed on closeDoor for Door A for time limit 6670
```

6 Other extensions

6.1 Assert rules

We consider here an example of checking that an input sequence of events matches against a simple context free grammar. We will require that the input events are of the form lock^n unlock^n. for some $n \geq 0$. We will provide two encodings, both of which will use the use of rule continuations as arguments to rules in order to encode the context freeness of the languages using rules of regular appearance. The first encoding we provide is what we call a prescriptive encoding, in the sense that the required events appear only as obligations, i.e. in the consequents of the rules. The rule system schema uses two rules

```
ruler SimpleCFL {
  ruleIDs { L, U, E }
  observes { lock, unlock }
  rules {
    L(c): -> lock, L(U(c)) # unlock, c;
    U(c): -> unlock, c;
    E:    -> Fail;
  }
  initials { lock, L(U(E)) | E }
  forbidden { L, U }
}
monitor {
  uses { S: SimpleCFL }
  run S .
}
```

Figure 12: A simple context free example

L and U that both are parameterized by a "continuation rule". The rule L obligates that either (i) the next monitoring step contains a lock event, in which case the rule is reactivated but with a continuation argument that will invoke the rule U with existing argument c, or (ii) the next monitoring step contains a unlock event, and the rule represented by the argument c is activated. The symbol # is used for choice for reasons we don't wish to bore the reader with, however, in future systems, this will be brought in line with the more usual symbol | (used in logical expressions and in the initials declaration). The rule U requires that the next monitoring step contains an unlock event and activates the rule's argument. Initially, it is required that either a lock event occurs in which case the rule L(U(E)) is activated, or the rule E is activated. The latter rule is used to indicate the point at which termination is expected; it forces failure if the end of input doesn't occur. The forbidden rule declaration, furthermore, indicates that monitoring should fail if rules L or U remain active.

Figure 13 contains a first version of the second, conditional, encoding. This version now uses four rules. S is an initial rule (along with E). It checks the input for a lock event; if present, then the first rule body of S activates the rule L with rule argument E. If a lock event is not present, then the rule fails, which in this deterministic system will cause failure of the rule system. Consider now the application of the rule L. It checks for either a lock event, in which case the rule is reactivated with rule argument U(c) where c was the original argument, and it checks for an unlock event, in which case the rule's argument is activated, and if neither lock nor unlock occur it fails (which again will cause the rule system to fail). The rule U wants to find an unlock event; if successful then the rule's argument is activated, and if unsuccessful then the rule fails. The rule E is used in the same way as in the first

```
ruler SimpleCFLV1 {
  ruleIDs { S, L, U, E }
  observes { lock, unlock }
  rules {
    step S {
      lock -> L(E);
      !lock -> Fail;
    }
    step L(c) {
      lock -> L(U(c));
      unlock -> c;
      !lock, !unlock -> Fail;
    }
    step U(c) {
      unlock -> c;
      !unlock -> Fail;
    }
    step E {
        -> Fail;
    }
  }
  initials { S | E }
  forbidden { S, L, U }
}
monitor {
  uses { S: SimpleCFLV1 }
  run S .
}
```

Figure 13: An alternative encoding of a simple context free example

encoding. The system is initialized with a choice of either rule S or rule E, thus allowing empty input sequence. In this encoding, however, the three rules S, L and U are forbidden to be active at the end of the monitored sequence.

Note that in each of the rules S, L and U we "force" failure if the expected events are not present. Although this is not too tedious to write out in this situation, for more complex patterns it will become a bind. We have therefore introduced an optimization into the rule engine that will eliminate the need to explicitly incorporate the forcing of failure. We have, as an experiment, introduced a new rule form, called an assert rule. It is supplied with a collection of rule names. Its semantics is such that on each monitoring step, at least one of the asserted rules must "fire", i.e. the consequent of the rule is applied. Such an optimization clearly reduces the number of rule bodies that must be checked on each step. Figure 14 provides a revised description of this example using an **assert** rule.

6.2 Succeed rules

A RuleR monitor can yield one of five signal values for its current status. So far, we have used the key words **Ok** and **Fail** to stand (alone) in a rule consequent to force success or failure, by making the future frontier either a single set containing an empty set, i.e. no constraints, or an empty set of choices, hence denoting falsity. Let us consider again the lock/unlock example of Figure 14. This monitor can clearly generate a monitoring status of signal value FALSE, either through activation of the assert rule if one of the specified rules S, L or U is not applied, or through the application of rule E. However, can the monitor ever generate a status of signal value TRUE? One might reasonably expect that to be the result of monitoring n lock events followed by n unlock events. For the monitor as written, that is not the case. At no point, does the future frontier reduce to the single set containing the empty set; indeed, the point at which success should occur will have the rule E in the future frontier. To overcome this

