Jet Propulsion Laboratory
California Institute of Technology

ARTIST2 Summer School 2008 in Europe
Autrans (near Grenoble), France
September 8-12, 2008

Rule-Based Runtime Verification

Howard Barringer
David Rydeheard
U. Of Manchester

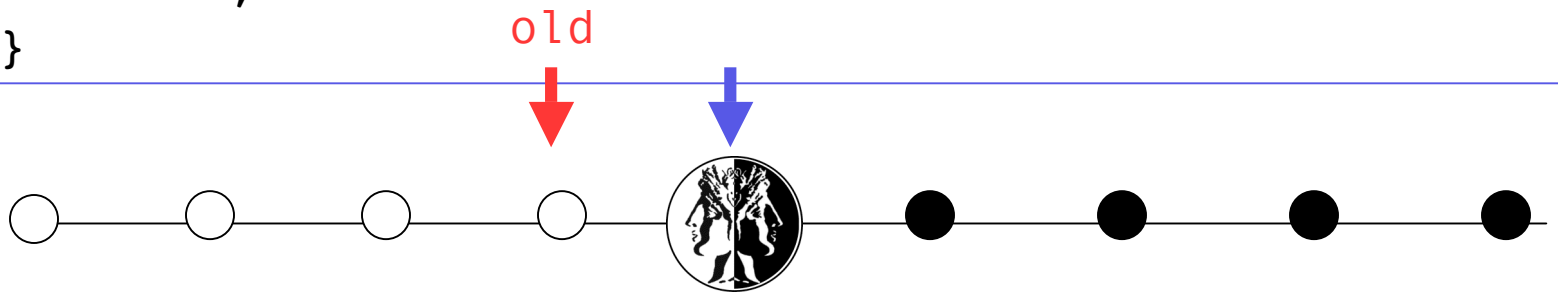
Lecturer: Klaus Havelund
Principal Scientist
Jet Propulsion Laboratory
California Institute of Technology

outline

- what is runtime verification?
- goal of our work
- fundamentals of rule-based monitoring
- extensions
- specification of Java APIs
- conclusion

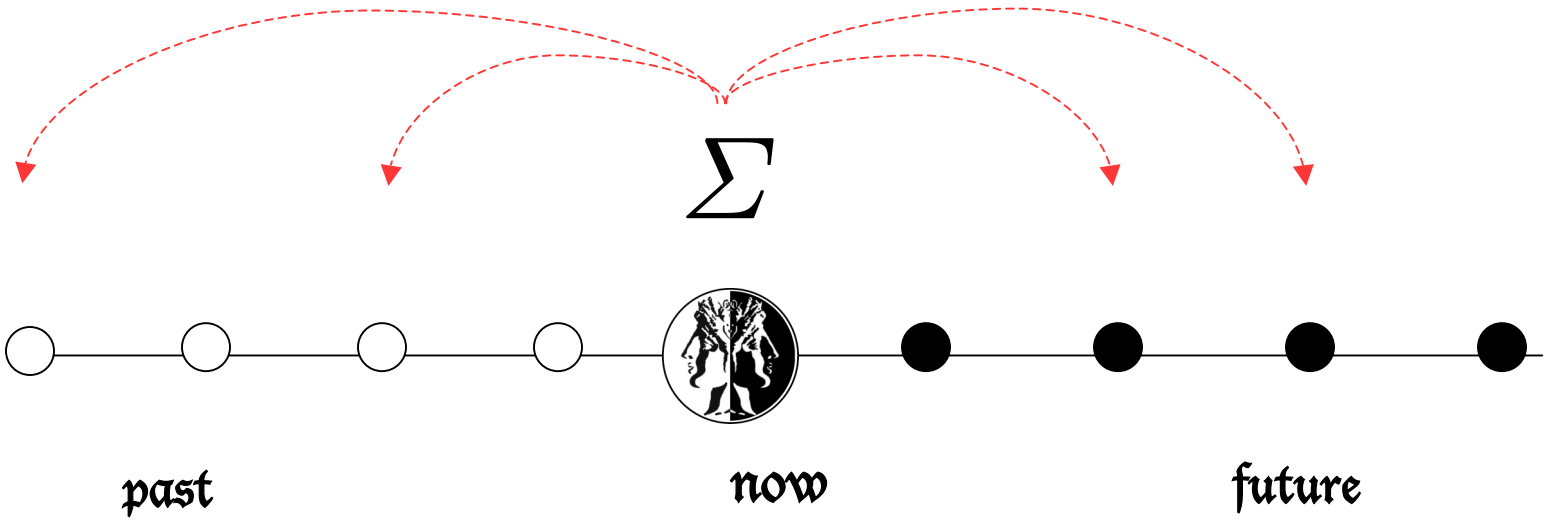
design by contract (JML)

```
//@ public invariant 0 <= size;
/*@ requires size < elems.length-1;
  @ assignable elems[size], size;
  @ ensures size == \old(size+1);
  @ ensures elems[size-1] == x;
  @ ensures_redundantly
  @   (\forall int i; 0 <= i && i < size-1;
  @      elems[i] == \old(elems[i]));
  @*/

public void push(Object x) {
  elems[size] = x;
  size++;
}
```



monitor should keep rich state Σ

$$\Sigma = \texttt{past events} \times \texttt{future obligations}$$



definition

runtime verification is the study of how to design artifacts for monitoring and analyzing system executions. Such artifacts can be used for a variety of purposes, including testing, program understanding and fault protection.

field still has many names

- runtime verification
- runtime monitoring
- runtime checking
- runtime result checking
- runtime reflection
- monitoring oriented programming
- runtime analysis
- dynamic analysis
- trace analysis
- fault protection
- supervised execution
- …

a broader view

- checking execution against a spec
  - state machines
  - temporal logic
  - regular expressions
  - grammars
  - …
- checking execution with algorithms
  - data race potentials
  - deadlock potentials
- learning specs from executions
  - invariants
  - temporal formulas
- trace visualization

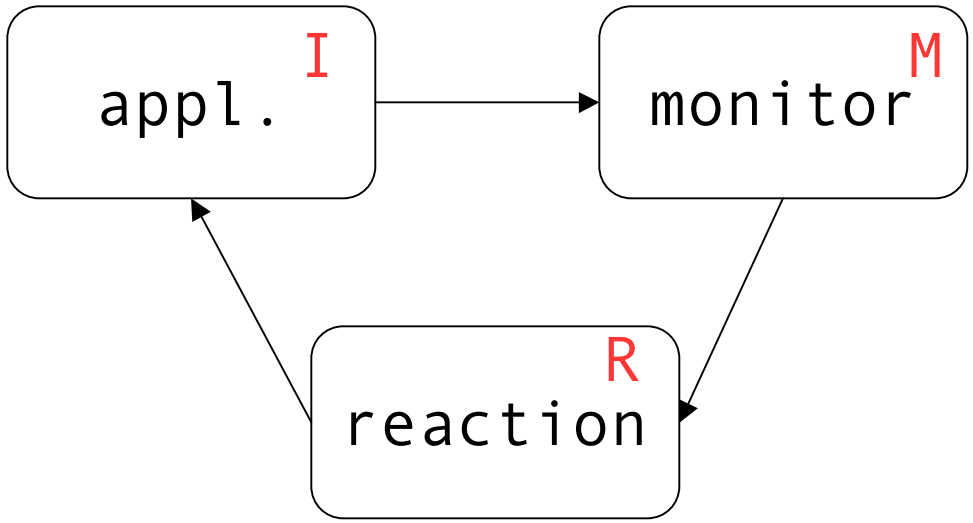
get as much out of runs as possible

components

Instrumentation

Monitoring

Reaction



instrumentation

- manual
  - assertions
  - pre/post conditions in design by contract solutions
- automated
  - instrumentation of source code:
    - CIL (C)
  - instrumentation of byte/object code:
    - BCEL (Java)
    - Valgrind (C)
  - high level bytecode APIs:
    - Sofya
  - aspect orinted programming:
    - AspectJ (Java)
    - AspectC (C)
    - AspectC++ (C++)

logfile analysis

```
begin of task 10!
execute subtask 1
execute subtask 2
end of task 10!
send data to ground
…
```

```
event begin(t) = begin of task (\d+)
event end(t)   = end of task (\d+)
event send()   = send data
```

```
begin("10")
end("10")
send()
…
```

