super logics
from Eagle to RuleR

CS 119

the search for a unifying monitoring logic
Phoenix has landed!
sister vehicle
Mars Polar Lander crashed in 1999

1999

crash believed to be caused by software error.
Mars Polar Lander
landing sensor activated too early

- Normally, the shake of a touch down would signal that engines should be shut down.
- However, shake of legs opening could cause the same effect in some cases. It was known.
- System was designed to ignore such shakes above 40 feet where legs were to open.
- System above 40 feet correctly ignored landed-flag, but flag was not reset to false, and triggered engine shut-off as soon as 40 feet were reached.
there is a unifying and practical logic

there is no such!
a unifying logic

- state machines
- context free grammars
- past time temporal logic
- future time temporal logic
- regular expressions
From Eagle to RuleR

- two attempts to define a super logic:
  - **Eagle**: recursion-based
    - inspired by mu-calculus
    - very convenient logic
    - difficult to implement
  - **RuleR**: rule-based
    - inspired by rule systems from AI and earlier TL work
    - less convenient logic at first sight
    - easy to implement
    - has potential for being basis for convenient logic
Eagle

\[ p \land @ q \]

temp. logic

\[ \lambda \]

functions

recursion

Howard Barringer
Allen Goldberg
Klaus Havelund
Koushi Sen
temporality + recursion

• propositional logic + two main temporal connectives:
  – Next: \( @F \)
  – Previous: \( #F \)

• Recursion:
  \[
  \text{safe always}(\text{Term } p) = p \land @\text{always}(p)
  \]
  \[
  \text{live eventually}(\text{Term } p) = p \lor @\text{eventually}(p)
  \]
  \[
  \text{live previously}(\text{Term } p) = p \lor #\text{previously}(p)
  \]
example:

```plaintext
Mon M = always(y>0 -> 
    (previously(x>0) \Eventually(z>0))) .
```

syntax:

```plaintext
S ::= dec D obs O
D ::= R*
O ::= M*
R ::= \{max | min \} N(T_1 \,x_1,\ldots,T_n \,x_n) = F
M ::= N = F
T ::= Form | java primitive type
F ::= java expression | True | False | \neg F | F_1 \land F_2 | F_1 \lor F_2 | F_1 \to F_2 |
    \Box F | \Diamond F | F_1 \cdot F_2 | N(F_1,\ldots,F_n)
```
semantics

\[
\sigma, t \models_D \text{exp} \quad \text{iff} \quad 1 \leq t \leq |\sigma| \text{ and } \text{evaluate}(\text{exp})(\sigma(t)) = true \\
\sigma, t \models_D \text{true} \\
\sigma, t \not\models_D \text{false} \\
\sigma, t \models_D \neg F \quad \text{iff} \quad \sigma, t \not\models_D F \\
\sigma, t \models_D F_1 \land F_2 \quad \text{iff} \quad \sigma, t \models_D F_1 \text{ and } \sigma, t \models_D F_2 \\
\sigma, t \models_D F_1 \lor F_2 \quad \text{iff} \quad \sigma, t \models_D F_1 \text{ or } \sigma, t \models_D F_2 \\
\sigma, t \models_D F_1 \rightarrow F_2 \quad \text{iff} \quad \sigma, t \models_D F_1 \text{ implies } \sigma, t \models_D F_2 \\
\sigma, t \models_D \bigcirc F \quad \text{iff} \quad t \leq |\sigma| \text{ and } \sigma, t + 1 \models_D F \\
\sigma, t \models_D \bigotimes F \quad \text{iff} \quad 1 \leq t \text{ and } \sigma, t - 1 \models_D F \\
\sigma, t \models_D F_1 \cdot F_2 \quad \text{iff} \quad \exists j \text{ s.t. } t \leq j \leq |\sigma| + 1 \text{ and } \sigma^{(1, j-1)}, t \models_D F_1 \text{ and } \sigma^{[j, |\sigma|]}, 1 \models_D F_2 \\
\left\{ \begin{array}{l}
\text{if } 1 \leq t \leq |\sigma| \text{ then:} \\
\quad \sigma, t \models_D F[x_1 \mapsto F_1, \ldots, x_m \mapsto F_m] \\
\text{where } (N(T_1 x_1, \ldots, T_m x_m) = F) \in D \\
\text{otherwise, if } t = 0 \text{ or } t = |\sigma| + 1 \text{ then:} \\
\quad \text{rule } N \text{ is defined as max in } D
\end{array} \right.
\]

\[
\sigma, t \models_D N(F_1, \ldots, F_m) \quad \text{iff} \quad \left\{ \begin{array}{l}
\text{if } 1 \leq t \leq |\sigma| \text{ then:} \\
\quad \sigma, t \models_D F[x_1 \mapsto F_1, \ldots, x_m \mapsto F_m] \\
\text{where } (N(T_1 x_1, \ldots, T_m x_m) = F) \in D \\
\text{otherwise, if } t = 0 \text{ or } t = |\sigma| + 1 \text{ then:} \\
\quad \text{rule } N \text{ is defined as max in } D
\end{array} \right.
\]
 maximal S1() = init -> @ S1() \\
    /\ open -> @ S2() \\
    /\ ~(access /\ close) .