```
ruler SimpleCFLV2 {
  ruleIDs { S, L, U, E }
  observes { lock, unlock }
  rules {
    step S {
      lock -> L(E);
    }
    step L(c) {
      lock -> L(U(c));
      unlock -> c;
    }
    step U(c) {
      unlock -> c;
    }
    step E {
        -> Fail;
    }
    assert { S, L, U }
  }
  initials { S | E }
  forbidden { S, L, U }
}
monitor {
  uses { S: SimpleCFLV2 }
  run S .
}
```

Figure 14: Using the assert rule

issue, we extended core RuleR to include a **succeed** rule that is supplied with a list of rule identifiers. If, on any monitoring step, one of the "succeed' rules is present in the future frontier, i.e. the obligations, then the monitoring yields a status signal TRUE. Figure 15 lists an updated version of the lock/unlock example.

6.3 Watch rules

6.4 Output rules

The dispatch methods of the RuleR class return a status value of the four-valued enumerated type RuleSystem.Signal. The terminating status value is also implicitly dispatched to the output file, passed to the RuleR constructor, as an observation, i.e. one of status(0), status(1), status(2), or status(3), corresponding to the Signal values FALSE, TRUE, STILL_FALSE and STILL_TRUE (String constants are currently not implemented in RuleR).

RuleR has been extended to allow the user to define and output other "reporting" information as observations. The observation names to be possibly output must be declared as such. Then the RuleR interpreter checks at the end of each monitoring step whether any of those observation names appear as obligations in the monitoring state; if so, the observations are also dispatched to the output. For example, in Figure 16 we have modified the door monitor example of Figure 11 to output an alarm observation and then fail in the subsequent monitoring step. Note how we have introduced a specific rule FailNext, which is activated when the alarm observation is also obligated. The rule FailNext simply forces failure in the next monitoring state.[4]

[4]This is currently done this way as a consequence of the way output events are currently implemented; the alarm observation is obligated for the next monitoring step and therefore failure, which corresponds to a vacuous obligated rule and observation monitoring state, is not possible.

```
ruler SimpleCFLV2 {
  ruleIDs { S, L, U, E }
  observes { lock, unlock }
  rules {
    step S {
      lock -> L(E);
    }
    step L(c) {
      lock -> L(U(c));
      unlock -> c;
    }
    step U(c) {
      unlock -> c;
    }
    step E {
        -> Fail;
    }
    assert { S, L, U }
    succeed { E }
  }
  initials { S | E }
  forbidden { S, L, U }
}
monitor {
  uses { S: SimpleCFLV2 }
  run S .
}
```

Figure 15: Using the succeed rule

6.5 Internal Observations

Core RuleR has also been extended to support internal, or local, observations. In a sense, these are akin to rules without any bodies and can be used as memory devices — indeed, it is, effectively, such internal rules that are used to encode past time logic values.

```
ruler AlarmedMonitor(maxopen) {
  ruleIDs { Start, Opened, FailNext }
  observes { openDoor, closeDoor, passDoor, time }
  locals { alarm }
  ...
  ...
  outputs { alarm }
}
monitor {
  uses { A: AlarmedMonitor }
  run A(3) .
}
```

In the above, alarm has been declared as local "observation". There are a few subtleties here. The first point to note is that on creation of the monitor A of rule system schema AlarmedMonitor all rule and local names of the schema are prefixed by the identifer A in order to create unique instances. This doesn't happen for the observation names since they are, in effect, globally defined names. Thus, the name of the output observation from A will be A.alarm. The second point to note is that the presence of such local observations in the "obligated" monitoring state, i.e. the state computed for the next monitoring event, can never be in conflict with input observations. This is not the case, for example, for alarm declared in