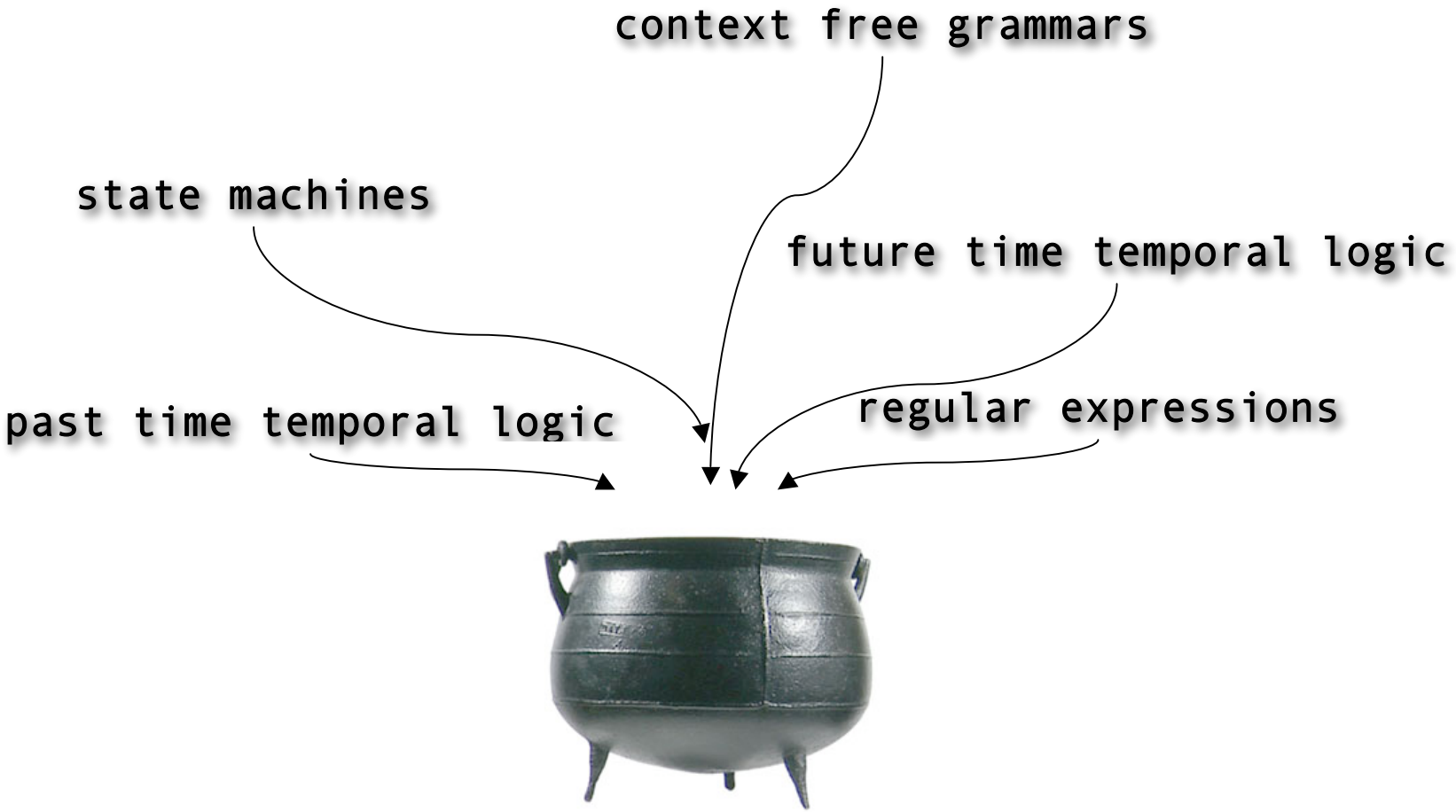
monitoring

- Java's assert statements
- JML, pre/post conditions
- Temporal Rover (commercial), metric future time temporal logic/UML
- JLO, future time temporal logic
- Java MaC, past time temporal logic
- Tracematches, regular expressions
- PQL, context free grammars
- PTQL, SQL
- T-UPPAAL, timed automata
- Jass, CSP (process algebra, pre/post conditions)
- Java MOP, plugin for different logics
- Java PathExplorer, past + future time temporal logic

there is a unifying and practical logic

there is no such!

a unifying monitoring logic?



our goal

- many different logics and automata used for trace monitoring
- which one should we use?
- goal: develop a general purpose rule-based system that can encode well-known logics - a “byte-code” for monitoring logics
- into which one can easily compile syntactically richer logics
- should go beyond regular
- in which one can program monitors
- with efficient monitoring algorithm

from Eagle to RuleR

our two attempts to define such a logic:

- Eagle : recursion-based
  - linear mu-calculus with past time, seq. comp, data
  - convenient logic
  - difficult to implement
- RuleR : rule-based
  - inspired by rule systems
  - less convenient logic at first sight
  - basis for building convenient logic
  - easy to implement

formalizing concept of monitor

Alphabet:

Given an alphabet (set) A of symbols, also called observations

Language:

A language $L \subseteq A^*$ over A is a subset of A^*

Property:

A property P over A is a language:

$$P \subseteq A^*$$

examples

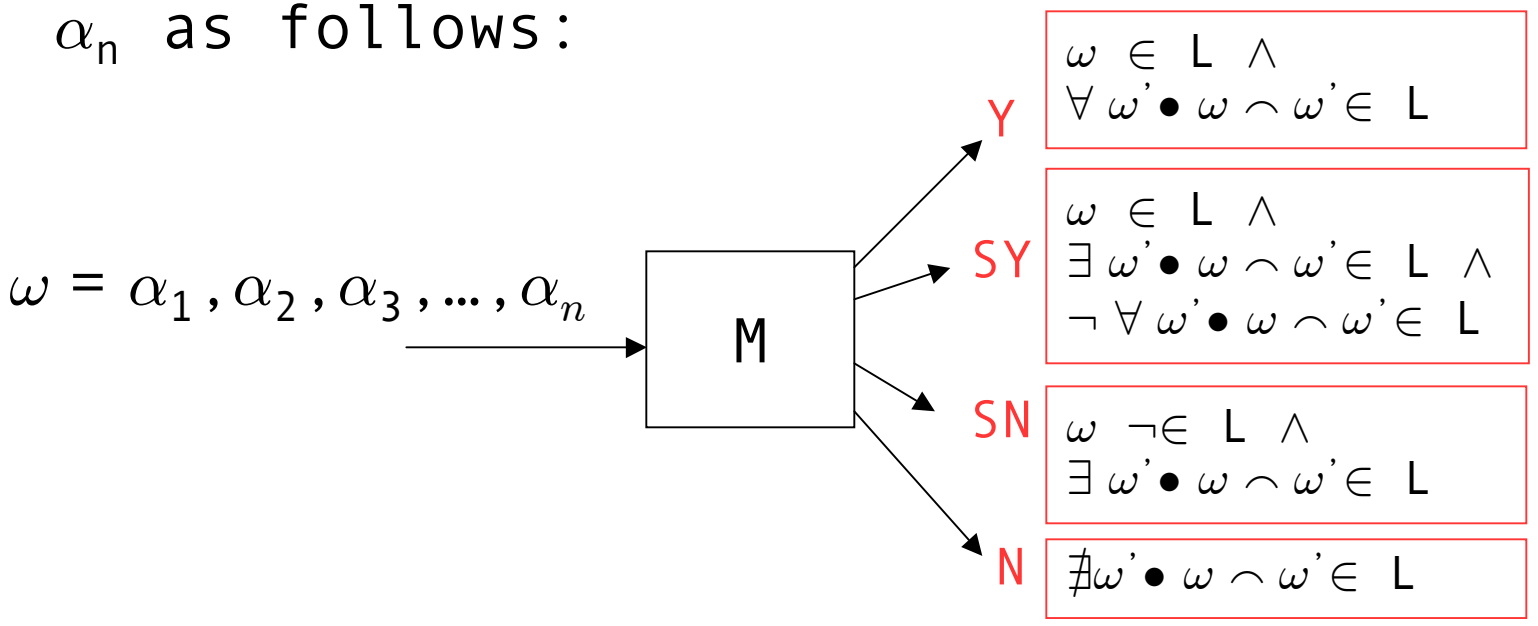
formula φ	language $L(\varphi)$
$\Box(a \rightarrow \Diamond b)$	$\{\epsilon, cc, b, ab, aab, \ldots\}$ liveness
$\Box(a \rightarrow \blacklozenge b)$	$\{\epsilon, c, ba, baa, baba, \ldots\}$ safety
$(ab)^*$	$\{\epsilon, ab, abab, ababab, \ldots\}$
$S \rightarrow a\ S\ b \mid \epsilon$	$\{\epsilon, ab, aabb, aaabbb, \ldots\}$