 minimal S2() = access -> @ S2() \\
    /\ close -> @ S1() \\
    /\ ~(init /\ open) .

 mon M = S1() .
combining automatae and temporal logic

\[
\begin{align*}
\max S1() &= \text{init} \rightarrow @ S1() \\
&\quad \land \text{open} \land \text{Prev(init)} \rightarrow @ S2() \\
&\quad \land \neg(\text{access} \lor \text{close}) .
\end{align*}
\]

\[
\begin{align*}
\min S2() &= \text{access} \rightarrow @ S2() \\
&\quad \land \text{close} \rightarrow @ S1() \\
&\quad \land \neg(\text{init} \lor \text{open}) .
\end{align*}
\]

\[
\begin{align*}
\mon M &= S1() .
\end{align*}
\]
Property:
File accesses are always enclosed by open and close operations.

\[ M = (\text{idle}^*;\text{open};\text{access}^*;\text{close})^* \]

\[
\begin{align*}
\text{max } S(\text{Term } t) &= t \setminus \mathbin{\@} S(t) . \quad \text{// Star} \\
\text{min } P(\text{Term } t) &= t \setminus \mathbin{\@} P(t) . \quad \text{// Plus}
\end{align*}
\]

\[
\text{mon } M = \\
S(S(\text{idle}());\text{open}();S(\text{access}());\text{close}()) .
\]
Property: Locks are acquired and released nested.

// Match rule:

max Match (Term l, Term r) =
    Empty() \ (l;Match(l,r);r;Match(l,r))

// Monitor:

mon M = Match(lock(),release())
RuleR

• attempt to develop a core logic for monitoring, a monitoring “byte-code”.
• intended to be simple to implement, and rather low level.
• should support easy mapping from high level logics to RuleR.
• ended up becoming a specification language on its own.
RuleR’s core and extensions

Extensions conceptually map into core (although not in implementation)
the key concept: a rule

If the \textit{lhs} is true in the current State

then the \textit{rhs} is added to the new state (all old information is lost)

\[ \text{R : lhs} \rightarrow \text{rhs} \]

Spec = Rule-set
Rule = State $\rightarrow$ State
State = Fact-set

Example if \textit{a} is true now \textit{b} must be true next:

\[ \text{R : a} \rightarrow \text{b} \]
RuleR by Example

Examples taken from:

RuleR: A Tutorial Guide
DRAFT: Version 0.1

Howard Barringer
propositional RuleR
Always a

ruler Always {
  ruleIDs { Ra , Rna }
  observes { a }
  rules {
    Ra : a \rightarrow Ra , Rna ;
    Rna : ! a \rightarrow Fail ;
  }
  initials { Ra , Rna }
}

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

specification monitors ‘a’ events as well as other events.

rule Ra makes sure rule Rna is continuously activated.

rule Rna (Rule ‘not a’) emits failure when ‘a’ is not true.
Eventually a

```
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 .
}
```

rule Rnb makes sure rule Rb is continuously activated.

rule Rb emits built-in Ok signal when ‘b’ is true.

forbidden rules are not allowed to exists in state at end of monitoring.
ruler RegularPattern {
  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: RegularPattern }
  run RE .
}

□(g → (ab)*c)
calling RuleR from AspectJ
the RuleR interface

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

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

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

    ...
}
```

the RuleR constructor takes fileName prefix and looks for specification in fileName.ruler. Generates output in fileName.output.

two dispatch methods exist, with and without additional event arguments.
**Signal**

<table>
<thead>
<tr>
<th>Status</th>
<th>Signal Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status(0)</td>
<td>Signal.FALSE</td>
<td>Property violated, no more monitoring</td>
</tr>
<tr>
<td>status(1)</td>
<td>Signal.STILL_FALSE</td>
<td>Property still not satisfied, could be later</td>
</tr>
<tr>
<td>status(2)</td>
<td>Signal.STILL_TRUE</td>
<td>Property still not violated, could be late</td>
</tr>
<tr>
<td>status(3)</td>
<td>Signal.TRUE</td>
<td>Property satisfied, no more monitoring</td>
</tr>
</tbody>
</table>

emitted to output at end:
an example Java program

```java
public class ExampleThree {

    static void a() {...}
    static void b() {...}
    static void c() {...}
    static void g() {...}

    static void end() {...}

    public static void main(String[] args) {
        a(); b(); c();
        g(); a(); b(); a(); b(); c();
        a();
        g(); a(); c();
        b();
        end();
    }
}
```
the aspect calling RuleR

```java
package exp;

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

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

    pointcut scope():
        !cflow(adviceexecution()) && if(true);

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

    ...