```
ruler AlarmedMonitor(maxopen) {
  ruleIDs { Start, Opened, FailNext }
  observes { openDoor, closeDoor, passDoor, time, alarm }
  rules {
    state Start(x, max) {
      openDoor(door,timelimit), time(t)
        {: x<max -> Opened(door,timelimit,t), Start(x+1, max);
                 -> alarm, FailNext;
        :}
      }
    state Opened(door, timelimit, opentime) {
      time(now)
        {: now-opentime < timelimit
            {: closeDoor(door), Start(x,max)
                                -> !Start(x,max), Start(x-1,max);
               passDoor(door) -> Opened(door, timelimit, now);
            :}
              -> alarm, FailNext;
        :}
    }
    state FailNext {
          -> Fail;
    }
  }
  initials { Start(0, maxopen) }
  outputs { alarm }
}
monitor {
    uses { D: AlarmedMonitor }
    run D(3) .
}
```

Figure 16: Setting an alarm

the **observes** declaration as in Figure 16; indeed, in that situation, if the rules obligate alarm then the next input must contain it, otherwise failure would occur.

6.6 Monitor expressions

So far, all the RuleR examples we have presented define a monitor created from a single rule system schema instantiation. However, the RuleR prototype allows users to define monitors created from a composition of rule system instantiations. Indeed, currently as an experiment, RuleR supports a *chaining* composition, three forms of *conditional* composition, and a *looping* composition.

6.6.1 Chaining monitors together

Given the ability to output observation events, monitors can be viewed as event stream transducers, transforming an input event stream into an output event stream. All but the last monitor we've illustrated transformed the input stream of observation events to an output stream of status values (typically just the status value from the final monitoring step). The example of Figure 16, on the other hand, produced both an alarm output observation event as well as an implicit status event. However, so far, all such output observations have been sent to a file. In some circumstances it may be appropriate for another monitor to "monitor" the output stream of the first. Indeed, this can be viewed as a helpful design strategy for complex monitoring situations; it is supporting a form of abstraction.

Consider the following "toy" scenario. A Java application uses (recursive) tree structures for storage

```
ruler Trace {
  ruleIDs { Top, Stack }
  observes { call, return }
  locals { result }
  rules {
    state Top{
      call(tree,size) -> Stack(tree, 1, 1, size);
    }
    state Stack(tree, level, max, size){
      call(x, y) -> Stack(tree, level+1, max+1, size);
      return(x, b)
        {: level=1
             {: b=1 -> result(t, max, size), Top;
                    -> Top;
             :}
                -> Stack(tree, level-1, max, size);
        :}
    }
  }
  initials { Top }
  outputs { result }
}
```

Figure 17: Tracing calls and returns

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 will define two monitors. The first will determine the maximum depth reached in the associated structure to find an item and report this information to a second monitor that will collect and analyse statistics of use. It will make reports when undesirable performance occurs[5]. The Trace monitor given in Figure 17 assumes that, given an initial call, the subsequent calls and returns relate to recursive invocations of the instrumented application's search method. It thus matches calls and returns and keeps track of the maximum depth reached. When a top-level return event occurs, a result is output if the search call was successful. The result passes three items of data, the tree reference, the maximum depth and the size of the tree. The events output by the Trace rule system schema are to be consumed by a monitor based on the rule system StatsGatherer of Figure 18.[6] The schema has two percentage arguments G and B which are used to categorize the ratio of depth of search over size of tree (as a percentage) 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 is of type **always** but it only starts tracking results for a particular tree if it is not tracking the tree already. The variables t, m and s are bound by matching T.result(t, m, s) against a (grounded) observation T.Result(...). Thus the variable t appearing in the expression !Track(t, x, y) is already bound, however, the other two variables will be bound by matching against an appropriate observation event.[7] 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

[5] It is then of interest to see how one can use the monitoring output from RuleR to cause action in the application. We will look very briefly at this in Section 7.

[6] The limitations of the current prototype have meant we have had to cheat and be non-compositional with the specification — the observation "T.result" is the name of the observation "result" from a "T" instance of the schema "Trace". The observation name "result" should really be passed as a schema parameter.

[7] In a future version of RuleR, variables will be statically typed and hence it will become much clearer which are binding occurrences.