a formula in a logic compacts an infinite language

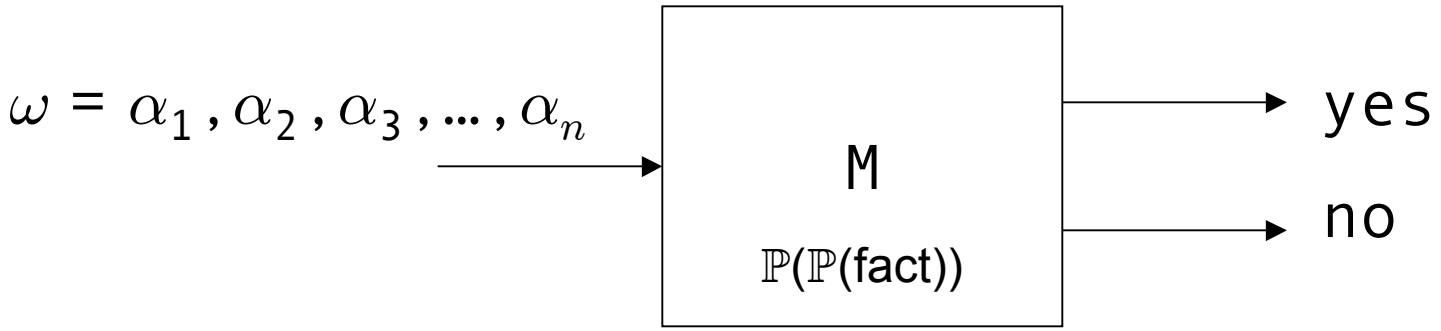
monitoring a language

A monitor:

A monitor M for a language $L \subseteq A^*$ is a "box" which as input takes a list of symbols $\alpha \in A$, one by one, and emits a value for the next symbol α_n as follows:



fact = observation | rule



$$r : f, \ldots, f \rightarrow f, \ldots, f \mid \ldots \mid f, \ldots, f$$

where each fact f is an observation or a rule, potentially negated.

definition

- A rule system is a tuple: <R,O,P,I, F>, where
  - R is a set of rule names
  - O is a set of observations
  - P is a set of rules (r in P)
  - I is an initial set of rule activations
  - F is a set of forbidden rules

- A rule r is a pair <C,B>
  - C is a conjunctive set of (R ∪ O)-literals
  - B is a disjunctive set of conjunctive sets of (R ∪ O)-literals

algorithm

create initial set of initial rule activation states

while observations exist **do**
  obtain next observation state;
  merge observation state with the fact sets;
  raise monitoring fault if there's conflict in all fact sets;
  for each resulting fact set **do**
    apply activated rules to generate successor set of fact sets
  od;
  union successor sets to form the new frontier of fact sets
od

temporal equations

always a:

$$[]a = a \text{ and next } []a$$

eventually b:

$$<>b = b \text{ or next } <>b$$

a until b:

$$a \ U \ b = b \text{ or } (a \text{ and next } a \ U \ b)$$

RuleR examples

always a

```
r : -> a,r
```

```
r1 : !a -> Fail
r2 : a -> r1,r2
```

eventually b

```
r : -> b | r
forbidden r
```

a until b

```
r : -> b | a,r
forbidden r
```

RuleR examples

sofar a

$r : a \rightarrow r$

previously b

$r_b : \rightarrow r_b$
$r_1 : !b \rightarrow r_1, r_2$
$r_2 : b \rightarrow r_b$

a since b

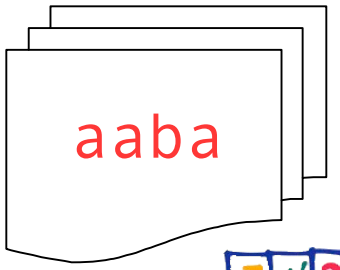
$r_s : a \rightarrow r_s$
$r_1 : \rightarrow r_1, r_2$
$r_2 : b \rightarrow r_s$

translation for TL formula

$$\Box((a \wedge \odot a) \Rightarrow \Diamond b)$$

r_0 : -> r_0, r_1, r_3
r_1 : a -> r_2
r_2
r_3 : a, r_2 -> b | !b, r_4
r_4 : -> b | !b, r_4

initial r_0, r_1, r_3
forbidden r_4



observation 1 : a

r_0 : -> r_0, r_1, r_3
r_1 : a -> r_2
r_3 : a, r_2 -> b | !b, r_4

a

$\Rightarrow$

r_0, r_1, r_3
r_2

observation 2 : aa

r_0 : -> r_0, r_1, r_3
r_1 : a -> r_2
r_2
r_3 : a, r_2 -> b | !b, r_4

a

$\Rightarrow$

r_0, r_1, r_3
r_2
b | !b, r_4

observation 3 : aab

r_0 : -> r_0, r_1, r_3
r_1 : a -> r_2
r_2
r_3 : a, r_2 -> b | !b, r_4
b

b

⇒

r_0, r_1, r_3

r_0 : -> r_0, r_1, r_3
r_1 : a -> r_2
r_2
r_3 : a, r_2 -> b | !b, r_4
!b
r_4 : -> b | !b, r_4