    ...

    advices

}
... 

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

```java
before() : b() && scope() {...}
before() : c() && scope() {...}
before() : g() && scope() {...}
```

```java
before() : call(void end()) && scope() {
    ruler.dispatchEnd();
}
```

... 

---

**advises**

reaction must be defined in aspect code.

aspect code has to emit end event.
output

Rule system RE.RegularPattern
a called
b called
c called
g called
a called
b called
a called
b called
c called
a called
g called
g called
a called
Symbol c incorrect
extending core-RuleR with three kinds of rule declarations
ruler RegularPattern {
  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: RegularPattern }
  run RE .
}

\(\Box(g \rightarrow (ab)^*c)\)
ruler RegularPatternV2 {
  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 }
}

reformulation of spec □(g → (ab)*c)

three kinds of rules, each with different notion of persistence:

- **always rules**: always active unless explicitly de-activated
- **state rules**: remain active until fired, unless as above
- **step rules**: the basic core semantics, survive one step only, unless re-activated

monitor {
  uses {RE : RegularPatternV2}
  run RE .
}
rule parameters
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;
    }
  }
  initials { Terminal }
  forbidden { A, B, C }
}

monitor this:
(a^n b^{2n} c^{3n})^+

... state Terminal {
  a -> A(1);
  !a -> Fail;
} ...

initials { Terminal }
forbidden { A, B, C }

monitor {
  uses {ABC : CountingABC}
  run ABC .
}
parameterized events
ruler RegularPatternV3 {
    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 this:
∀ n • □(g(n) → □((ab)^n c))

now our aspect must provide ‘x’ argument to dispatch method whenever ‘g’ is dispatched.

monitor {
    uses {RE : RegularPatternV3}
    run RE .
}
public aspect REPATTERNV3 {
    RuleR ruler =
        new RuleR("src/examples/REPatternV3", false);

    pointcut scope() :
        !cflow(adviceexecution()) && if(true);

    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() && scope() {...}
    before() : b() && scope() {...}
    before() : c() && scope() {...}
    ! before(int x) : g(x) && scope() {
        ! if (ruler.dispatch("g", new Object[]{x}) == Signal.FALSE) {
            System.err.println("Symbol g incorrect");
            System.exit(0);
        }
    }
    before() : call(void end()) && scope() {...}
}
conditionals
ruler RegularPatternV3 {
  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 : RegularPatternV3}
  run RE .
}

back to the spec:

\( \forall n \cdot \Box (g(n) \rightarrow \Diamond ((ab)^n c)) \)

common terms

arithmetic conditions are complementary \((x>0, x<=0)\)

conditionals
ruler RegularPatternV4 {
  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;
  }
}

using conditionals

∀ n • □(g(n) → 〇((ab)ⁿc))

conditional

conditional

initials { G }
forbidden { S, T }

monitor {
  uses {RE: RegularPatternV4}
  run RE .
}
parameterized rule schemas
ruler Always\((\(y\))\) {  
  ruleIDs \{ R \}  
  rules \{  
    always R\((x)\) {  
      x -> Ok;  
      !x -> Fail;  
    }  
  }  
  initials \{\(y,R(\(y\))\)\}  
}

ruler Eventually\((\(y\))\) {  
  ruleIDs \{ R \}  
  rules \{  
    step R\((x)\) {  
      x -> Ok;  
      !x -> R\((x)\);  
    }  
  }  
  initials \{R(\(y\))\}  
  forbidden \{R\}  
}

monitor this:
\(\Box(g \to \Box a) \land \Box(h \to \Diamond b)\)

ruler Comb {  
  uses \{A:Always,E:Eventually\}  
  ruleIDs \{ R \}  
  observes \{g,h,a,b\}  
  rules \{  
    always R \{  
      g -> A(a);  
      h -> E(b);  
    }  
  }  
  initials \{ R \}  
}

monitor \{  
  uses \{C:Comb\}  
  run C .  
\}
super rules
inspired by UML state charts

variables
char *file;

close opened files eventually
and don’t write after a read.

opened

unread
READ(f) [f==file]

modified
WRITE(f) [f==file]

write
WRITE(f) [f==file]

READ(f) [f==file]

read

opened

CLOSE(f) [f == file]/f := ""

modified

unread

OPEN(f)/file := f

/error

entry/ Error()

END

open

closed

/file := ""

write
WRITE(f) [f==file]

