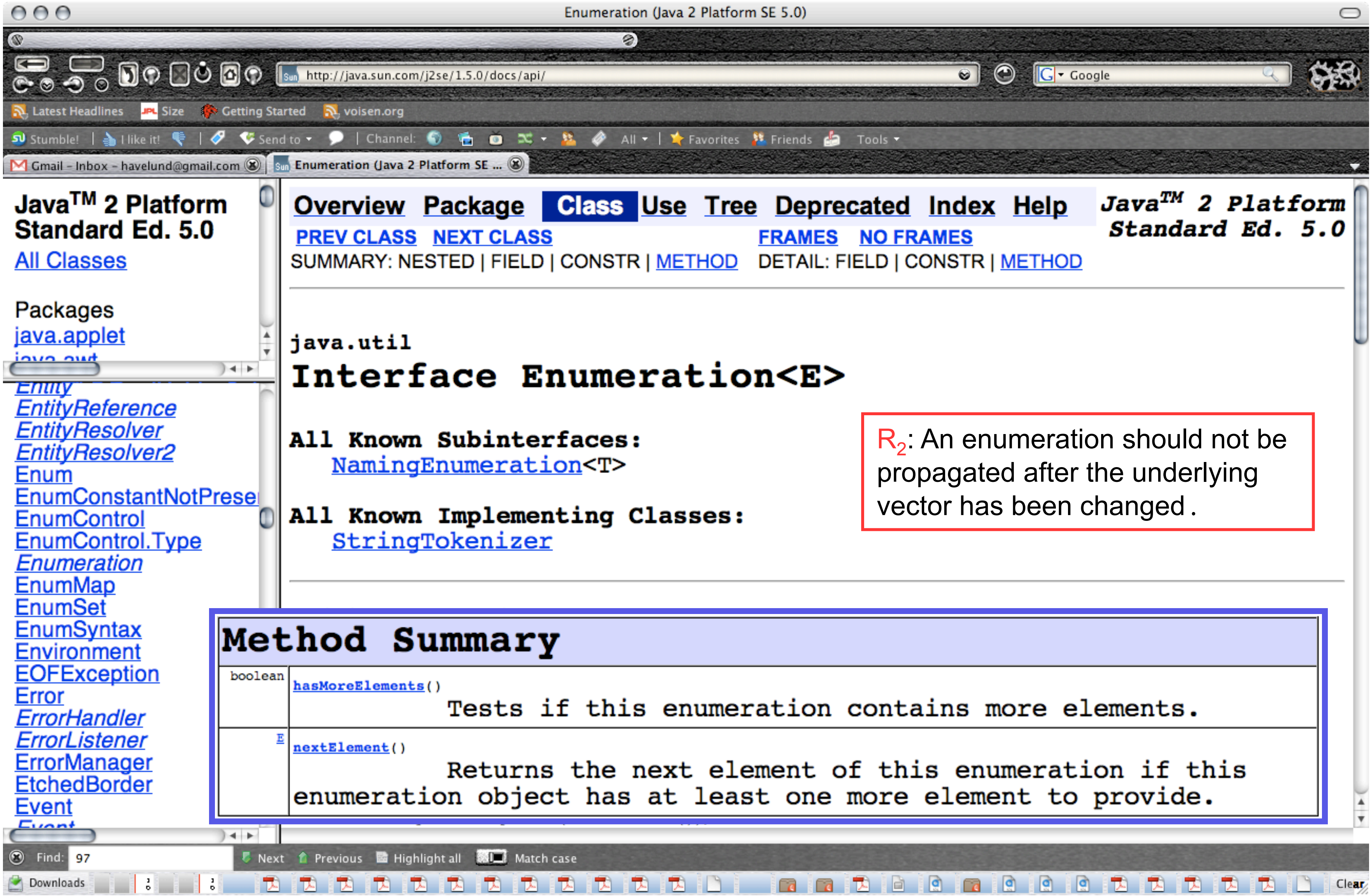
monitoring with state machines & regular expressions

CS 119

`the propositional case`

recall this property



example
monitored
run

```
…
v.add(3);
Enumeration en = v.elements();
while(en.hasMoreElements()) {
    Integer i = (Integer)en.nextElement();
    if (i == 2)
        v.add(4);
    else
        System.out.println(i);
    }
}
```

example
monitored
run

```
…
v.add(3);
Enumeration en = v.elements();
while(en.hasMoreElements()) {
    Integer i = (Integer)en.nextElement();
    if (i == 2)
        v.add(4);
    else
        System.out.println(i);
    }
}
```

ds $\longrightarrow$ $state_1$

update

example
monitored
run

```
…
v.add(3);
Enumeration en = v.elements();
while(en.hasMoreElements()) {
    Integer i = (Integer)en.nextElement();
    if (i == 2)
        v.add(4);
    else
        System.out.println(i);
    }
}
```

$$\begin{array}{ccc} \text{ds} & \longrightarrow & \text{state}_1 \\ \uparrow & & \\ \text{enum} & \longrightarrow & \text{state}_1 \end{array}$$

update create

example
monitored
run

```java
…
v.add(3);
Enumeration en = v.elements();
while(en.hasMoreElements()) {
    Integer i = (Integer)en.nextElement();
    if (i == 2)
        v.add(4);
    else
        System.out.println(i);
    }
}
```

$$
\begin{tikzcd}
\mathrm{ds} \arrow[r] & \mathrm{state}_2 \\
\mathrm{enum} \arrow[u] \arrow[r] & \mathrm{state}_1
\end{tikzcd}
$$