```
ruler StatsGatherer(G, B) {
  ruleIDs { Start, Track }
  observes { T.result }
  locals { report }
  rules {
    always Start {
      T.result(t, m, s), !Track(t,x,y)
        {: m*100 < G*s -> Track(t,1,0);
           m*100 > B*s -> Track(t,0,1);
                       -> Track(t,0,0);
        :}
    }
    state Track(tree, Gs, Bs){
      T.result(tree, m, s)
        {: m*100 < G*s -> Track(tree, Gs+1, Bs);
           m*100 > B*s -> Track(tree, Gs, Bs+1);
                       -> Track(tree, Gs, Bs);
        :}
      (Bs != 0) & (Gs !=0 ) & (2*Bs > 3*Gs) -> report(Bs, Gs);
    }
  }
  initials { Start }
  outputs { report }
}
```

Figure 18: Gathering and reporting statistics

```
monitor {
  uses { T: Trace, S: StatsGatherer }
  run (T >> S(33, 50)) .
}
```

Figure 19: The chained monitor

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. An example is shown in Figure 19.

6.6.2 Conditional monitors

RuleR also supports the conditional composition of monitor expressions. There are currently three forms of conditional monitor.

1. The monitor expression (M1 −T> M2) denotes an if...then form. It has the following semantics. The monitor M1 receives the monitoring input up to and including the observation input for which M1 returns a status signal value true, after which the monitoring input passes directly to the monitor M2. If there is no such input to cause M1 to return a status true signal then M1 continues to receive the monitoring input.

   By way of simple example, let us combine the Eventually and Always monitors from Figure 9 in a way that first uses E(p) (an instance of Eventually supplied with observation p), then, as soon as the monitor returns status true, continues with, i.e. redirects the input to, A(q) (an instance of Always supplied with observation q). Of course, we could have quickly written a rule system schema to achieve the same effect, however, our purpose is to show, and experiment with, different forms of monitor composition.

```
monitor {
    uses { A: Always, E: Eventually }
    locals { p, q }
    run  ( E(q) -T> A(p) ) .
}
```

Figure 20: If ... Then ... monitoring

2. The monitor expression (M1 −F> M2) denotes an if...else form. It is the obvious counterpart to the above conditional monitor. The monitor M1 will continue to receive monitoring input up to and including the observation input for which M1 returns a status signal value false, after which the monitoring input switches over to the monitor M2. If there is no such monitoring input to cause M1 yielding status value false then M1 continues to receive the monitoring input.

