contains material
from RuleR tutorial
written by
Howard Barringer

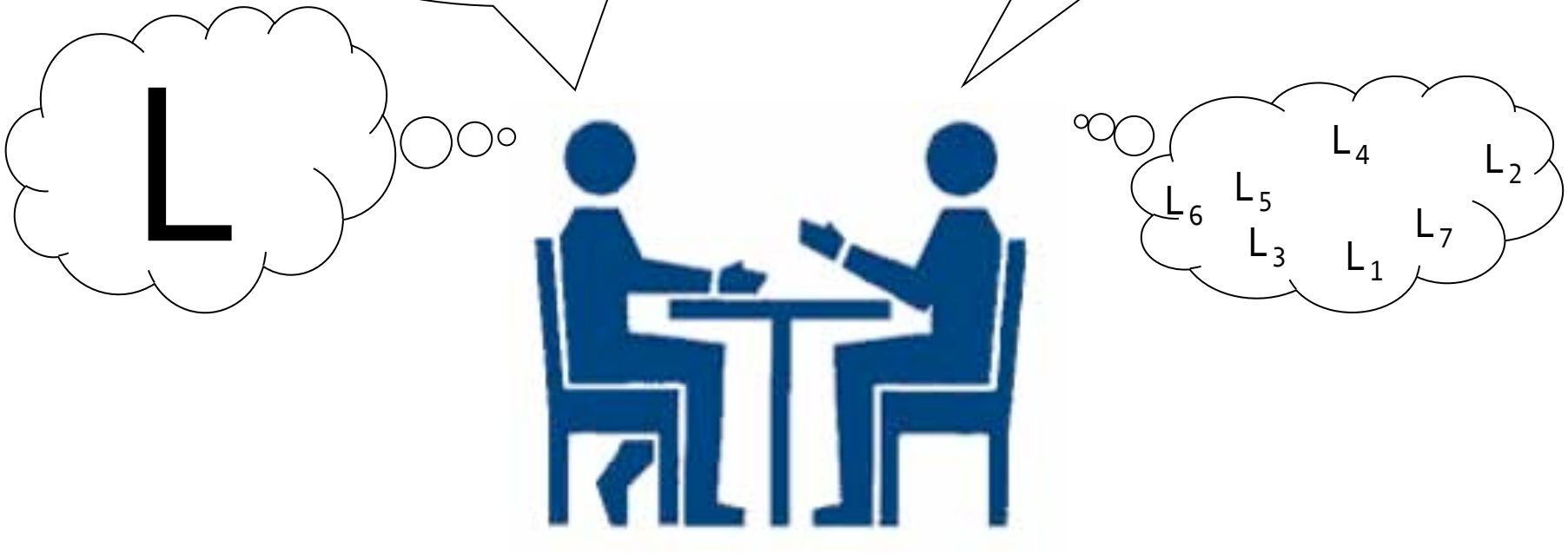
Rule-Based Systems

CS 119

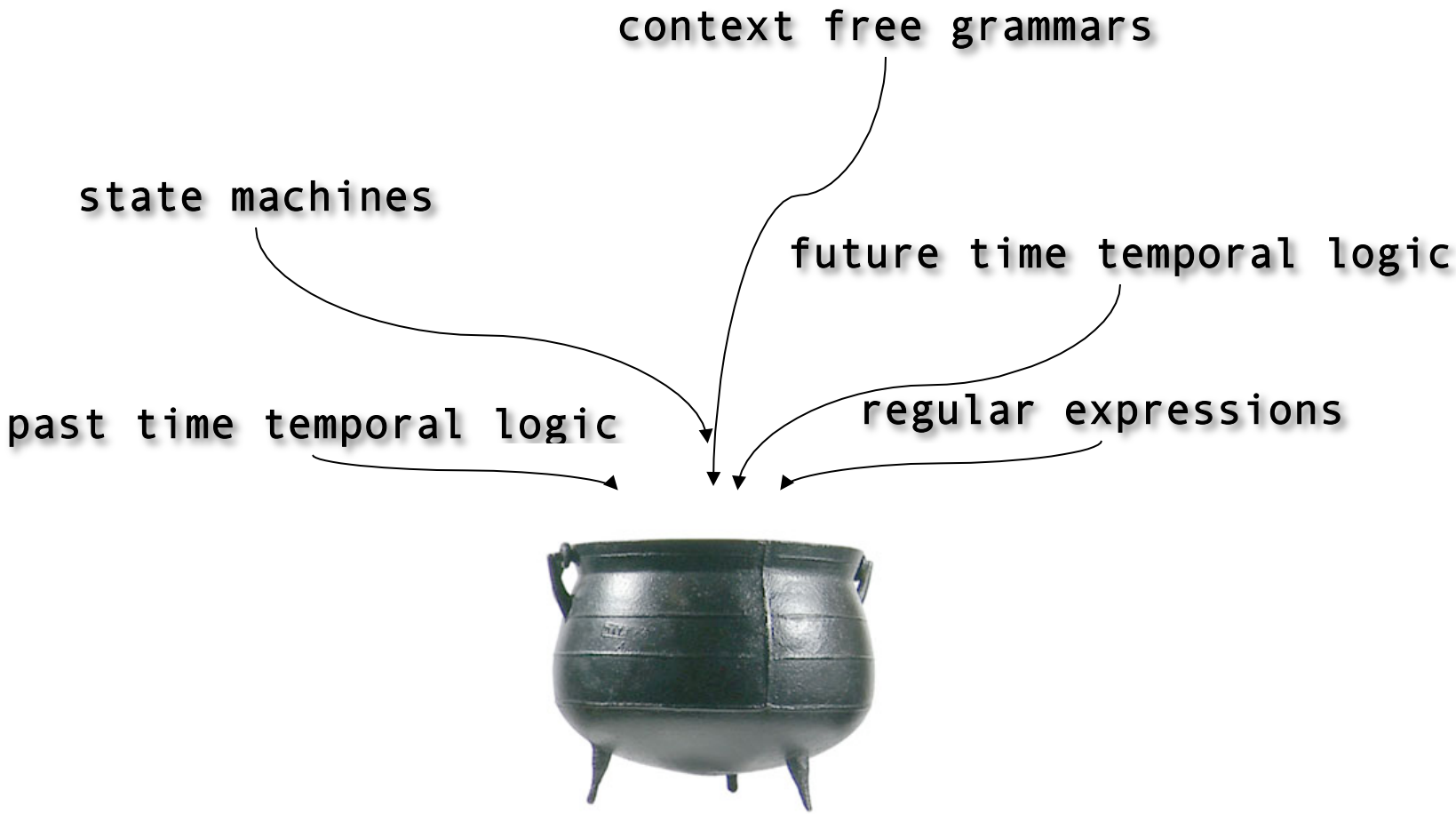
the search for a unifying
monitoring logic