update create update

example
monitored
run

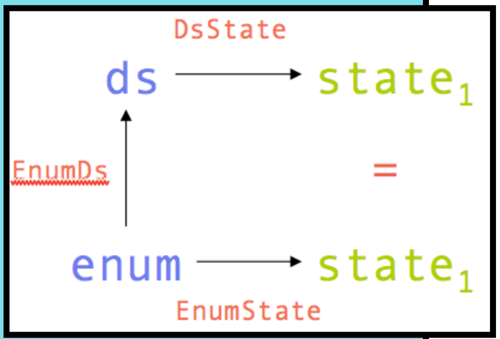
```java
…
v.add(3);
Enumeration en = v.elements();
while(en.hasMoreElements()) {
    Integer i = (Integer)en.nextElement();
    if (i == 2)
        v.add(4);
    else
        System.out.println(i);
    }
}
```

$$
\begin{tikzcd}
\mathrm{ds} \arrow[r] & \mathrm{state}_2 \arrow[d, phantom, "\neq"] \\
\mathrm{enum} \arrow[u] \arrow[r] & \mathrm{state}_1
\end{tikzcd}
$$

update → create → update → next

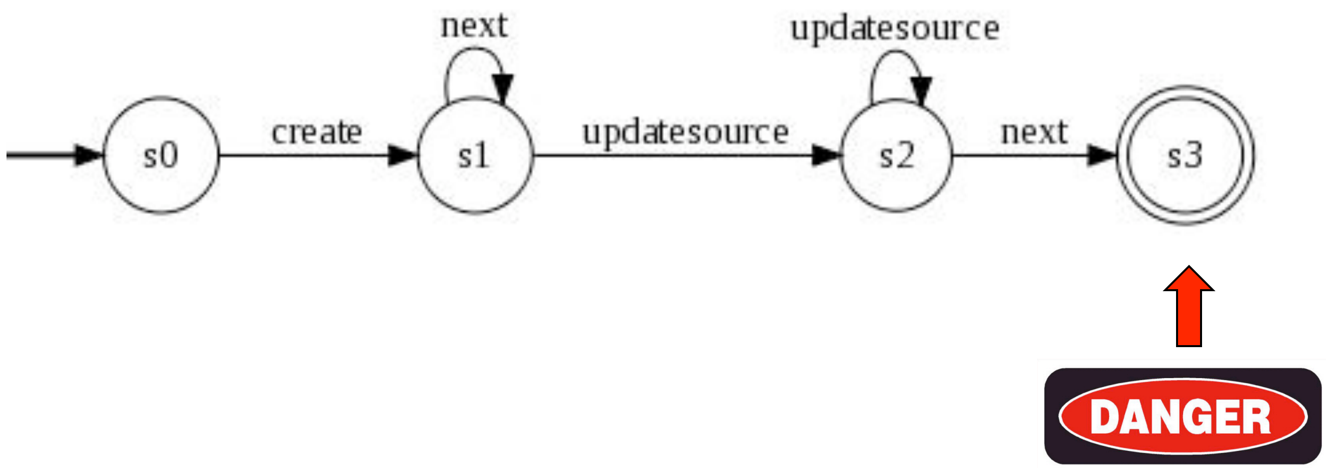
recall AspectJ solution

```
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,s);
    }

    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());
    }
}
```



some programming required!

state machines



systems

JavaMOP

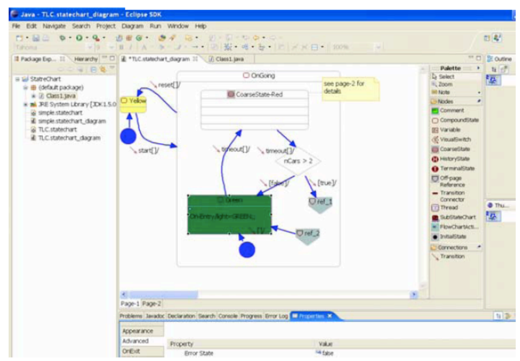
this course

RuleR

this course

commercial

First in Temporal Solutions



The StateRover graphical programming environment supports statecharts, flowcharts, and advanced software design constructs for representing, building, and verifying complex software modules and classes.

Software module specification

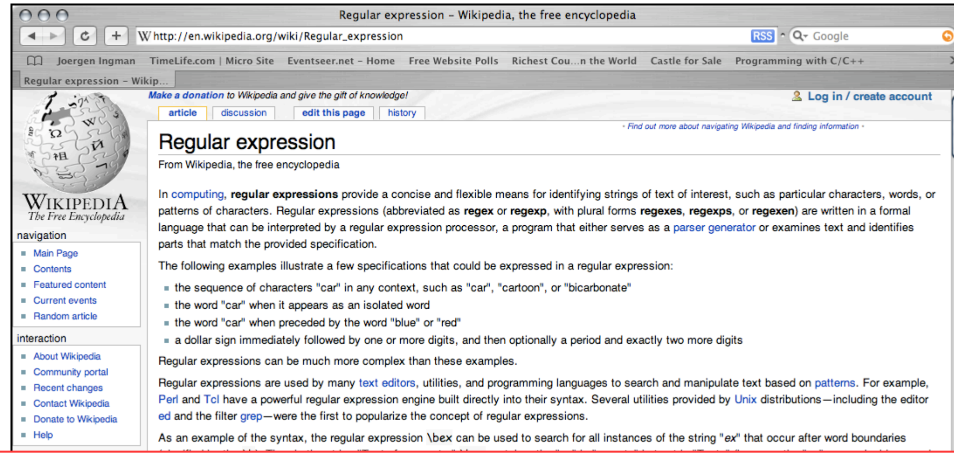
The StateRover provides a broad range of features that enable users to specify, generate, and verify the desired behavior of their software.