```
monitor {
    uses { A: Always, E: Eventually }
    locals { p, q }
    run  ( A(q) -F> E(p) ) .
}
```

Figure 21: If ... Else ... monitoring

The example given in Figure 21 first monitors with an A instance of the Always rule system supplied with observation q. If it fails, i.e. yields a status signal value false, then monitoring continues with an E instance of the Eventually rule system supplied with observation p.

Remark There is a semantic subtlety with this construct. The monitoring input that causes failure of the condition monitor whilst used by that monitor is not consumed by the monitor, the else monitor continues with the same monitoring input event. This is not the case for monitoring input that causes successful completion; in that case, the input is consumed and the "then" monitor continues with the next monitoring input. Thus, if one were to attempt to monitor, say, q^+p^+ via the monitor expression (A(q) −F> A(p)) an input sequence qqp would satisfy the monitor and an input sequence qqrp would cause failure. The first sequence is accepted since the input event p, which causes failure of A(q) is reused as the first input to the second monitor. For the second sequence, the r event, which caused failure of A(q), is passed to the second monitor A(p) and also causes failure.[8] Note also that, currently, for a monitor such as (A(q) −F> A(p)) running over the sequence qqqpp, the implementation generates two status value signals, status(false) from the first monitor and status(true) from the second (of course, if the RuleR interface method dispatchEnd() is then called a final status signal status(true) is generated).

3. The monitor expression (M1 ? M2 : M3) denotes an if...then...else form. If the monitor M1 returns a status signal value true then the monitoring input is switched to M2. If the monitor M1 returns a status signal value false then the monitoring input is switched to M3. Otherwise, the monitoring input stays with M1 Consider the small example in Figure 22. The rule system schema Test(x, y) succeeds if it receives the observation substituted for the variable x and fails if the observation substituted for y is received. The example monitor uses an instance of the Test rule system, where x is observation a and y is observation b, to subsequently check for either the eventual occurrence of p observation (the then case) or for b observations always occurring.

6.6.3 Looping monitors

Monitor expressions may also be formed using two forms of looping combinator:

[8] Clearly this subtlety can be removed but then other oddities exist.

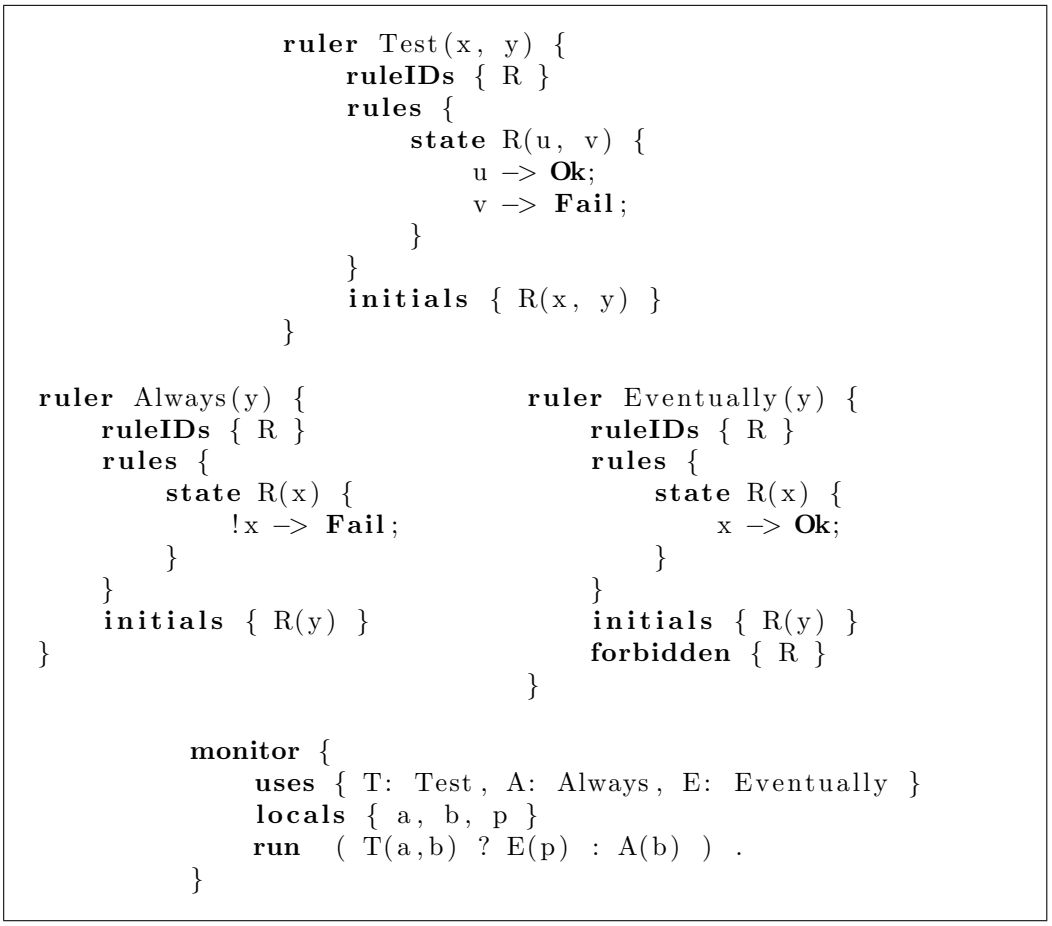

Figure 22: If ... Then ... Else ... monitor expression

(M ∗) : loop on success from monitor expression M;

(M ∗!) : loop on failure of monitor expression M.

To illustrate the use of loopy monitor expressions, consider the rule system schema Triple as presented in Figure 23. If we monitor using an instance of Triple such as below

```
monitor {
  uses { T: Triple }
  observes { a, b, c }
  run T(a, b, c) .
}
```

a status signal value true will result as soon as a prefix of an observation input sequence matches a single a followed by a b followed by a c (or has no such a, b or c observations), filtering out any other observation events. Suppose we wish to match on repeated patterns of abc (filtered) observation sequences. Clearly we could modify the rule system schema to reflect such looping by changing the subrule z −> **Ok** of state rule C to be the subrule z −> A. Of course, this is not in the spirit of re-using specification (code). However, we can use the loop on success monitor expression to good effect for this.