there is a unifying and practical logic

there is no such!



a unifying logic



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

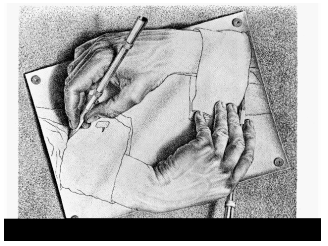
Eagle

Howard Barringer
Allen Goldberg
Klaus Havelund
Koushi Sen

p Λ @ q

temp. logic

functions



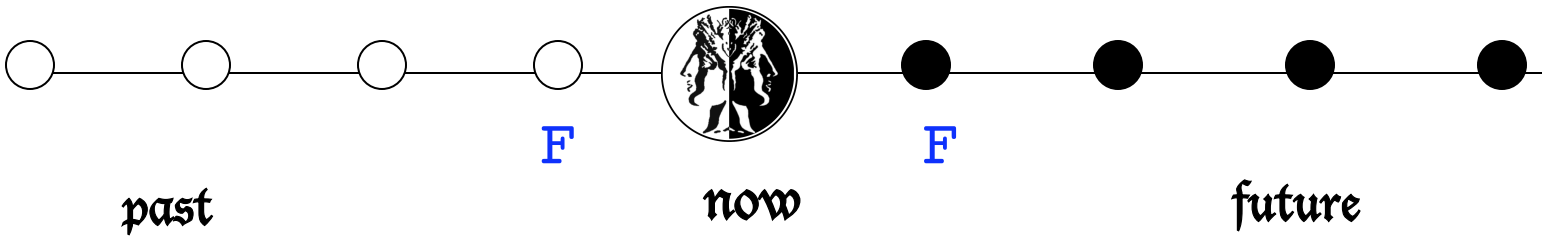
recursion

temporality + recursion

- propositional logic + two main temporal connectives:
  - **Next:** `@F`
  - **Previous:** `#F`

- Recursion:

```
safe always(Term p) = p /\ @always(p)
live eventually(Term p) = p \/ @eventually(p)
live previously(Term p) = p \/ #previously(p)
```



example:

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

syntax:

$$
\begin{aligned}
S &::= \textbf{dec } D \textbf{ obs } O \\
D &::= R^* \\
O &::= M^* \\
R &::= \{\textbf{max} \mid \textbf{min}\}\, N(T_1\, x_1, \ldots, T_n\, x_n) = F \\
M &::= N = F \\
T &::= \textbf{Form} \mid \textit{java primitive type} \\
F &::= \textit{java expression} \mid \textbf{True} \mid \textbf{False} \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid \\
&\quad \bigcirc F \mid \odot F \mid F_1 \cdot F_2 \mid N(F_1, \ldots, F_n)
\end{aligned}
$$

semantics

$\sigma, i \models_D exp$ iff $1 \leq i \leq |\sigma|$ and $evaluate(exp)(\sigma(i)) == true$

$\sigma, i \models_D \underline{\text{true}}$

$\sigma, i \not\models_D \underline{\text{false}}$

$\sigma, i \models_D \neg F$ iff $\sigma, i \not\models_D F$

$\sigma, i \models_D F_1 \wedge F_2$ iff $\sigma, i \models_D F_1$ and $\sigma, i \models_D F_2$

$\sigma, i \models_D F_1 \vee F_2$ iff $\sigma, i \models_D F_1$ or $\sigma, i \models_D F_2$

$\sigma, i \models_D F_1 \rightarrow F_2$ iff $\sigma, i \models_D F_1$ implies $\sigma, i \models_D F_2$

$\sigma, i \models_D \bigcirc F$ iff $i \leq |\sigma|$ and $\sigma, i+1 \models_D F$