b

⇒

observation 4: aaba

r_0 : -> r_0, r_1, r_3
r_1 : a -> r_2
r_3 : a,r_2 -> b | !b,r_4

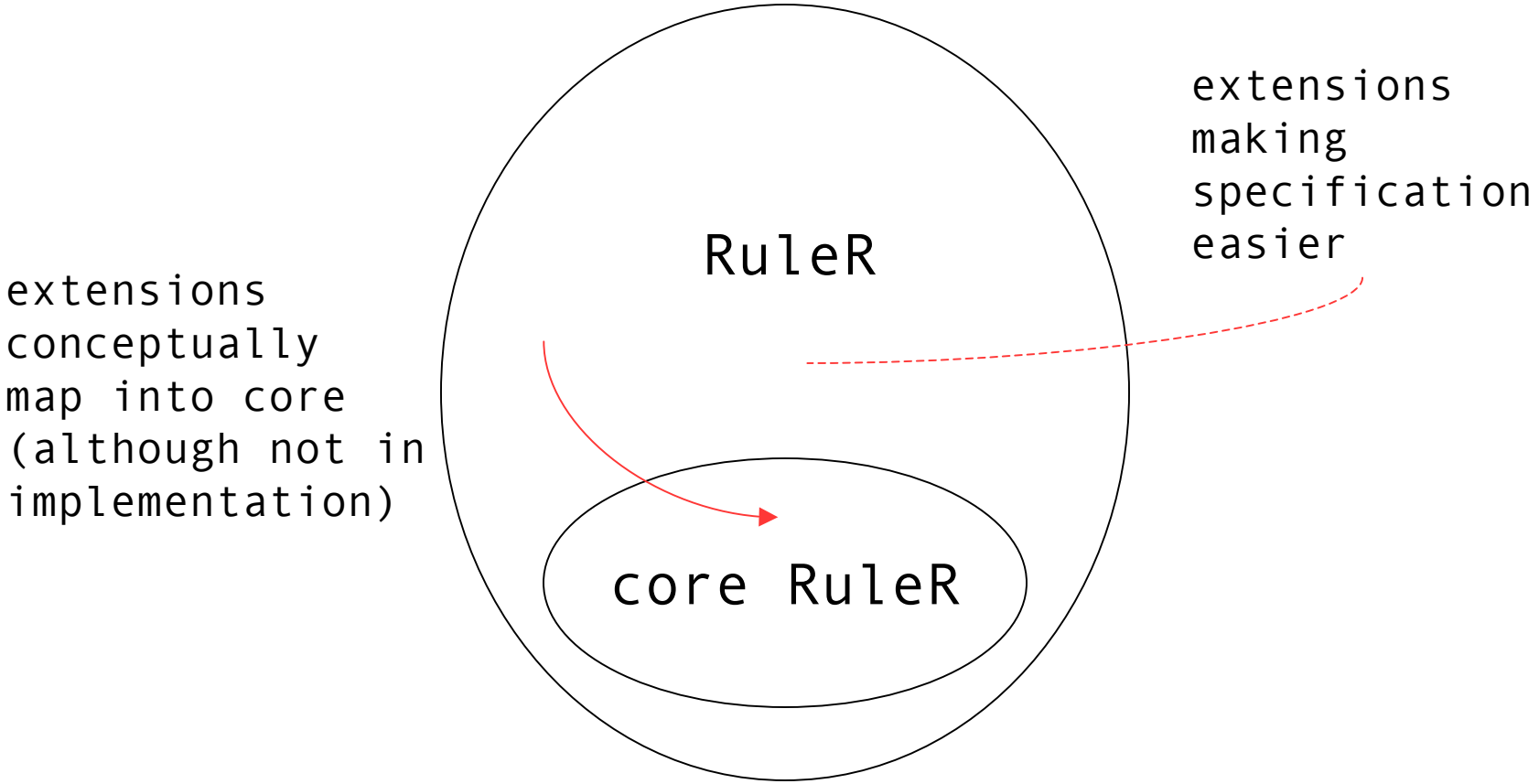
a

$\Rightarrow$

r_0, r_1, r_3
r_2

forbidden r_4

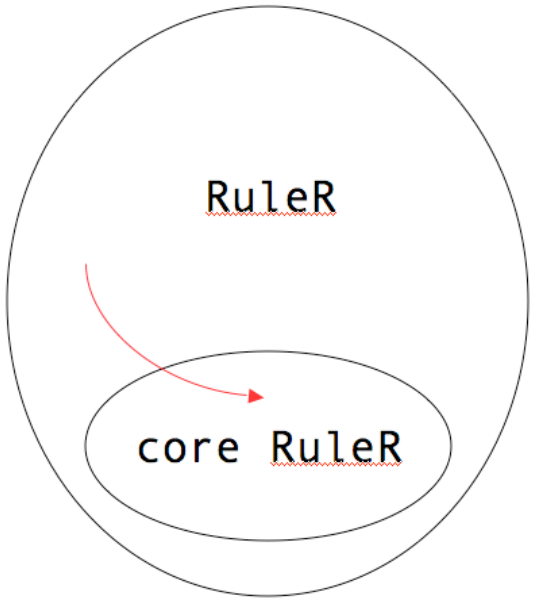
RuleR's core and extensions



```
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 }
}
```

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

extending core-RuleR with three kinds of rule declarations



```
ruler RegularPattern {
    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 }
}
```

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

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

reformulation of spec

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

```
ruler RegularPatternV2 {
  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 }
}
```

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

```
ruler CountingABC {
  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 this:

$(a^n\ b^{2n}\ c^{3n})^+$

```
ruler RegularPatternV3 {
    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.

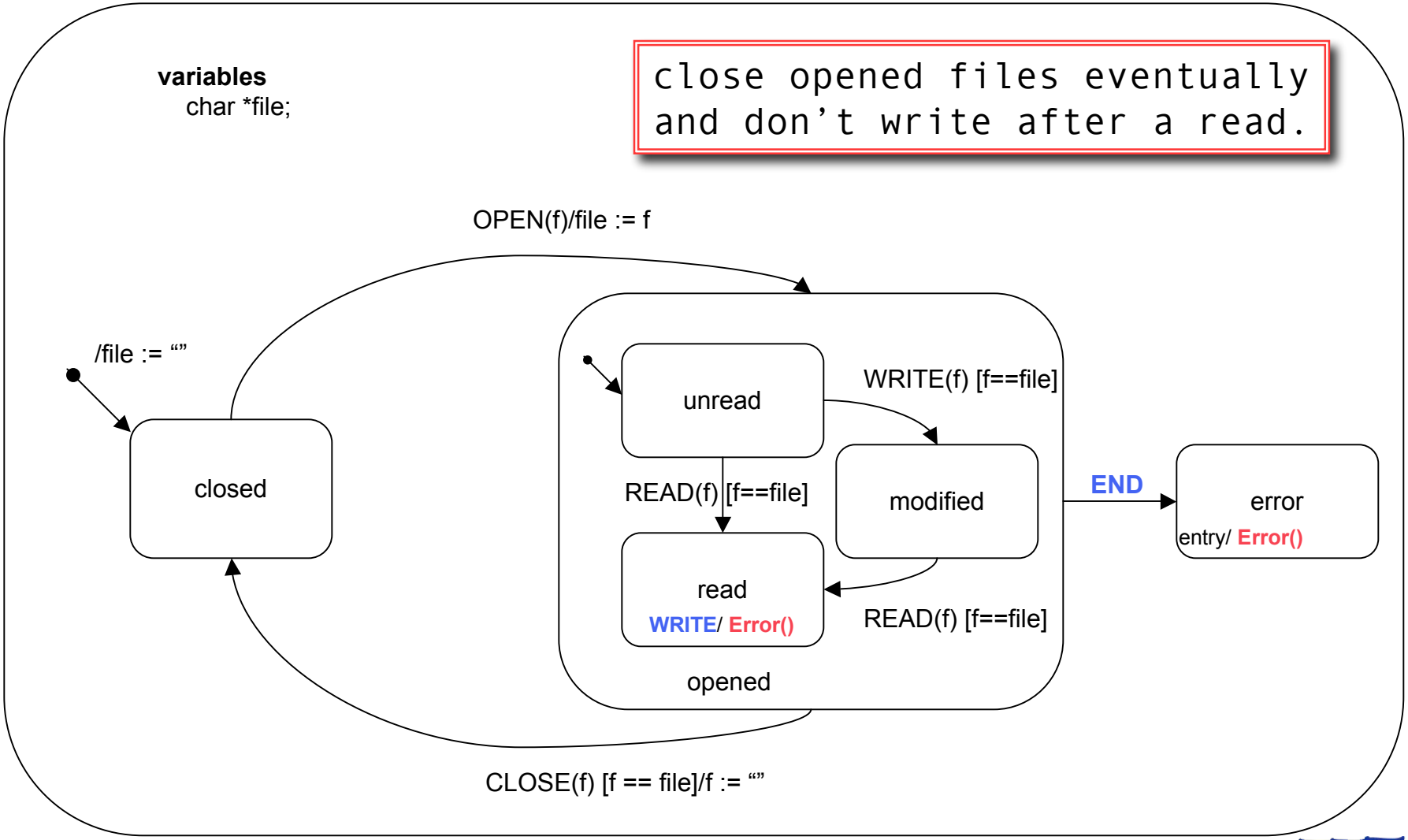
```
ruler Always(y) {
  rules {
    always R(x) {
      x -> Ok;
      !x -> Fail;
    }
  }
  initials {y,R(y)}
}

ruler Eventually(y) {
  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}
  rules {
    always R {
      g -> A(a);
      h -> E(b);
    }
  }
  initials { R }
}
```