```
monitor {
  uses { T: Triple }
  observes { a, b, c }
  run ( T(a, b, c)* ) .
}
```

```
ruler Triple (x, y, z) {
    ruleIDs { A, B, C }
    rules {
        state A {
            x -> B;
            y -> Fail;
            z -> Fail;
        }
        state B {
            y -> C;
            x -> Fail;
            z -> Fail;
        }
        state C {
            z -> Ok;
            x -> Fail;
            y -> Fail;
        }
    }
    initials { A }
    forbidden { B, C }
}
```

Figure 23: A simple pattern recognizer

The (informal) semantics is that as soon as the monitor T(a, b, c) returns a status signal true then the monitor is restarted. However, as soon as the monitor T(a, b, c) fails, i.e. returns a status signal value false, the looping monitor also fails. Thus, given an input sequence, say, g a b g g c g g a b c g g g c g the looping monitor will fail on the third c observation, since an a was expected.

Now consider the following monitor.

```
monitor {
  uses { T: Triple }
  observes { a, b, c }
  run ( ( T(a, b, c)* ) –F> ( T(c, b, a)* ) ) .
}
```

This will first accept repeated patterns of a b c (possibly interspersed with other non a, b and c observations) and then accept repeated patterns of c b a events (also possibly interspersed with other observation events). For example, given the input sequence a g b c g a b c g c b a g c g b a the monitor will return a status signal value still_true for the final a observation. Whereas, had there been a further c observation, the monitor would have returned a status signal still_false (as a complete pattern is required by the Triple rule system schema.

Let's make one further iteration on this. Suppose we now wish to repeat the whole of the above pattern. That is, assume the second pattern accepted by the monitor T(c, b, a) fails, we now wish to restart with the original monitor. We can not now use the "loop on success" combinator, since success is used to keep looping on the second monitor. Indeed, this demonstrates a requirement for the "loop on failure" combinator.

```
monitor {
  uses { T: Triple }
  observes { a, b, c }
  run ( (( T(a, b, c)* ) –F> ( T(c, b, a)* ))*! ) .
}
```

An input observation sequence a b c c b a a b c c b a is now happily not failed. The second c observation, which causes failure of (T(a, b, c)*), switches to monitoring with (T(c, b, a)*). The third

a observation, which causes failure of (T(c, b, a)*), will now cause monitoring to be switched back to the original monitor expression, and hence, via the conditional, into (T(a, b, c)*)

Whilst the above example specification will monitor the desired observation sequences correctly, it also accepts sequences that one might believe should not be accepted. For example, an input sequence a b c c c c b a a b c does not cause failure of the overall monitor.

6.6.4 Parallel monitors

It is natural to extend the range of monitor expressions to include parallel composition even though the effect of parallel composition of rule system schemas can be effected through at least two other ways. Assume that we have rule schema definitions available for recognizing the languages $a^n b^n c^m$ and $a^m b^n c^n$ for $n > 1, m \geq 0$, i.e. AnBnCm and AmBnCn, the parallel monitor expression given below will accept input traces of the context sensitive language $a^n b^n c^n$ for $n > 0$.