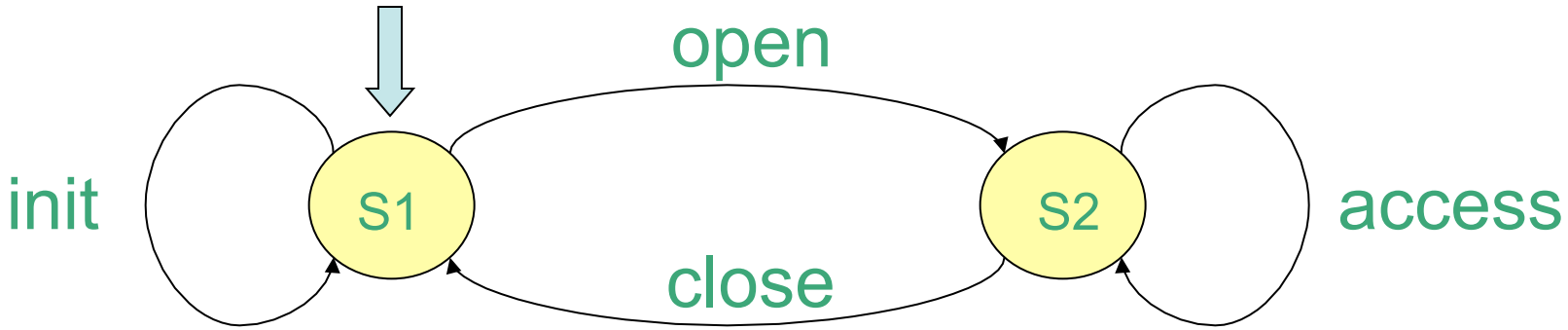
$\sigma, i \models_D \odot F$ iff $1 \leq i$ and $\sigma, i-1 \models_D F$

$\sigma, i \models_D F_1 \cdot F_2$ iff $\exists j$ s.t. $i \leq j \leq |\sigma| + 1$ and $\sigma^{[1,j-1]}, i \models_D F_1$ and $\sigma^{[j,|\sigma|]}, 1 \models_D F_2$

$\sigma, i \models_D N(F_1, \ldots, F_m)$ iff

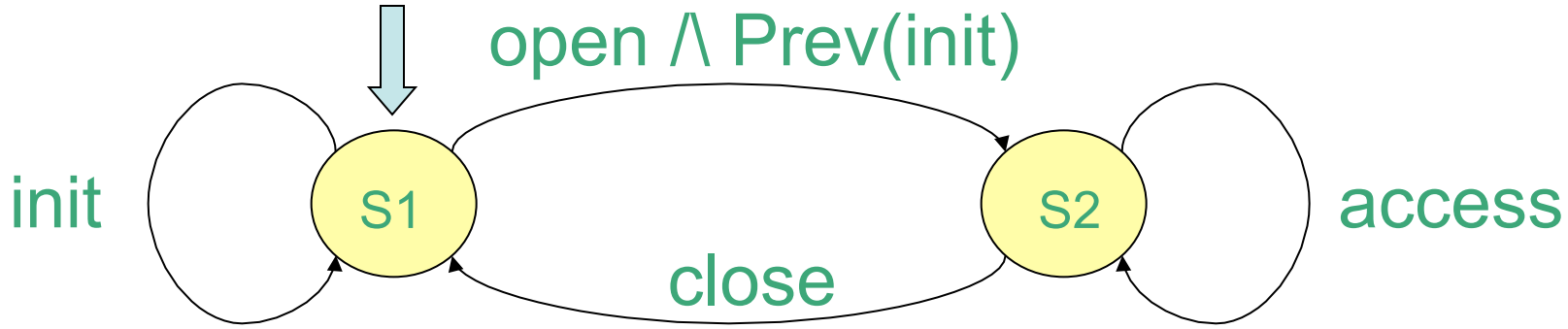
- if $1 \leq i \leq |\sigma|$ then:
  - $\sigma, i \models_D F[x_1 \mapsto F_1, \ldots, x_m \mapsto F_m]$
  - where $(N(T_1\ x_1, \ldots, T_m\ x_m) = F) \in D$
- otherwise, if $i = 0$ or $i = |\sigma| + 1$ then:
  - rule N is defined as $\underline{\text{max}}$ in D

automatae




```
max S1() = init -> @ S1()
         /\ open -> @ S2()
         /\ ~(access \/ close) .
min S2() = access  -> @ S2()
         /\ close -> @ S1()
         /\ ~(init \/ open) .
mon M = S1() .
```

combining automatae and temporal logic




```
max S1() = init -> @ S1()
         /\ open /\ Prev(init) -> @ S2())
         /\ ~(access \/ close) .
min S2() = access  -> @ S2()
         /\ close -> @ S1()
         /\ ~(init \/ open) .
mon M = S1() .
```

regular expressions

Property:
File accesses are always enclosed by open and close operations.

M = (idle*;open;access*;close)*

```
max S(Term t) = t /\ @ S(t) . // Star
min P(Term t) = t /\ @ P(t) . // Plus


mon M =
  S(S(idle());open();S(access());close()) .
```

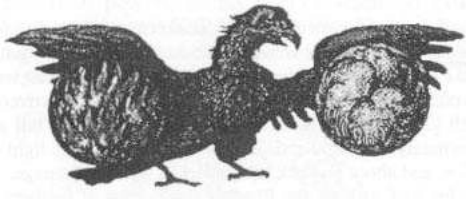
Property:
Locks are acquired and released nested.