Features

- Eclipse graphical user interface for mixed UML statechart and flowchart models and specifications.
- Automatic C, C++, and Java code generation.
- Simultaneous diagram and source level debugging.
- Run-time verification of behavioral requirements.

Benefits

- Intuitive graphical programming tool with automatic modular and portable code generation



Regular expression - Wikipedia, the free encyclopedia

http://en.wikipedia.org/wiki/Regular_expression

Make a donation to Wikipedia and give the gift of knowledge!

Log in / create account

article | discussion | edit this page | history

WIKIPEDIA
The Free Encyclopedia

navigation
- Main Page
- Contents
- Featured content
- Current events
- Random article

interaction
- About Wikipedia
- Community portal
- Recent changes
- Contact Wikipedia
- Donate to Wikipedia
- Help

Regular expression

From Wikipedia, the free encyclopedia

In computing, **regular expressions** provide a concise and flexible means for identifying strings of text of interest, such as particular characters, words, or patterns of characters. Regular expressions (abbreviated as **regex** or **regexp**, with plural forms **regexes**, **regexps**, or **regexen**) are written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

The following examples illustrate a few specifications that could be expressed in a regular expression:

- the sequence of characters "car" in any context, such as "car", "cartoon", or "bicarbonate"
- the word "car" when it appears as an isolated word
- the word "car" when preceded by the word "blue" or "red"
- a dollar sign immediately followed by one or more digits, and then optionally a period and exactly two more digits

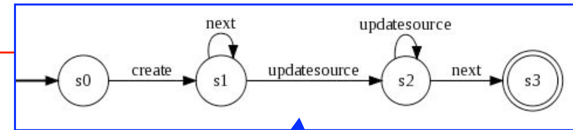
Regular expressions can be much more complex than these examples.

Regular expressions are used by many text editors, utilities, and programming languages to search and manipulate text based on patterns. For example, Perl and Tcl have a powerful regular expression engine built directly into their syntax. Several utilities provided by Unix distributions—including the editor ed and the filter grep—were the first to popularize the concept of regular expressions.

As an example of the syntax, the regular expression \bex can be used to search for all instances of the string "ex" that occur after word boundaries

.	Matches any single character (many applications exclude newlines, and exactly which characters are considered newlines is flavor, character encoding, and platform specific, but it is safe to assume that the line feed character is included). Within POSIX bracket expressions, the dot character matches a literal dot. For example, a.c matches "*abc*", etc., but [a.c] matches only "*a*", ".", or "*c*".
[ ]	A bracket expression. Matches a single character that is contained within the brackets. For example, [abc] matches "*a*", "*b*", or "*c*". [a-z] specifies a range which matches any lowercase letter from "*a*" to "*z*". These forms can be mixed: [abcx-z] matches "*a*", "*b*", "*c*", "*x*", "*y*", and "*z*", as does [a-cx-z]. The - character is treated as a literal character if it is the last or the first character within the brackets, or if it is escaped with a backslash: [abc-], [-abc], or [a\-bc].
[^]	Matches a single character that is not contained within the brackets. For example, [^abc] matches any character other than "*a*", "*b*", or "*c*". [^a-z] matches any single character that is not a lowercase letter from "*a*" to "*z*". As above, literal characters and ranges can be mixed.
^	Matches the starting position within the string. In line-based tools, it matches the starting position of any line.
$	Matches the ending position of the string or the position just before a string-ending newline. In line-based tools, it matches the ending position of any line.
\(\)	Defines a marked subexpression. The string matched within the parentheses can be recalled later (see the next entry, *n*). A marked subexpression is also called a block or capturing group.
n	Matches what the *n*th marked subexpression matched, where *n* is a digit from 1 to 9. This construct is theoretically **irregular** and was not adopted in the POSIX ERE syntax. Some tools allow referencing more than nine capturing groups.
*	Matches the preceding element zero or more times. For example, ab*c matches "*ac*", "*abc*", "*abbbc*", etc. [xyz]* matches "", "*x*", "*y*", "*z*", "*zx*", "*zyx*", "*xyzzy*", and so on. \(ab\)* matches "", "*ab*", "*abab*", "*ababab*", and so on.
\{*m*,*n*\}	Matches the preceding element at least *m* and not more than *n* times. For example, a\{3,5\} matches only "*aaa*", "*aaaa*", and "*aaaaa*". This is not found in a few, older instances of regular expressions.

systems



this course

using AspectJ

```
SafeIterator(Collection c, Iterator i) {

     event create after(Collection c) returning(Iterator i) :
          call(Iterator Collection+.iterator()) && target(c) && BaseAspect.notwithin(){}

     event updatesource after(Collection c) :
          (call(* Collection+.remove*(..)) || call(* Collection+.add*(..)) ) && target(c) && BaseAspect.notwithin(){}

     event next before(Iterator i) :
          call(* Iterator.next()) && target(i) && BaseAspect.notwithin(){}

     ere : create next* updatesource updatesource* next

     @validation {
          System.out.println("improper iterator usage");
     }
}
```

JavaMOP : http://fsl.cs.uiuc.edu/index.php/Special:JavaMOP2.0

```
public aspect FailSafeEnum {

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

     tracematch(Vector ds, Enumeration e) {
        sym create_enum after returning(e) : ((call(* Vector+.elements()) && target(ds)) || (call(Enumeration+.new(..)) && args(ds)));
        sym call_next before :  call(Object Enumeration.nextElement()) && target(e);
        sym update_source before : vector_update() && target(ds);

        create_enum call_next* update_source+ call_next
        {
           if(System.getProperty("TMTEST_ACTIVE")!=null) {
                System.err.println("##### Tracematch "+getClass().getName()+" matched!");
           }
        }
     }
}
```