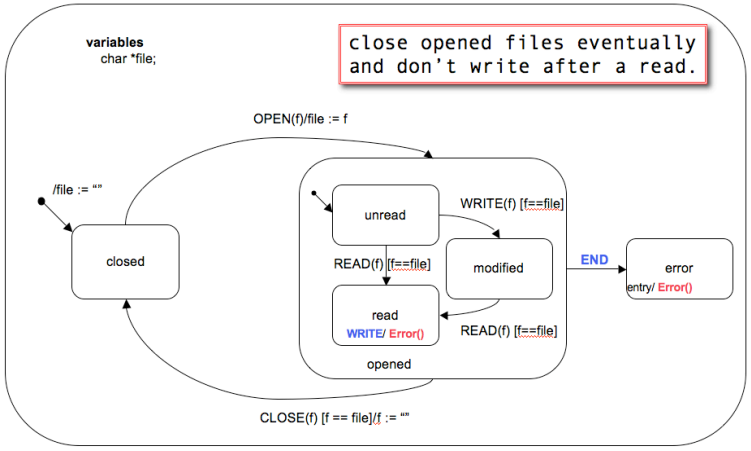
inspired by UML state charts



```
ruler FileMonitor {
  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.



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) {
  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) }
```

```
ruler SimpleCFLV2 {
  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.

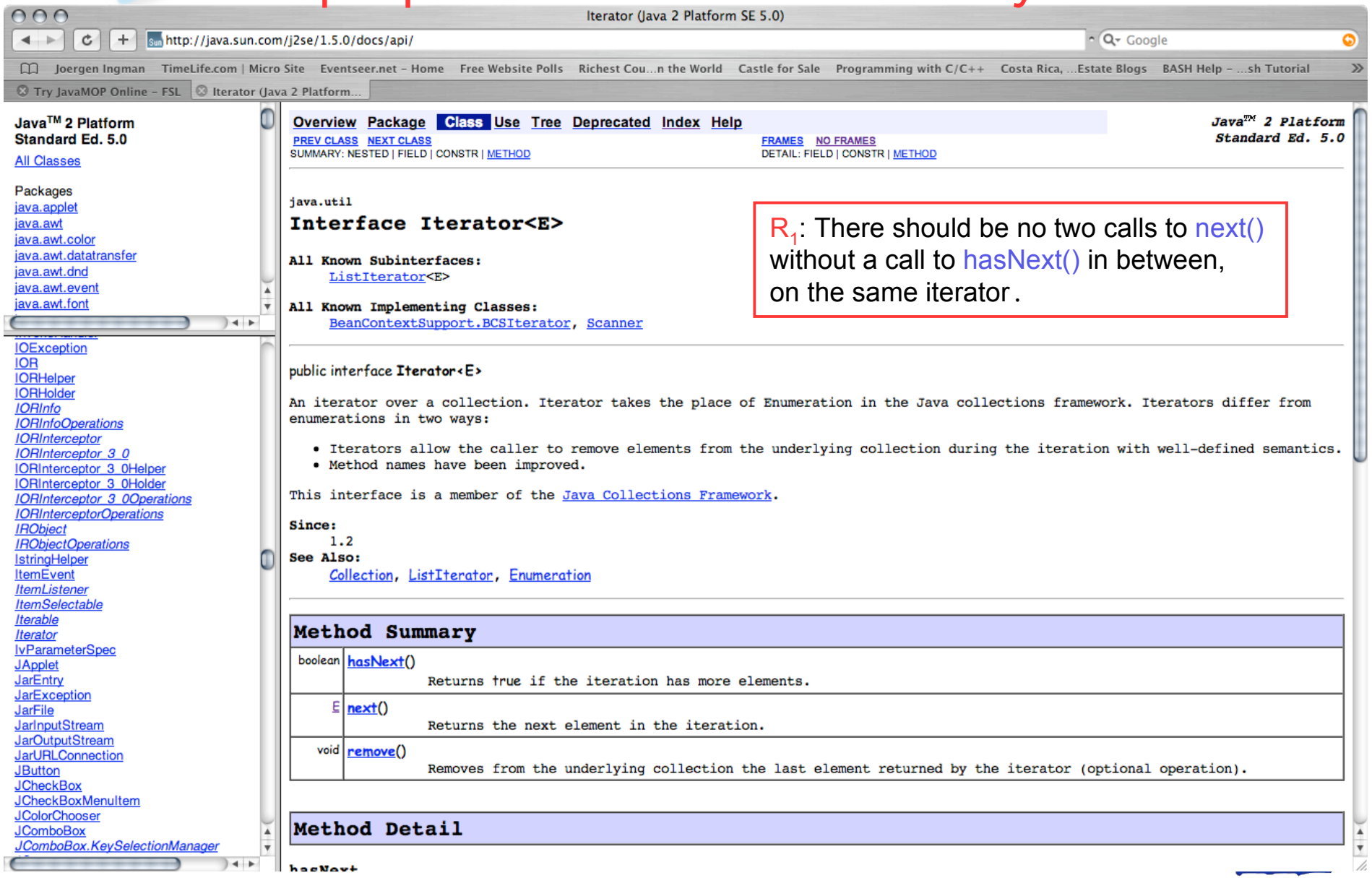
```
ruler SimpleCFLV2 {
  rules {
    step S {
      lock -> L(E);
    }
    step L(c) {
      lock -> L(U(c));
      unlock -> c;
    }
    step U(c) {
      unlock -> c;
    }
    step E {
        -> Fail;
    }
    assert { S, L, U }
  }

  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

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.

use of iterators

There should be no two calls to Iterator.next() without a call to Iterator.hasNext() in between, on same iterator