grammars

lock
release

lock
  lock
  release
release

```
// Match rule:

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


// Monitor:

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

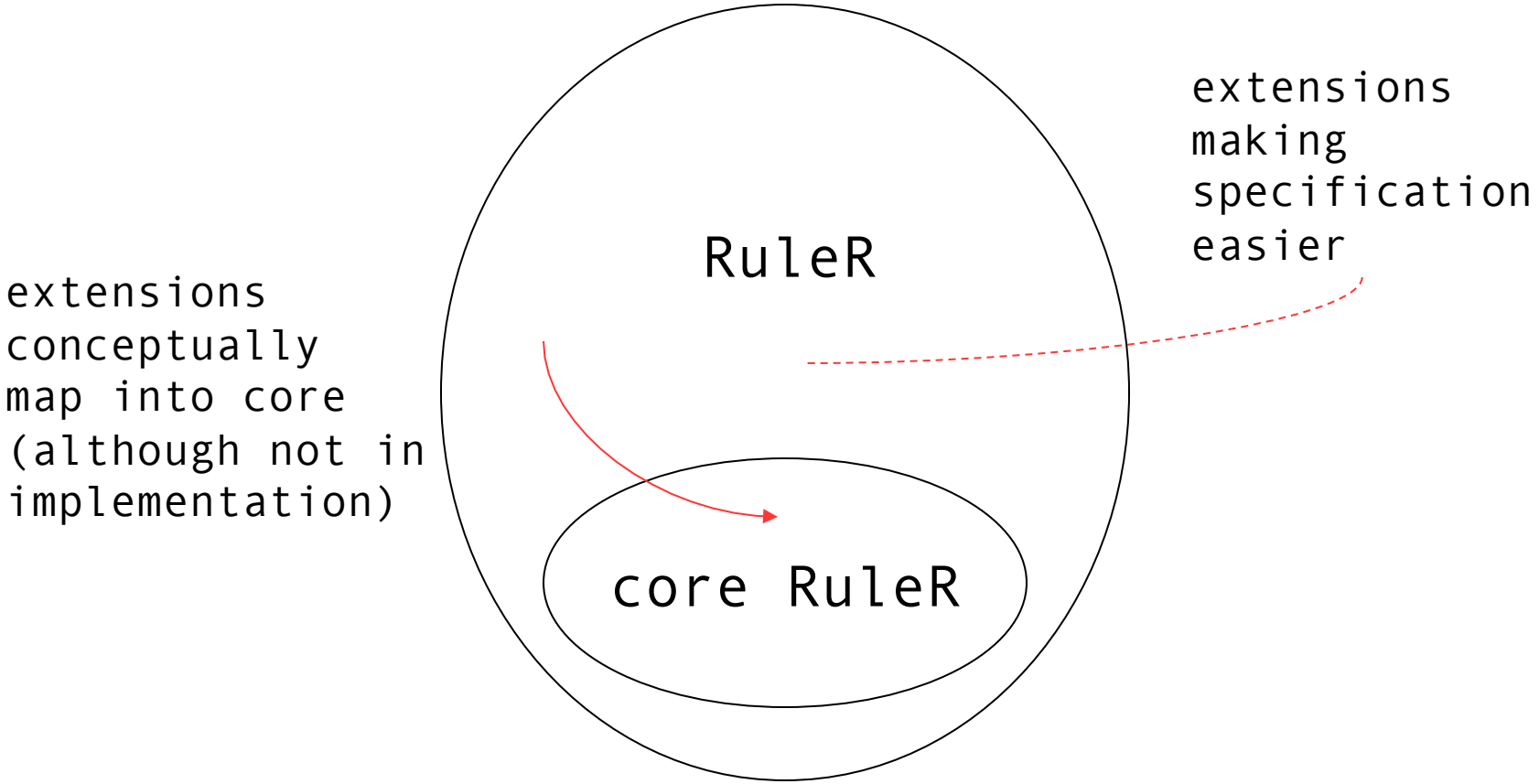


RuleR

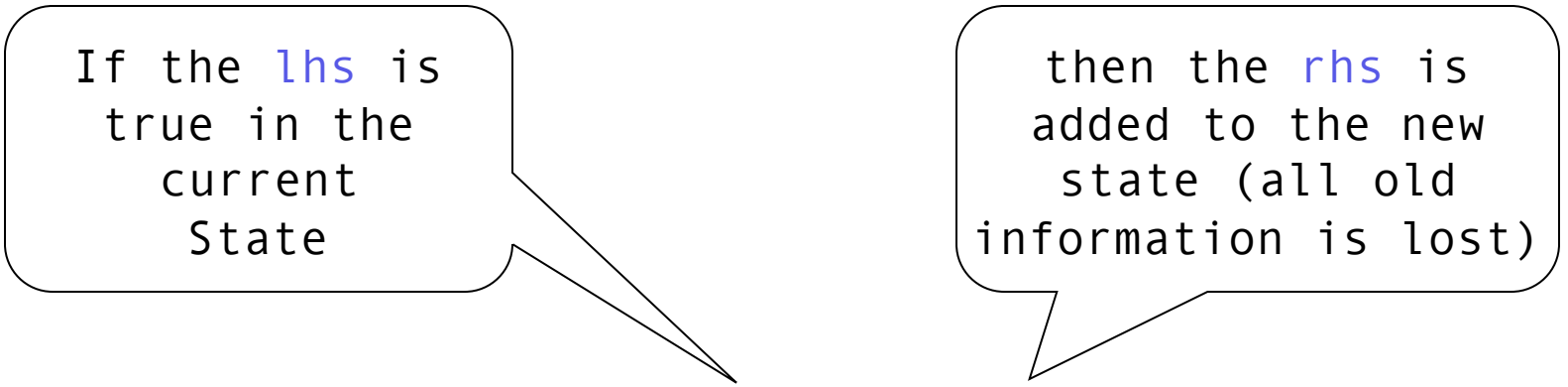
Howard Barringer
David Rydeheard
Klaus Havelund

- 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



the key concept : a rule



Spec = Rule-**set**
Rule = State → State
State = Fact-**set**

Example if a is true now b must be true next:

R : a → 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 -> Ra , Rna ;
    Rna : ! a -> Fail ;
  }
  initials { Ra , Rna }
}

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

□ 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.

$\Diamond a$