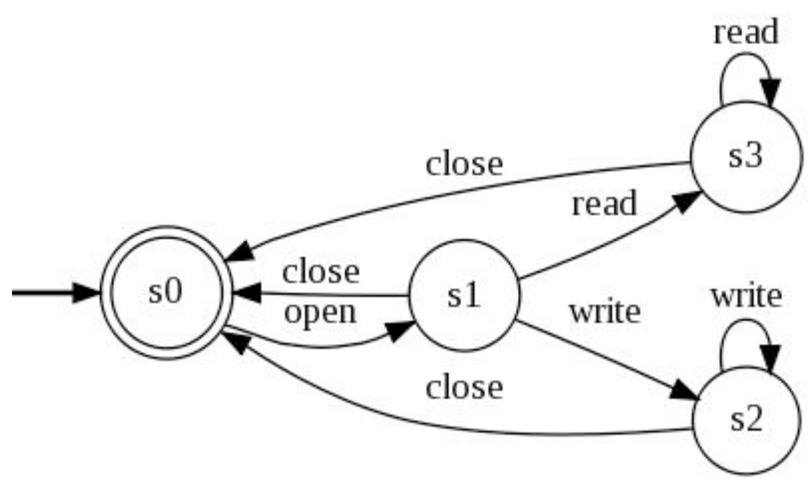
extending AspectJ

Tracematches : http://abc.comlab.ox.ac.uk/papers

regular expression

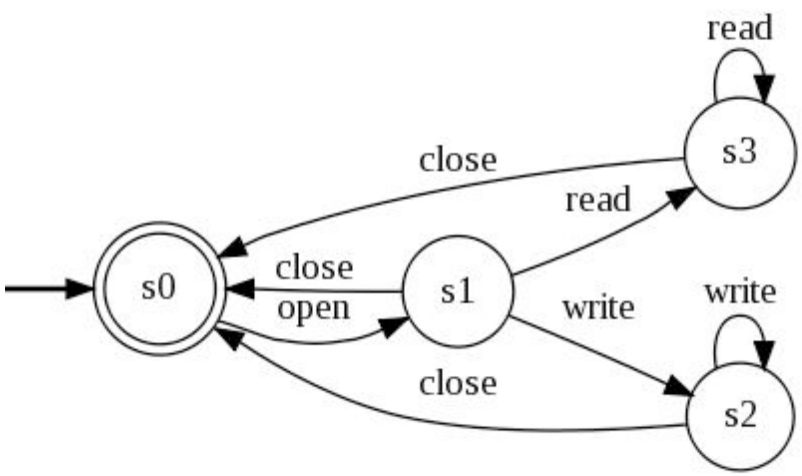
- file access property: a file should be opened, uniformly accessed (either only read from or only written to) zero or more times, and then closed:

```
(open (read* + write*) close)*
```



only propositional specifications in this lecture: no data

state machine



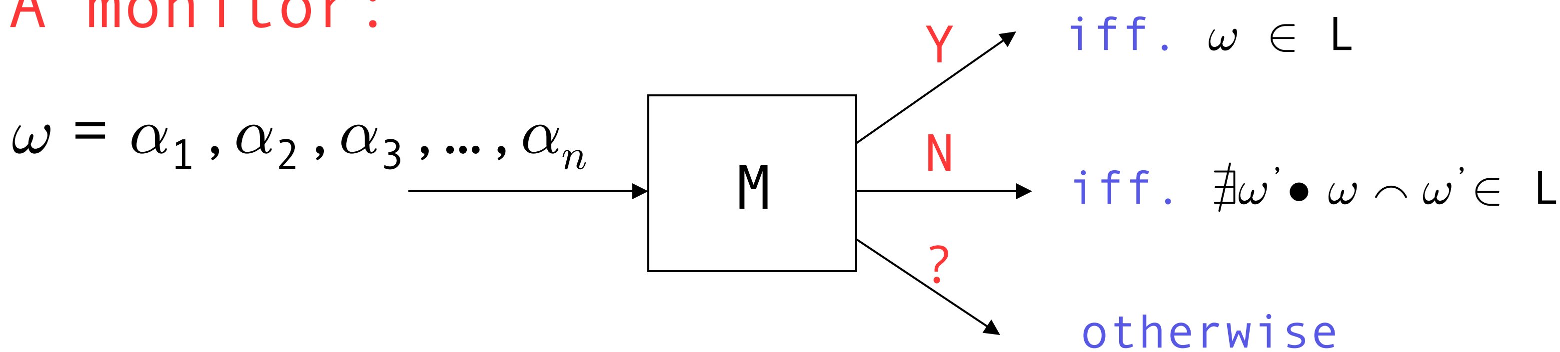

```
fsm:
!s0[
  open -> s1
]
S1[
  read -> s3
  write -> s2
  close -> s0
]
S2[
  write -> s2
  close -> s0
]
s3[
  read -> s3
  close -> s0
]
```

recalling formal setting

Property:

A property P over A is a language: $P \subseteq A^*$

A monitor:



but languages are usually infinite

The monitoring problem:

Given logic $\mathcal{L}$, and formula $\varphi \in \text{Formulas}(\mathcal{L})$,
construct monitor for the language of φ: $M_{L(\varphi)}$

from regular expressions to monitors via state machines

regular expression (RE)

$\rightarrow$ step 1

deterministic finite state automaton (DFA)