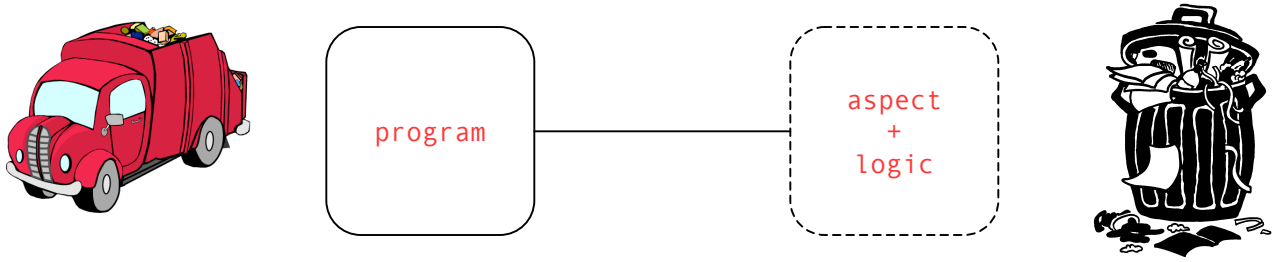
```
public class Associations {
  static Map<String,String> assoc = new HashMap();

  public void associate(Vector<String> words) {
    Iterator it = words.iterator();
    while(it.hasNext()) {
      String w1 = (String)it.next();
      String w2 = (String)it.next();
      assoc.put(w1,w2);
    }
  }

  …
}
```

error

```
aspect HasNextPolicy1 {
  WeakIdentityHashMap monitors = new WeakIdentityHashMap();

  pointcut createiter():
    call(* java.util.Collection+.iterator());

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

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

  after() returning (Iterator it): createiter() {
    monitors.put(it,false);
  }

  before(Iterator it): hasNext(it) {
    monitors.put(it,true);
  }

  before(Iterator it): next(it) {
    v.v((Boolean)monitors.get(it) == true , "hasNext not called before next");
    monitors.put(it,false);
  }
}
```

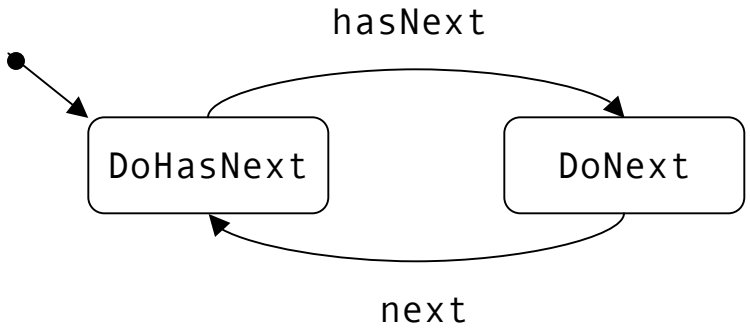
weak & identity

weak identity hash maps

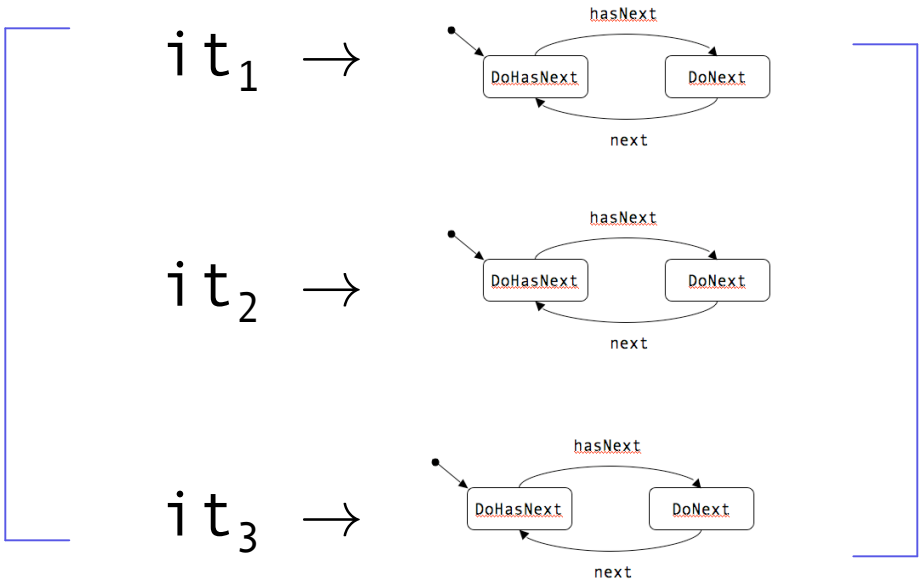


- hasmap an identity hashmap (using `==‘)
- a normal (identity) hashmap keeps a mapping until it is explicitly deleted with `Map.remove()’.
- this becomes a problem since the monitor will then accumulate bindings between iterators and state machines. The garbage collector cannot collect the iterators when they are no longer used by the monitored application.
- a weak collection releases an object to the garbage collector when it is no longer used by any other part of the program.

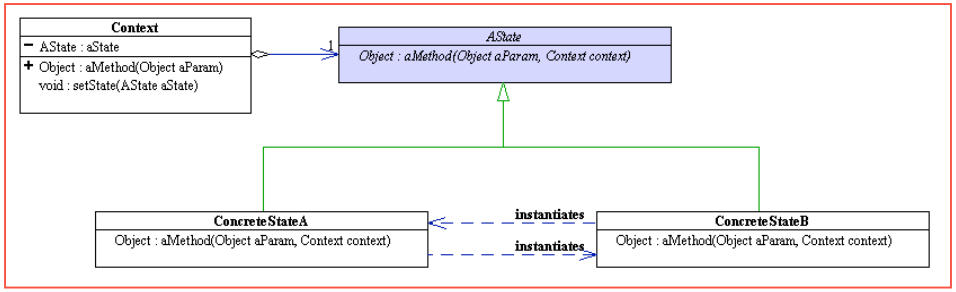
slightly stronger property expressed as a state machine



we need a state machine per iterator



```
class Machine {
    State state = State.doHasNext;
    void hasNext() {
        state = state.hasNext();
    }
    void next() {
        state = state.next();
    }
}
```



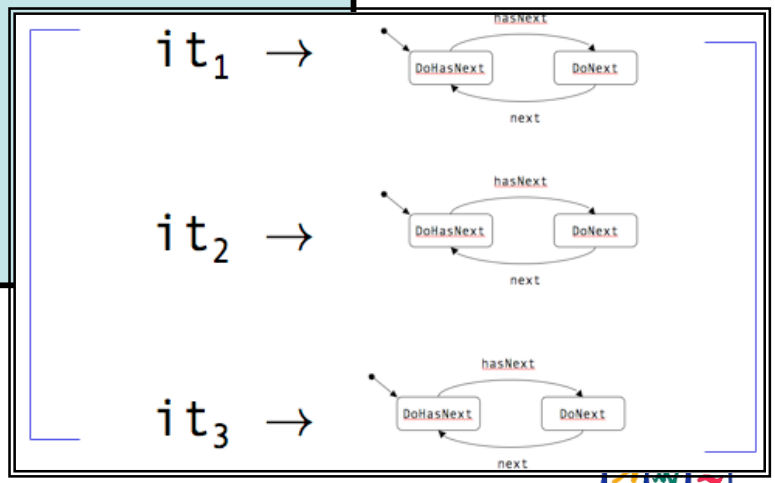

```
class State {
    static final State doNext = new DoNext();
    static final State doHasNext = new DoHasNext();

    State hasNext(){
        System.out.println("*** warning: hasNext called unnecessarily");
        return this;
    }

    State next(){
        System.out.println("*** error: next called illegally");
        return this;
    }
}
```

```
class DoHasNext extends State {
  State hasNext() {
      return doNext;
  }
}
```

```
class DoNext extends State {
    State next() {
        return doHasNext;
    }
}
```