```
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 .
}
```

a g should be followed by a sequence of ab's and then a c.

rules Rg and Rng are always active.

in case g occurs, rules Sa, Sc and Snac (not a or c) are activated.

rule Sa and Sb represent the (ab)*-iteration. Sc terminates the (ab)*c loop.

$\Box(g \rightarrow (ab)^*c)$

calling RuleR from AspectJ

the RuleR interface

```
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 consructor takes fileName prefix and looks for specification in fileName.ruler. Generates output in fileName.output.

two dispatch methods exists, with and without additional event arguments.

Signal

```
emitted
to output
at end:
```

`status(0)`
- Signal.FALSE:
  - Property violated, no more monitoring

`status(1)`
- Signal.STILL_FALSE:
  - Property still not satisfied, could be later

`status(2)`
- Signal.STILL_TRUE:
  - Property still not violated, could be late

`status(3)`
- Signal.TRUE:
  - Property satisfied, no more monitoring

an example Java program

```
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();   <-- error
    b();
    end();
  }
}
```

the aspect calling RuleR

```
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(tru

  pointcut a() : cal(void a());

 pointcut b() : ca                  ;

 pointcut c() : call(void c());
```

no timestamps

specification is in the file: REPattern.ruler

output is written to the file: REPattern.output

advices

```
…

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

 }
}

before() : b() && scope() {…}
before() : c() && scope() {…}
before() : g() && scope() {…}



before() : call(void end()) &&
  ruler.dispatchEnd();

}

…
```

advices

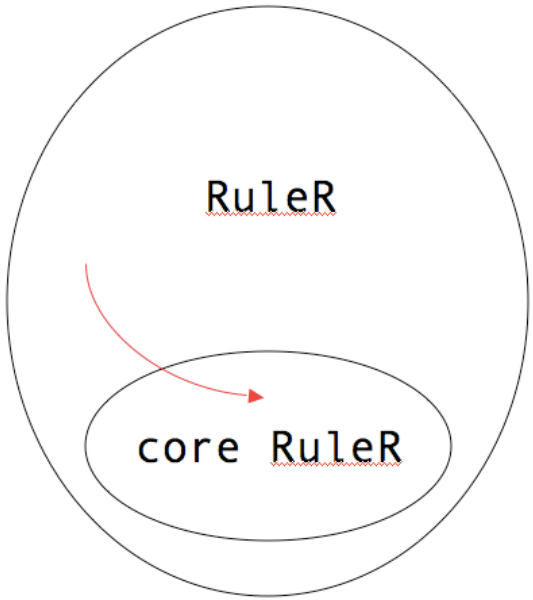
reaction must be defined in aspect code.

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
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 .
}
```

recall this spec

rules in core-RuleR are "single-shot":

for a rule to persist, some (perhaps other) rule must generate it in each step.

some rules belong together:

- Sa, Sc and Snac
- Sb, Snb

$\Box(g \rightarrow (ab)^*c)$

```
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

$\Box(g \rightarrow (ab)^*c)$

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;
    }
    …
```

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:

$$\forall\ n \bullet \Box(g(n) \rightarrow \bigcirc((ab)^n c))$$

now our aspect must provide '*x*' argument to dispatch method whenever 'g' is dispatched.

```
monitor {
  uses {RE : RegularPatternV3}
  run RE .
}
```

the aspect

```
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 }
}
```

back to the spec:

$$\forall\, n \bullet \Box(g(n) \rightarrow \bigcirc((ab)^n c))$$

common terms

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

⇓

conditionals

```
monitor {
  uses {RE : RegularPatternV3}
  run RE .
}
```

using conditionals

$\forall\ n \bullet \Box(g(n) \rightarrow \bigcirc((ab)^n c))$