$\rightarrow$ step 2

monitor (M)

Step 2 : DFA $\rightarrow$ M

Given an automaton (DFA):

$$\mathcal{A} = \{Q, i \in Q, F \subseteq Q, \sigma : Q \times A \rightarrow Q_{\perp}\}$$

Q : set of states
i : the initial state
F : set of final states
σ : partial transition function

Step 2 : DFA $\rightarrow$ M

Given an automaton (DFA):

$\mathcal{A}$ = {Q, i $\in$ Q, F $\subseteq$ Q, σ:Q$\times$A $\rightarrow$ $Q_\perp$}

Extend σ to A^* as follows:

σ^* : Q $\times$ A^* $\rightarrow$ $Q_\perp$

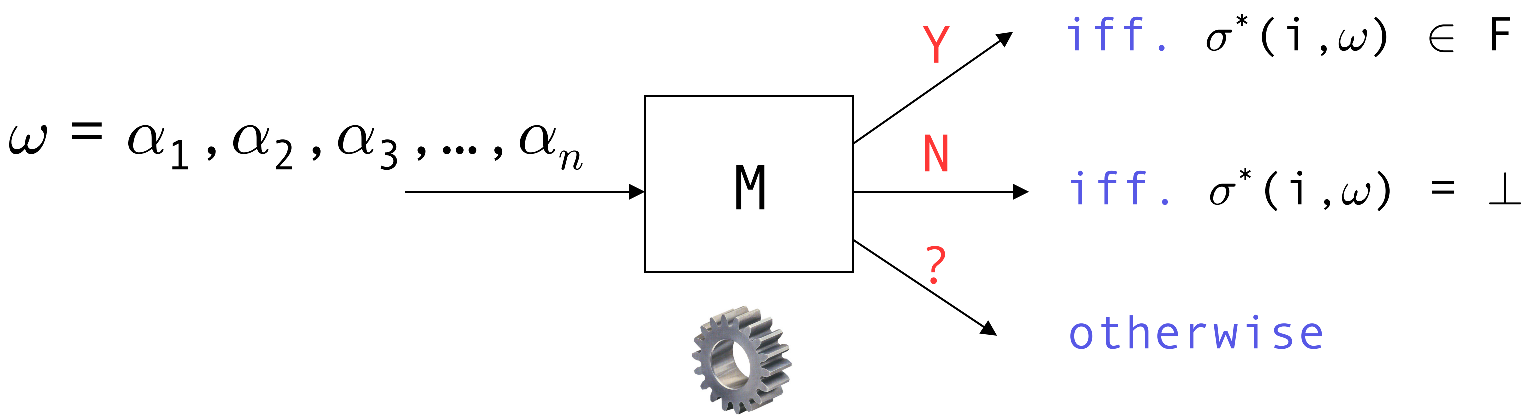
σ^*(q,ϵ) = q
σ^*(q,α) = σ(α)
σ^*(q,$\omega\alpha$) =
    if σ^*(q,ω) $\neq$ $\perp$ **then** σ(σ^*(q,ω),α) **else** $\perp$

Step 2 : DFA $\rightarrow$ M

Given an automaton (DFA):

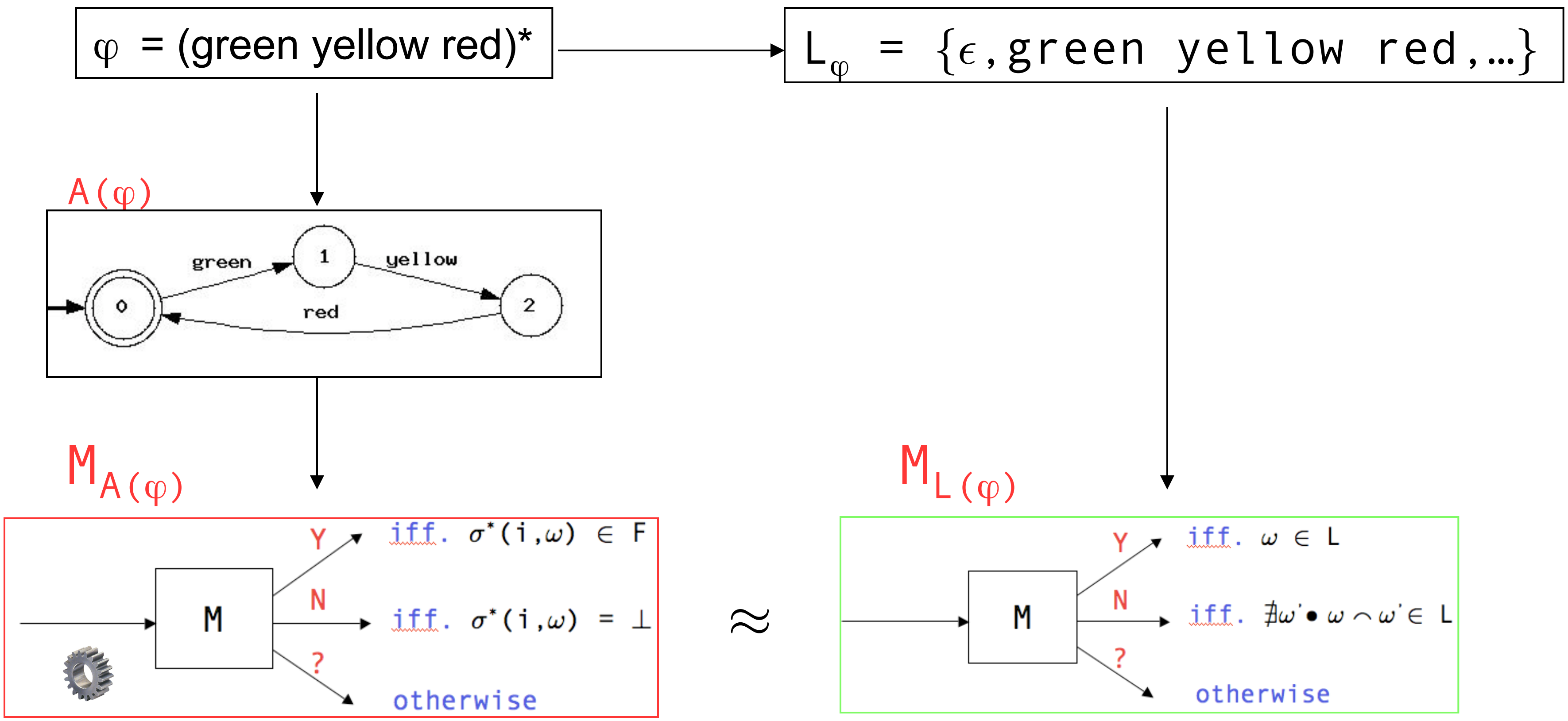
$$\mathcal{A} = \{Q, i \in Q,\ F \subseteq Q,\ \sigma : Q \times A \rightarrow Q_{\perp}\}$$