the aspect

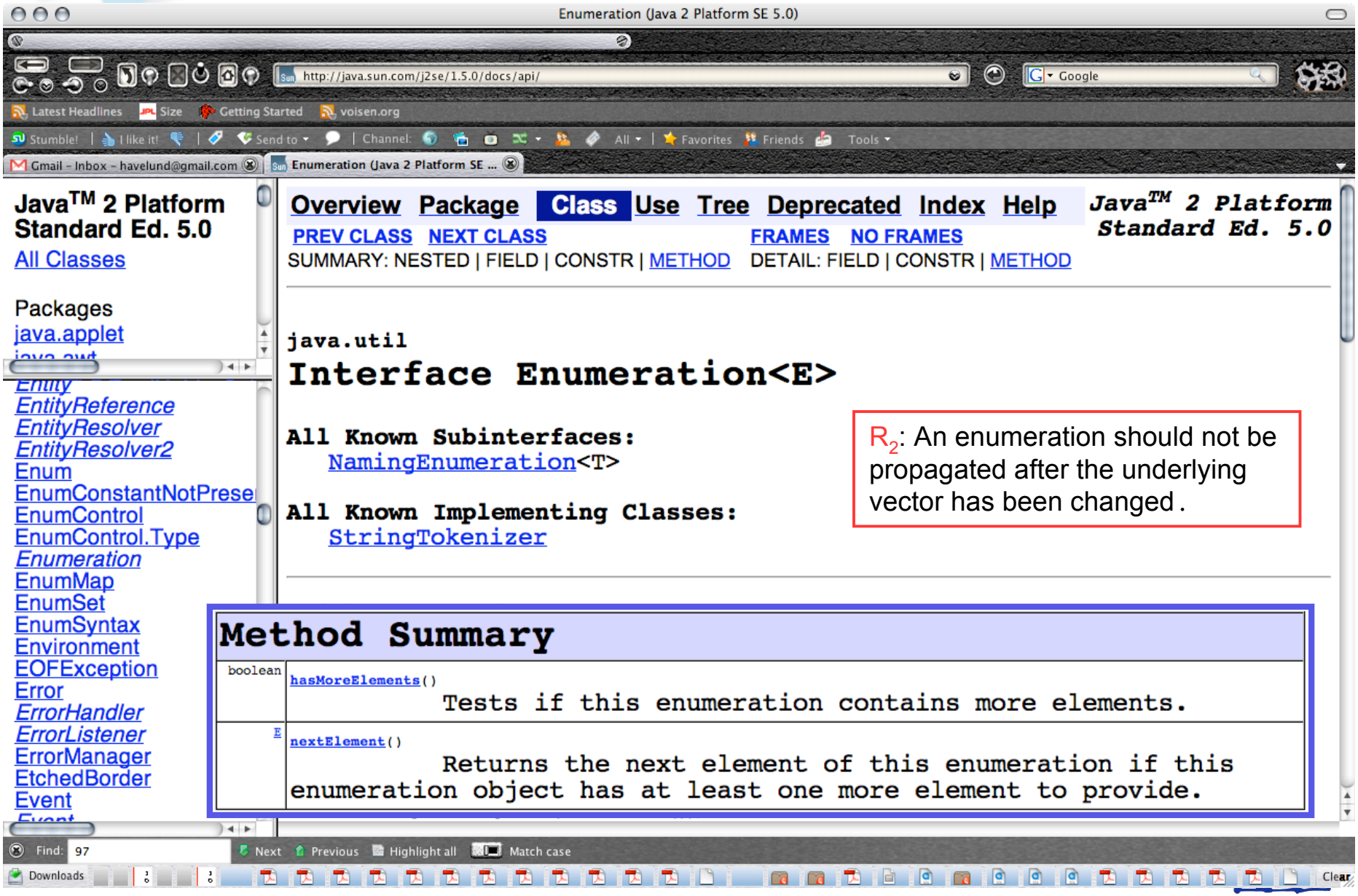
```
aspect HasNextPolicy2 {
    WeakIdentityHashMap monitors = new WeakIdentityHashMap();

    pointcut createiter():
        call(* java.util.Collection+.iterator());
    pointcut hasNext(Iterator it):
        call(* java.util.Iterator+.hasNext()) && target(it);
    pointcut next(Iterator it):
        call(* java.util.Iterator+.next()) && target(it);

    after() returning (Iterator it): createiter() {
        monitors.put(it,new Machine());
    }

    before(Iterator it): hasNext(it) {
       ((Machine)monitors.get(it)).hasNext();
    }

    before(Iterator it): next(it) {
        ((Machine)monitors.get(it)).next();
    }
}
```



```
ruler Monitor{
  rules {
    always Start {
      create(i) -> Untested(i);
    }

    state Untested(i) {
      hasnext(i) -> Tested(i);
      next(i) -> Fail;
    }

    state Tested(i) {
      next(i) -> Untested(i);
    }
  }

  initials { Start }
}
```

```
aspect HasNextPolicy3 {
  RuleR ruler = new RuleR("src/hasnext/hasnext", false);

  pointcut createiter():
    call(* java.util.Collection+.iterator());

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

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

  after() returning (Iterator it): createiter() {
    ruler.dispatch("create", new Object[]{it});
  }

  before(Iterator it): hasNext(it) && scope() {
    ruler.dispatch("hasnext", new Object[]{it});
  }

  before(Iterator it): next(it) && scope() {
    ruler.dispatch("next", new Object[]{it});
  }

  after() : execution(static void Associations.main(..)) && scope() {
    ruler.dispatchEnd();
  }
}
```

properties of Java library APIs



enumerators faster than iterators but less safe

"I'd tried using Iterator and Enumeration to compare their performance on a Vector object containing 100 000 Strings. Enumeration was consistently about 50% faster". - *web blog*

An Iterator next() operation throws a:

ConcurrentModificationException

if it detects that the underlying collection has been modified while iteration is underway.

Enumerator does not!

use of enumerators

An enumeration should not be propagated after the undelying vector has been changed.

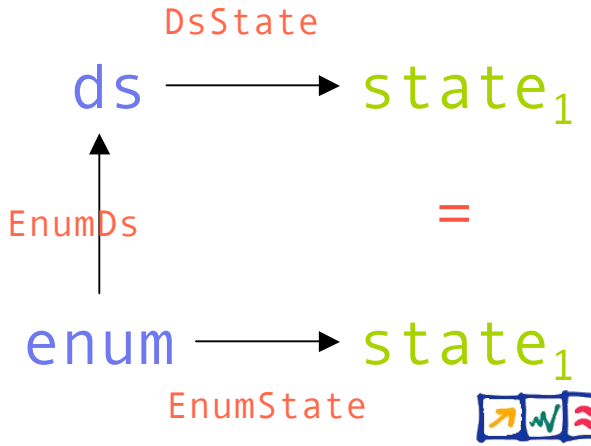
```
public class Associations {
  static Map<String,String> assoc = new HashMap();

  public void associate(Vector<String> words) {
    Enumeration e = words.elements();
    while(e.hasMoreElements()) {
      String w1 = (String)e.nextElement();
      if (!e.hasMoreElements())
        words.add("undefined");
      String w2 = (String)e.nextElement();
      assoc.put(w1,w2);
    }
  }

  …

}
```

error

three maps are needed

- DsState: recording when a data structure was last updated (maps to unique object)
- EnumState: recording the state of the data structure of an enumeration at creation time
- EnumDs: recording what data structure corresponds to what enumeration

$$\begin{aligned} \mathit{DsState} &= \mathit{Ds} \rightarrow \mathit{State} \\ \mathit{EnumState} &= \mathit{Enum} \rightarrow \mathit{State} \\ \mathit{EnumDs} &= \mathit{Enum} \rightarrow \mathit{Ds} \end{aligned}$$