```
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;
    }
  }
```

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:

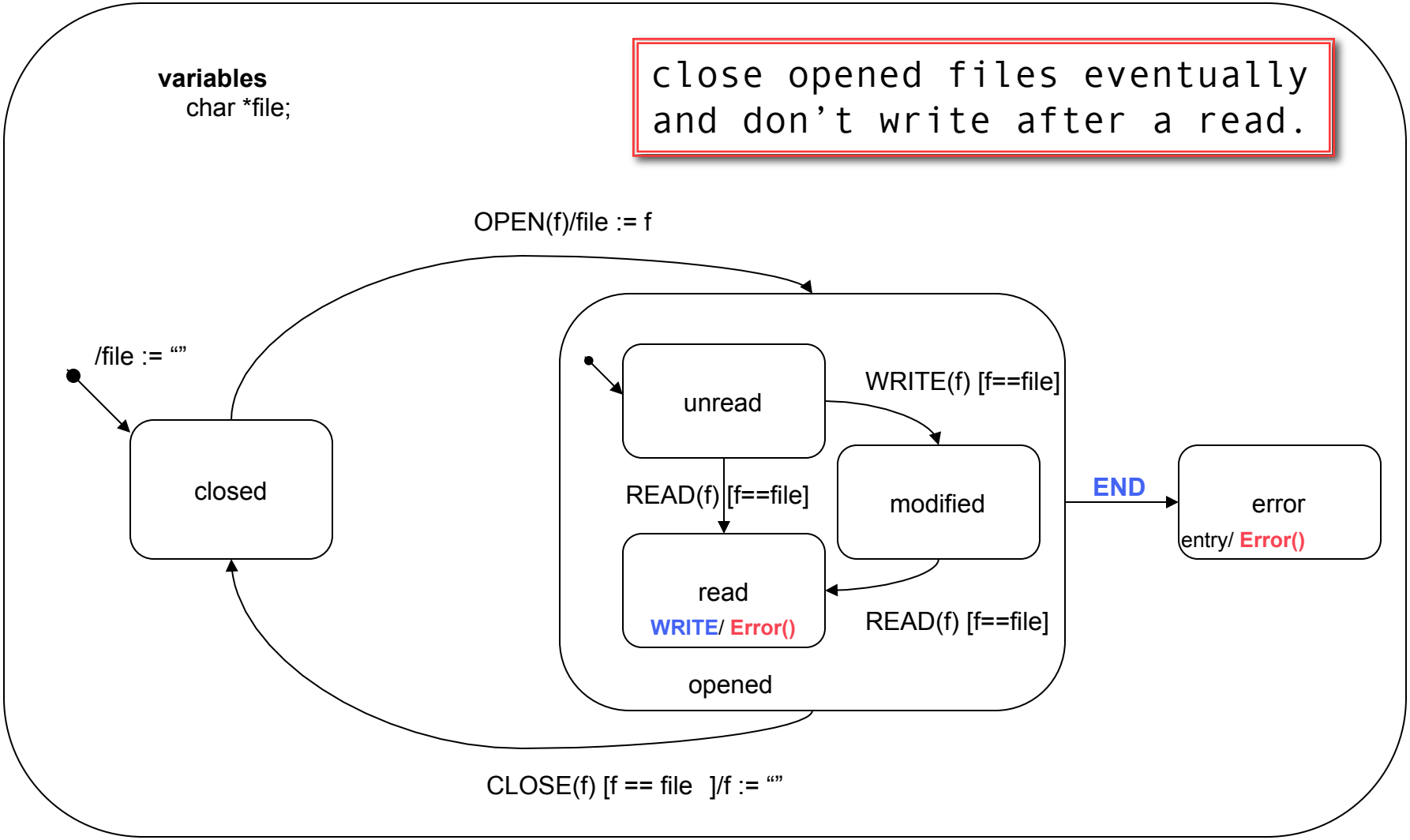
$$\Box(g \rightarrow \Box\ a) \wedge \Box(h \rightarrow \Diamond\ b)$$

```
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

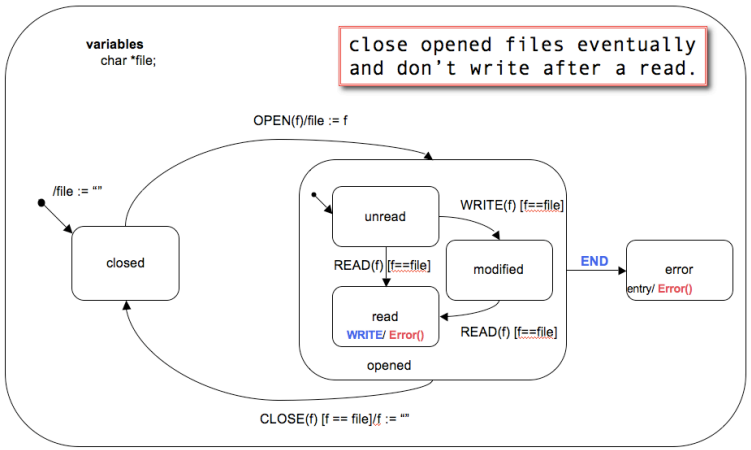
inspired by UML state charts



```
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 Modified(f) extends Opened {
      READ(f) -> Read(f);
    }
  }
  initials {Start}
}
```

super rules

activating a sub-rule also activates super rule.



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();
}
```

an application

turn time-stamping on

```
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 this:

$lock^n\ unlock^n$

assert $\{R_1,\ldots,R_n\}$

fails if not at least one of the rules $R_1, \ldots, R_n$ fire.

```
monitor {
  uses {S: SimpleCFLV2}
  run S .
}
```

```
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:

$lock^n\ unlock^n$

	S | E
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

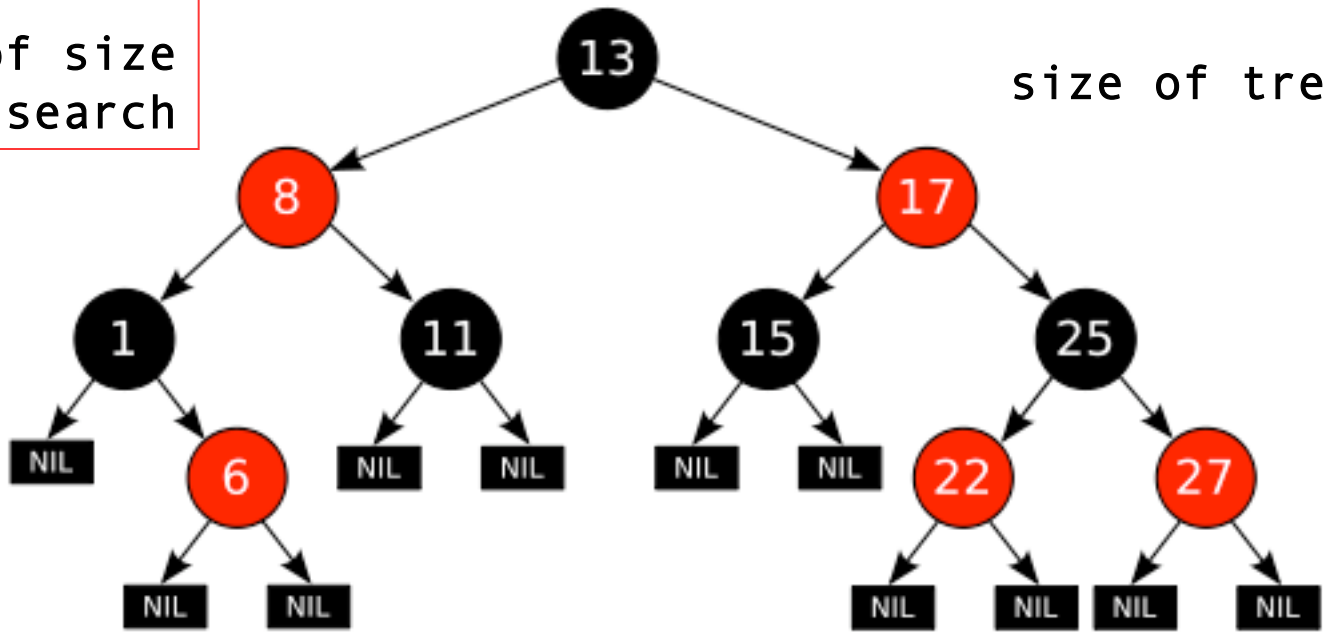
Repeated searches in binary tree

report when there are 1.5 times more bad searches than good searches:

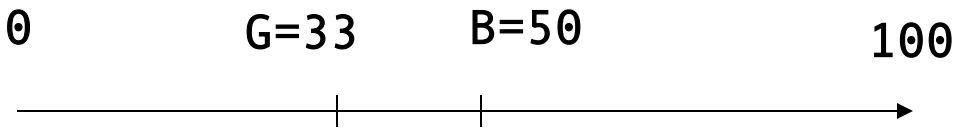
- good seach: max search depth < G% of tree size
- bad search: max seach depth > B% of tree size

```
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

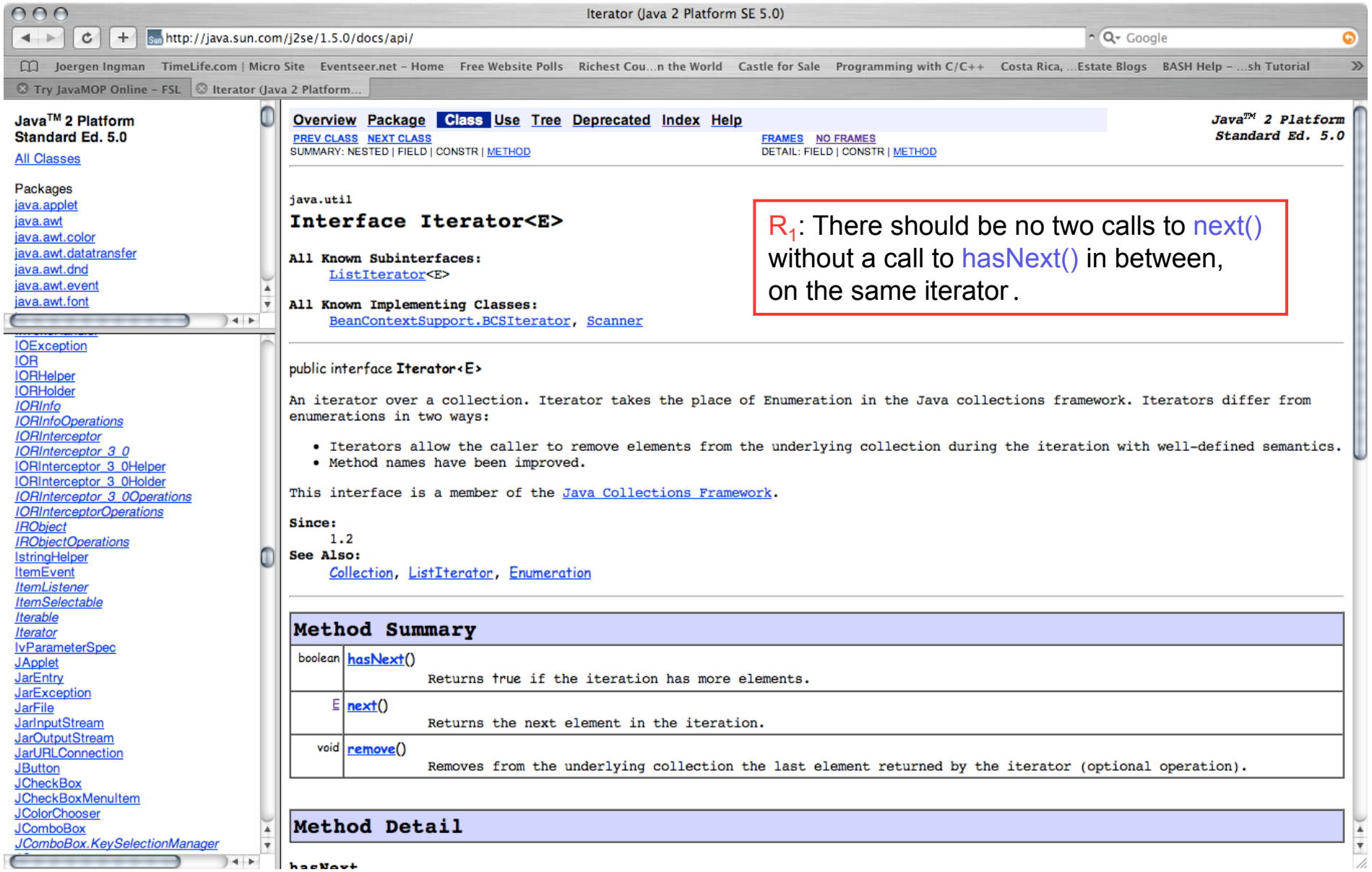
```
ruler Trace {
  ruleIDs { Top, Stack }
  observes { call, return }
  locals { result }
  rules {
    state Top{
      call(tree,size) -> Stack(tree, 1, 1, size);
    }
    state Stack(tree, level, max, size){
      call(x, y) -> Stack(tree, level+1, max+1, size);
      return(x, b)
        {: level=1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

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