Define monitor $M_{\mathcal{A}}$:



theorem

$M_{A(\varphi)}$ is equivalent to $M_{L(\varphi)}$



syntax for extended regular expressions

$$E ::= \varnothing \mid \epsilon \mid A \mid E\ E \mid E^* \mid E{+}E \mid E\&E \mid \neg E$$

extended with negation

examples A = {a,b,c}:

aab	{aab}
(ab)*	{ϵ,ab,ababab,…}
$(a+b)^*\ \&\ \neg(ab)^*$	words of randomly interleaved a's and b's but not any words of form: ababab … {a,aa,abba,bbbb,…}

semantics

$L(\varnothing)$	$=$	$\{\}$
$L(\epsilon)$	$=$	$\{\epsilon\}$
$L(\alpha)$	$=$	$\{\alpha\}$
$L(E_1\ E_2)$	$=$	$\{\omega_1\omega_2 \mid \omega_1 \in L(E_1) \wedge \omega_2 \in L(E_2)\}$
$L(E^*)$	$=$	$\{\omega_1\omega_2 \ldots\omega_i\ldots\ \omega_n \mid \omega_i \in L(E)\}$
$L(E_1 + E_2)$	$=$	$L(E_1) \cup L(E_2)$
$L(E_1\ \&\ E_2)$	$=$	$L(E_1) \cap L(E_2)$
$L(\neg\ E)$	$=$	$A^* \setminus L(E)$

from regular expressions to monitors via state machines

recall:

```
regular expression (RE)

→  step 1

deterministic finite state automaton (DFA)

→ step 2

monitor (M)
```

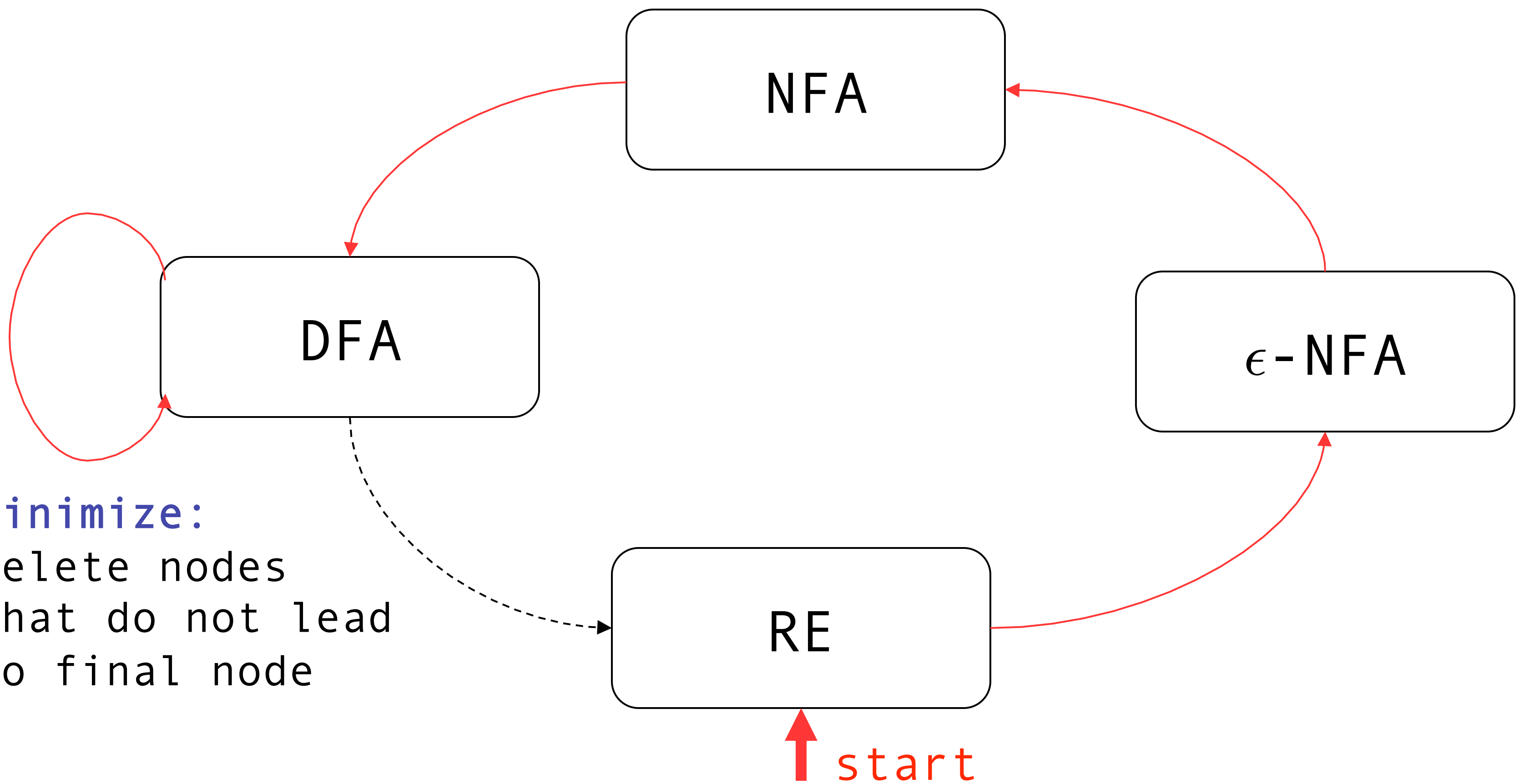
four representations of regular languages