```
monitor {
    uses { P: AnBnCm, Q: AmBnCn }
    observes { a, b, c }
    run ( P(a,b,c) || Q(a,b,c) ) .
}
```

There are, however, several possible semantics that can be given for this construct. RuleR currently gives it the following interpretation. An input event for the parallel expression is sent, in turn, to both sub monitor expressions P(a,b,c) and Q(a,b,c). The output of the monitor expression is then obtained by constructing the union of the non-status output events from the sub monitor expressions (potential conflicts are currently ignored) and taking the conjunction of any associated status results that may also be output. The signal value returned by the dispatcher is also the conjunction of the signal values from the individual sub monitors. It is the use of this semantics that requires the status signal value UNKNOWN. For example, one sub monitor may return signal value STILL_FALSE but the other yields STILL_TRUE. The state of monitoring is actually unknown, further input may change it to TRUE, may change it to FALSE, or just remain the same. For the sake of completeness, Figure 24 lists the two rule system schemas for the above monitor. Note that we have used a continuation-based programming styles for the rule schema definitions. The schema AnBnCm uses continuation arguments, building up the rule expression B (....(C)....) to remember how many b events are required to match the a events, when the correct number of b events have been consumed, the C rule will be made active. On each monitoring step, one of the specified hour rules must be active and the system will fail if termination of input occurs before c events start appearing. The rule system schema AmBnCn is written in a similar way, however, as illustrated in Section 6.2, a rule End has been introduced to mark the successful termination point — hence its appearance in the **succeed** rule.

An alternative specification that achieves the same monitoring effect, but constructed from "smaller" rule schema systems (recognizing simpler patterns), might be as below.

```
monitor {
    uses { S1: Symbol, S2: Symbol, P1: AnBn, P2: AnBn }
    observes { a, b, c }
    run ( ( ( S1(a)* ) -F> P1(b,c) ) || ( P2(a,b) -T> ( S2(c)* ) ) ) .
}
```

The rule schema Symbol recognizes a single occurrence of a symbol given as argument to the schema. The rule schema AnBn recognizes the pattern $u^n v^n$ for symbols u and v also passed as arguments. The left sub-monitor expression S1(a)* -F> P1(b,c) will recognize the $a^m b^n c^n$; the monitor expression S1(a)* will yield a status value FALSE when a doesn't match the input, which is then switched to the monitor expression P1(b,c) to match the context free pattern $b^n c^n$. The right sub-monitor expression matches against $a^n b^n c^m$ in a similar way, except one should note that this time the switch occurs from P2(a,b) to S2(c)* when the former monitor expression yields a signal value TRUE. Again, for completeness, the associated rule schema definitions are presented in Figure 25.

A question of termination

```
ruler AnBnCm(u,v,w) {
    ruleIDs { S, A, B, C }
    rules {
        step S {
            u -> A(C);
        }
        step A(x) {
            u -> A(B(x));
            v -> x;
        }
        step B(x) {
            v -> x;
        }
        step C {
            w -> C;
        }
        assert { S, A, B, C }
    }
    initials { S }
    forbidden { S, A, B }
}
```

```
ruler AmBnCn(u,v,w) {
    ruleIDs { A, B, C, End }
    rules {
        step A {
            u -> A;
            v -> B(End);
        }
        step B(x) {
            v -> B(C(x));
            w -> x;
        }
        step C(x) {
            w -> x;
        }
        step End {
              -> Fail;
        }
        assert { A, B, C }
        succeed { End }
    }
    initials { A }
    forbidden { A, B, C }
}
```

Figure 24: AnBnCm and AmBnCn rule system schema

```
ruler Symbol(s) {
    ruleIDs { S }
    rules {
        step S {
            s -> Ok;
        }
        assert { S }
    }
    initials { S }
}
```

```
ruler AnBn(u,v) {
    ruleIDs { S, A, B, End }
    rules {
        step S {
            u -> A(End);
        }
        step A(x) {
            u -> A(B(x));
            v -> x;
        }
        step B(x) {
            v -> x;
        }
        step End {
            -> Fail;
        }
        assert { S, A, B }
        succeed { End }
    }
    initials { S }
    forbidden { S, A, B }
}
```

Figure 25: Symbol and AnBn rule system schema

7 Incorporating Action

7.1 The interface class: RuleMonitorInterface

The prototype RuleR implementation incorporates a lower-level interface for rule system monitors.