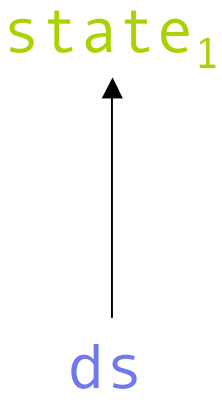


example monitored run

```
public void associate(Vector<String> words) {
   Enumeration e = words.elements();
   while(e.hasMoreElements()) {
     String w1 = (String)e.nextElement();
     if (!e.hasMoreElements())
       words.add("undefined");
     String w2 = (String)e.nextElement();
     assoc.put(w1,w2);
   }
}
words.add("car");
assoc.associate(words);
```

example
monitored
run

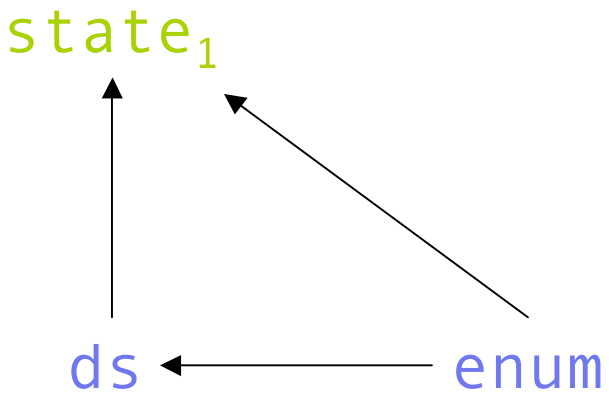
```
public void associate(Vector<String> words) {
    Enumeration e = words.elements();
    while(e.hasMoreElements()) {
      String w1 = (String)e.nextElement();
      if (!e.hasMoreElements())
        words.add("undefined");
      String w2 = (String)e.nextElement();
      assoc.put(w1,w2);
    }
}
words.add("car");
assoc.associate(words);
```



update

example monitored run

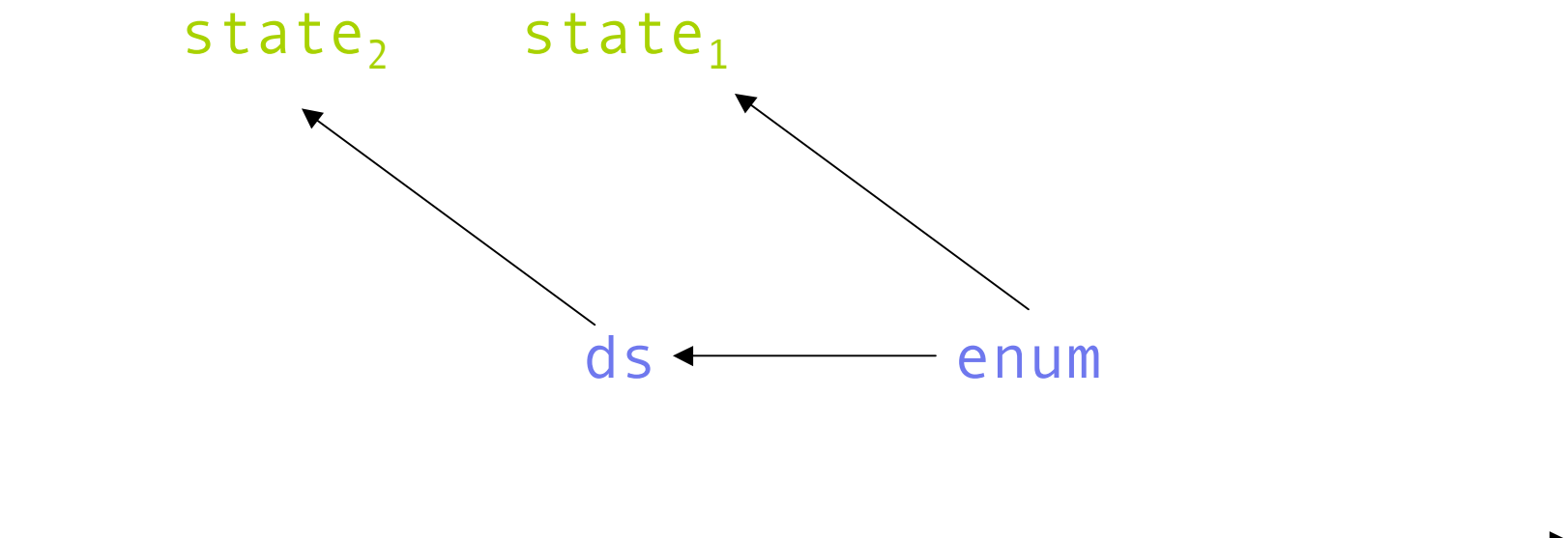
```
public void associate(Vector<String> words) {
   Enumeration e = words.elements();
   while(e.hasMoreElements()) {
     String w1 = (String)e.nextElement();
     if (!e.hasMoreElements())
       words.add("undefined");
     String w2 = (String)e.nextElement();
     assoc.put(w1,w2);
   }
}
words.add("car");
assoc.associate(words);
```



update create

example monitored run

```
public void associate(Vector<String> words) {
   Enumeration e = words.elements();
   while(e.hasMoreElements()) {
     String w1 = (String)e.nextElement();
     if (!e.hasMoreElements())
       words.add("undefined");
     String w2 = (String)e.nextElement();
     assoc.put(w1,w2);
   }
}
words.add("car");
assoc.associate(words);
```



example
monitored
run


```
public void associate(Vector<String> words) {
    Enumeration e = words.elements();
    while(e.hasMoreElements()) {
      String w1 = (String)e.nextElement();
      if (!e.hasMoreElements())
        words.add("undefined");
      String w2 = (String)e.nextElement();
      assoc.put(w1,w2);
    }
}
words.add("car");
assoc.associate(words);
```

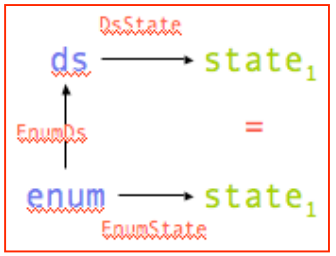
$state_2 \neq state_1$

ds ← enum

update create update next

the checker

```
aspect SafeEnum {
    private Map ds_state   = new WeakIdentityHashMap();
    private Map enum_state = new WeakIdentityHashMap();
    private Map enum_ds    = new WeakIdentityHashMap();

    private static class StateId {}

    pointcut vector_update() :
        call(* Vector.add*(..)) || call(* Vector.clear()) ||
        call(* Vector.insertElementAt(..)) || call(* Vector.remove*(..)) ||
        call(* Vector.retainAll(..)) || call(* Vector.set*(..)) && scope();

    after(Vector ds) returning(Enumeration e) :
        call(Enumeration Vector.elements()) && target(ds) {
        enum_ds.put(e,ds);
        Object s = ds_state.get(ds);
        if (s != null) enum_state.put(e,ds_state.get(ds));
    }

    before(Enumeration e):
        call(Object Enumeration.nextElement()) && target(e) {
        if (ds_state.get(enum_ds.get(e)) != enum_state.get(e))
            error("nextElement called on enumerator after update");
    }

    after(Vector ds) : vector_update() && target(ds) {
        ds_state.put(ds,new StateId());
    }
}
```



```
ruler Monitor {
  rules {
    always Start {
      create(v,e) -> Next(v,e);
    }

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

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

  initials { Start }
}
```

```
public aspect SafeEnumPolicy2 {
  RuleR ruler = new RuleR("src/safeenum/safeenum", false);

  pointcut vector_update() :
    call(* Vector.add*(..)) || call(* Vector.remove*(..)) || …


  after(Vector ds) returning(Enumeration e) :
    call(Enumeration Vector.elements()) && target(ds) {
    dispatch("create", new Object[]{ds,e});
  }

  before(Enumeration e):
    call(Object Enumeration.nextElement()) && target(e) {
    dispatch("next", new Object[]{e});
  }

  after(Vector ds) : vector_update() && target(ds) {
    dispatch("update", new Object[]{ds});
  }

  after() : execution(static void Associations.main(..)) {
    ruler.dispatchEnd();
  }
}
```

conclusions

- RuleR has a simple monitoring algorithm
- implemented in Java
- current effort focuses on re-implementation in C and Python
- implement as library as well as special syntax
- experiments with logfile analysis as part of MSL testing framework