1. regular expressions (RE)
2. non-deterministic ϵ-automata (ϵ-NFA)
3. non-deterministic finite automata (NFA)
4. deterministic finite automata (DFA)

- a DFA denotes a so-called regular language (set of words accepted by the automaton).
- for any specification in any of the above 4 representations, there exist specifications in the other representations denoting the same language. Especially: RE $\rightarrow$ DFA

Step 1 : RE $\rightarrow$ DFA

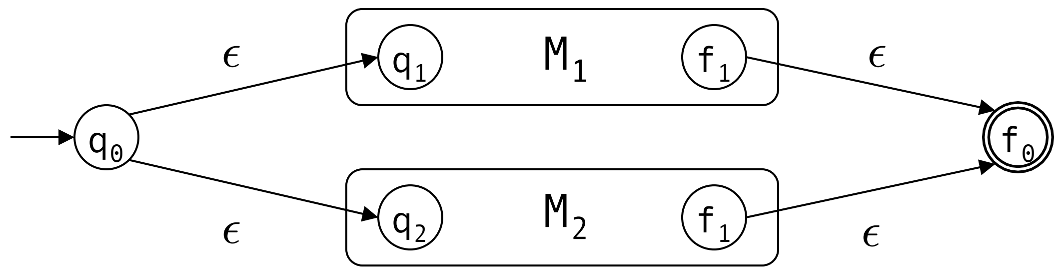
DFA $\rightarrow$ minimized DFA



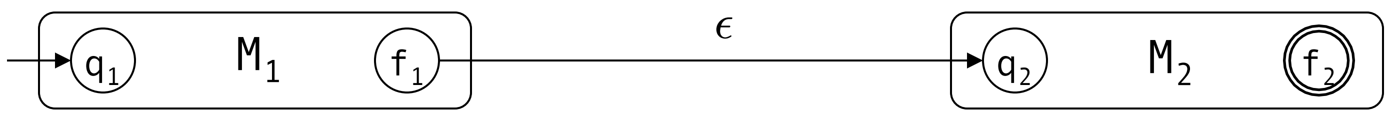
minimize:
delete nodes
that do not lead
to final node