```
initials { Start }
outputs { report }
```

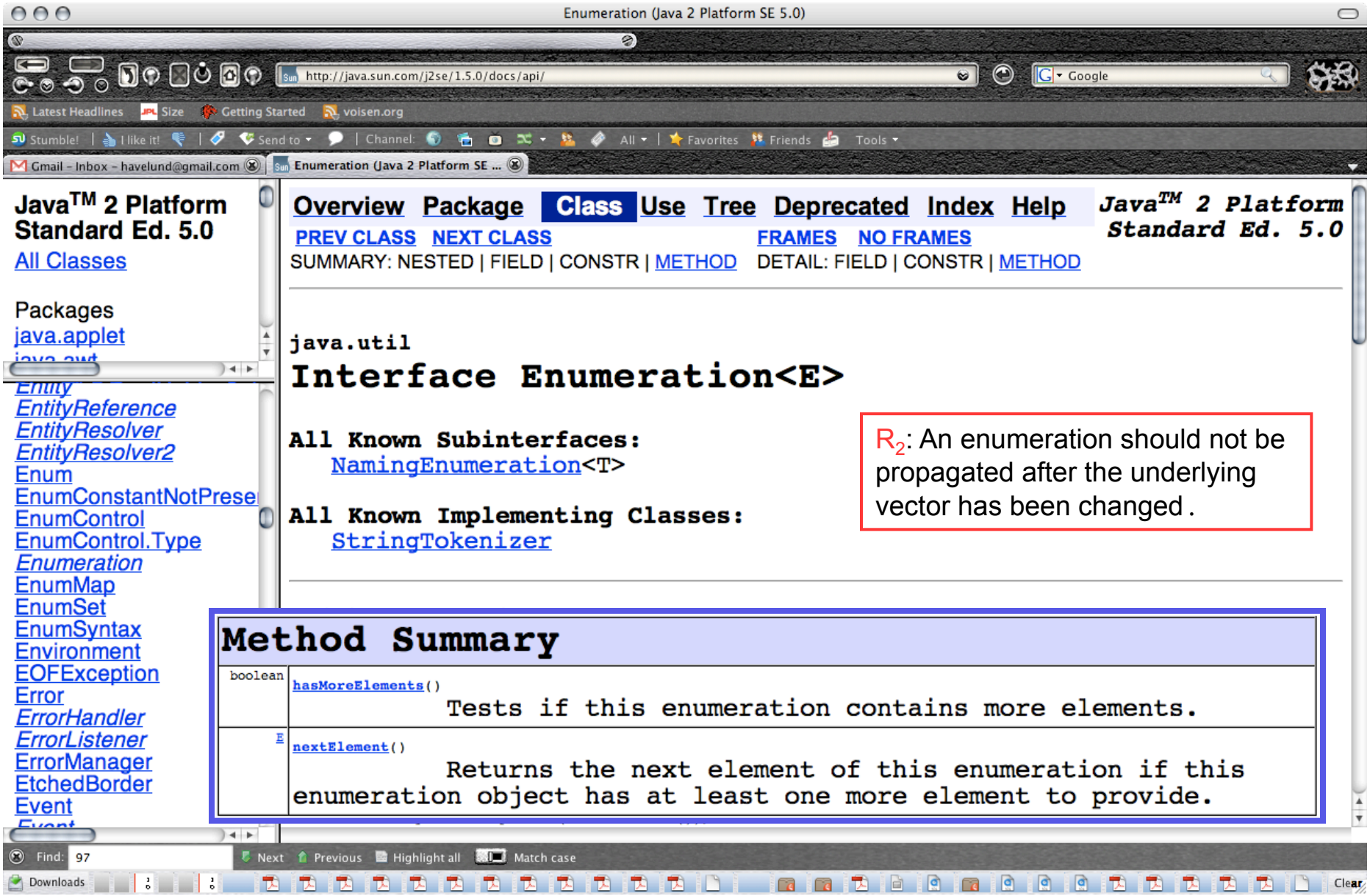
properties of Java library APIs



```
ruler IteratorMonitor{
  ruleIDs { Start, Next }
  observes { hasNext, next }
  rules {
    always Start {
      hasNext(i), !Next(i) -> Next(i);
    }
    state Next(i) {
      next(i) -> 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});
  }
}
```

properties of Java library APIs



```
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