READ(f) [f==file]

read

WRITE/ Error()
ruler FileMonitor {
  ruleIDs {Start, Opened, Unread, Read, Modified}
  observes {OPEN, END, CLOSE, READ, WRITE}
  rules {
    always Start {
      OPEN(f), !Opened(f) \rightarrow Unread(f);
    }
    state Opened(f) super {Unread, Read, Modified} {
      END \rightarrow Fail;
      CLOSE(f) \rightarrow Ok;
    }
    state Unread(f) extends Opened {
      READ(f) \rightarrow Read(f);
      WRITE(f) \rightarrow Modified(f);
    }
    state Read(f) extends Opened {
      WRITE(f) \rightarrow Fail;
    }
    state Modified(f) extends Opened {
      READ(f) \rightarrow Read(f);
    }
  }
  initials {Start}
}
real time and rule piping
requirements

- the events: openDoor, passDoor, closeDoor.
- 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 resets the timer.
- only a limited number of doors may be open at any one time.
ruler AlarmedDoorMonitor(maxopen, alarm) {
   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);
            -> alarm;
         }
      }
      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;
            :}
         :}
      }
   }
}
initials { Start(0, maxopen) }
outputs { alarm }

ruler AlarmHandler(alarm) {
  ruleIDs { Start }
  rules {
    state Start {
      alarm -> Fail;
    }
  }
  initials { Start }
}

rule schemas are being piped together communicating via the bell

monitor {
  uses { D: AlarmedDoorMonitor,
          H: AlarmHandler }
  locals { bell }
  run (D(3,bell) >> H(bell)) .
}
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();
}
public aspect AlarmedMonitor {
    RuleR ruler = new RuleR("...", 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) && scope()
        {
            Signal s = ruler.dispatch("openDoor", new Object[]{d, limit});
            if (s==Signal.FALSE){...
        }

    before(CheckDoor.Door d): passDoor(d) && scope() {...
    before(CheckDoor.Door d): closeDoor(d) && scope() {...
    after(): call(void end()) && scope() {...
}

the aspect
asserting progress
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 .
}

monitor this:
lock^n unlock^n

assert {R_1,...,R_n}
fails if not at least one of the rules R_1,...,R_n fire.
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 this:

lock^n unlock^n

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
</tr>
</thead>
</table>
lock | L(E) |
lock | L(U(E)) |
lock | L(U(U(E))) |
unlock | U(U(E)) |
unlock | U(E) |
unlock | E |

monitor {
  uses {S: SimpleCFLV2}
  run S .
}
advanced rule piping
using parameterized events
Repeated searches in binary tree

- good search: max search depth < G% of tree size
- bad search: max search depth > B% of tree size
find(x)
  = 1 if x is in tree
  = 0 if x is not in tree

find(6):
  depth=3
  < 33% of size
  = good search

size of tree = 10

percent of
Size of tree

0  G=33  B=50  100
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(tree, max, size), Top;
          -> Top;
        }
        -> Stack(tree, level-1, max, size);
      }
    }{Top}
  }
  initials { Top }
  outputs { result }
}
ruler Stats(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(tree, Bs, Gs);
  }
  initials { Start }
}
outputs { report }

monitor {
  uses {T: Trace,S: Stats}
  run (T >> S(33, 50)) .
}
properties of Java library APIs

R$_1$: There should be no two calls to `next()` without a call to `hasNext()` in between, on the same iterator.
ruler IteratorMonitor{
  ruleIDs { Start, Next }
  observes { hasNext, next }
  rules {
    always Start {
      hasNext(i), !Next(i) \rightarrow Next(i);
    }
    state Next(i) {
      next(i) \rightarrow Ok;
    }
    assert {Start, Next }
  }
  initials { Start }
}

monitor {
  uses { IM: IteratorMonitor }
  run IM .
}
public aspect iteratormonitor {
    RuleR ruler = new RuleR("src/examples/hasNext", false);

    void dispatch(String event, Object[] args) {...}

    pointcut scope() :
        !cflow(adviceexecution()) && if(true);

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

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

    before(Iterator i): hasNext(i) && scope() {
        dispatch("hasNext", new Object[] {i});
    }

    before(Iterator i): next(i) && scope() {
        dispatch("next", new Object[] {i});
    }
}
An enumeration should not be propagated after the underlying vector has been changed.
ruler SafeEnum {
    ruleIDs {Start, Next, Update}
    observes {create_enum, call_next, update_source}

    rules {
        always Start {
            create_enum(v,e) -> Next(v,e);
        }

        state Next(v,e) {
            update_source(v) -> Update(e);
        }

        state Update(e) {
            call_next(e) -> Fail;
        }
    }

    initials { Start }
}

monitor {
    uses {S : SafeEnum}
    run S .
}
current limitations of RuleR

• still not as succinct as for example regular expressions in some cases (RE: ‘next next’).
• optimization not begun.
• it is a prototype under development, including language design.
end