```
import terms.State;

public interface RuleMonitorInterface {

    public void setOutput(RuleMonitorInterface output);

    public RuleMonitorInterface getOutput();

    public void initialiseRun(RuleMonitorInterface output);

    public RuleSystem.Signal stepRun(State obs);

}
```

The dispatch method we have introduced and used in the aspect instrumentation calls the lower-level stepRun method of the rule monitor. The latter method is passed a State object; informally this is a collection of observations for the particular monitoring step. The dispatch method, on the other hand, is passed just one observation event, in order to provide a simple instrumentation and monitoring mechanism using AspectJ. The lower-level interface also provides methods for setting the output of a monitor. For example, for a monitor built from a single rule system schema, then the output is directed to the file fileName.output where fileName was the string passed as first argument to the RuleR constructor, which then constructs an "event acceptor" (that implements the rule monitor interface) and sends the events passed to it via the stepRun method to the associated file. For chained monitors, the first monitor has the second monitor in the chain passed to it as its "output".

7.2 An example of an "action" monitor

Previously we have indicated the possibility for triggering action in the monitored (application) program from a RuleR monitor. We now describe a simple scheme for achieving this in a limited fashion. In outline, we need to create a special "action" monitor that is passed to the RuleR constructor in order to be plugged in as an output. This action monitor then converts the output events it receives via the stepRun method to be calls of methods in the associated monitored application. For example, in the door monitor system of Figure 16, one might wish to ensure ALL open doors are immediately closed on the raising of an alarm, i.e. a lock down procedure.

To Be Completed!

Acknowledgement

The authors are very grateful to Djihed Afifi for his contribution in creating a very first parser for the core RuleR system that then enabled initial experimentation and easy extension for the developments outlined in this tutorial. Any problems with the parser, however, should not be attributed to Djihed, only to HB.

References

[1] H. Barringer, D.E. Rydeheard and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. Proc. of the *7th International Workshop, RV2007, Revised Selected Papers*, Vancouver, Canada, Vol. 4839, Springer-Verlag, 2007, pp 111–125.

[2] H. Barringer, D.E. Rydeheard and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR— Extended Verion. *submitted for journal publication*, 2008.

A Answers to exercises

Exercise 1 *Modify the rule system REPattern so that the trigger observation g pattern is only sought again after successful matching of the pattern (ab)∗c rather than on every monitoring step.*

The rules Rg and Rng are removed from the consequent of rule Rg and then used to replace the **Ok** consequent of the rule Sc. Thus, the rules that react to the presence of a g event, i.e. Rg and Rng, are reactivated when a c has occurred correctly.

```
ruler ExerciseOne {
    ruleIDs { Rg, Rng, Sa, Sb, Sc, Snb, Snac }
    observes { g, a, b, c }
    rules {
        Rg: g -> Sa, Sc, Snac;
        Rng: !g -> Rg, Rng;

        Sa: a -> Sb, Snb;
        Sc: c -> Rg, Rng;
        Snac: !a, !c -> Fail;

        Sb: b -> Sa, Sc, Snac;
        Snb: !b -> Fail;
    }
    initials { Rg, Rng }
    forbidden { Sa, Sb, Sc }
}

monitor {
    uses { RE: ExerciseOne }
    run RE .
}
```

Exercise 2 *Modify the rule schema definition of Figure 6 to recognise patterns of events of the form* $(a^{3n}b^{2n}c^{n})+$.

The state rule A has been replaced by three rules, A0, A1 and A2, which together recognize a sequence of 3 a events. Thus, for $3n$ copies of a, the rule A0 will have been active n times when the system starts with Terminal.

```
ruler ExerciseTwo {
    ruleIDs { A0, A1, A2, B, C, Terminal }
    observes { a, b, c }
    rules {
        state A0(x) {
            a -> A1(x);
            b -> B(2*x-1, x);
            c -> Fail;
        }
        state A1(x) {
            a -> A2(x);
            !a -> Fail;
        }
        state A2(x) {
            a -> A0(x+1);
            !a -> Fail;
        }
        state B(x, y) {
            x>0, b -> B(x-1, y);
            x=0, c -> C(y-1);
            x!=0, c -> Fail;
            a -> Fail;
        }
        state C(y) {
            y>1, c -> C(y-1);
            y=1, c -> Terminal;
            a -> Fail;
            b -> Fail;
        }
        state Terminal {
            a -> A1(0);
            !a -> Fail;
        }
    }
    initials { Terminal }
    forbidden { A0, A1, A2, B, C }
}
monitor {
    uses { E2: ExerciseTwo }
    run E2 .
}
```