RE $\rightarrow$ ϵ-NFA

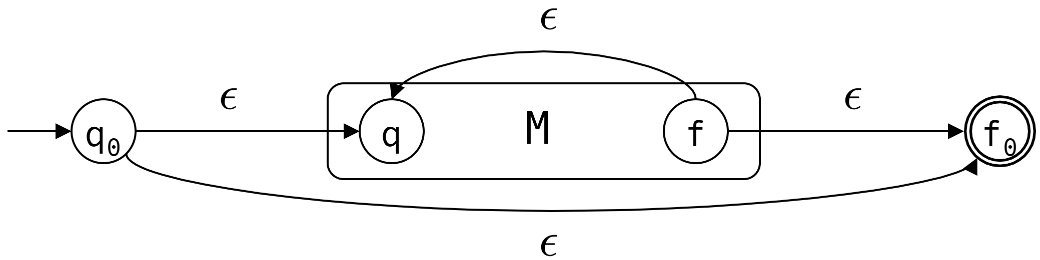
r_1+r_2



$r_1 r_2$

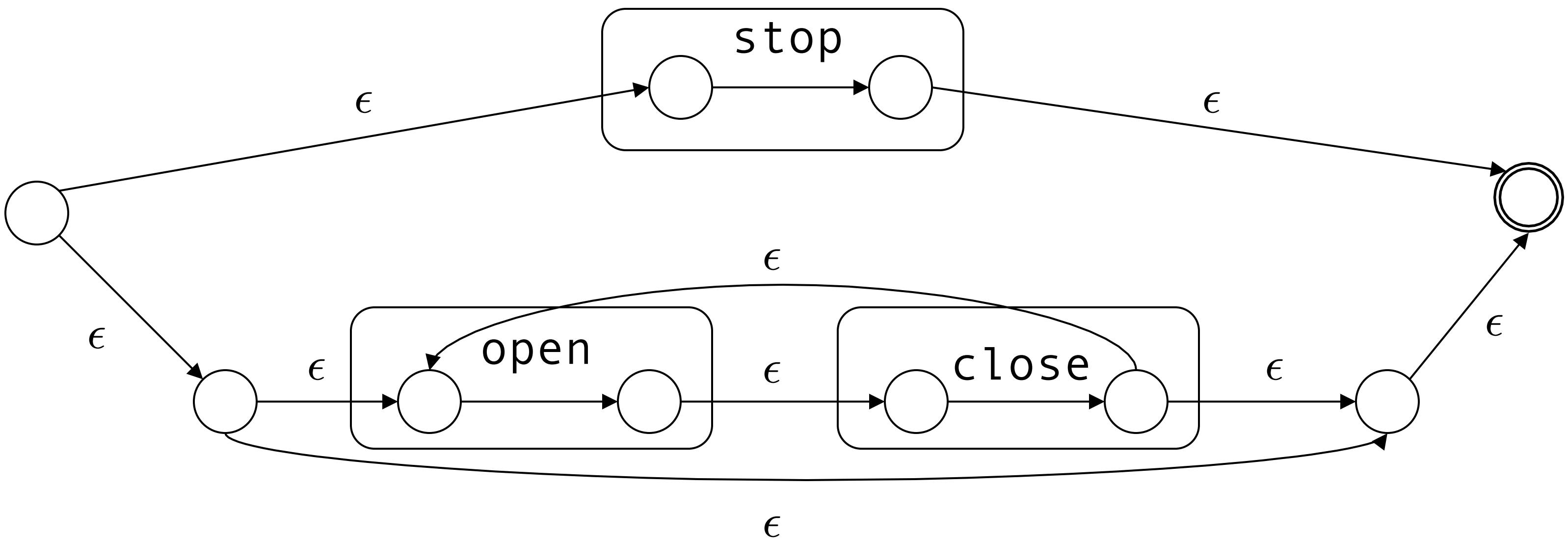


r^*



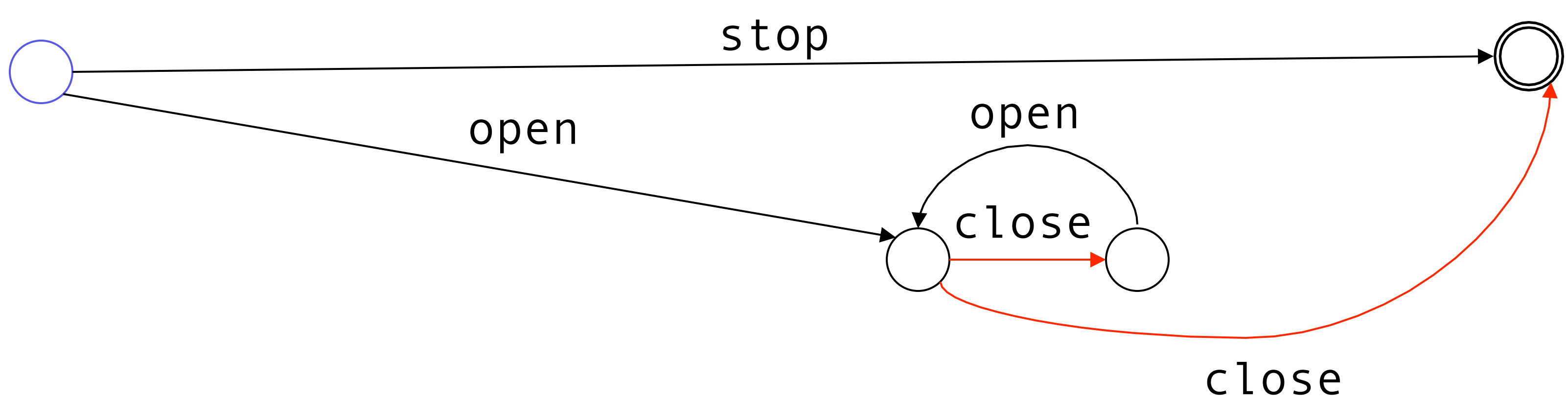
example derivation
RE $\rightarrow$ ϵ-NFA

`(open close)* + stop`



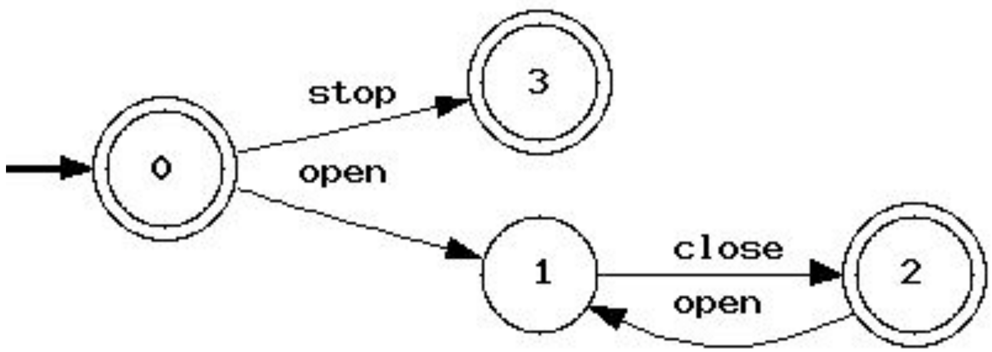
example derivation ϵ-NFA $\to$ NFA

`(open close)* + stop`

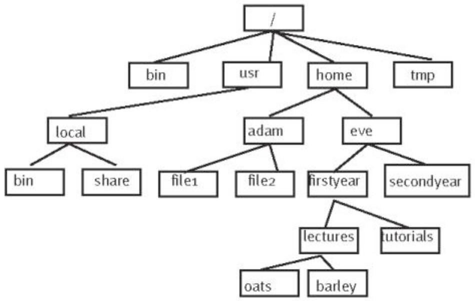


example derivation NFA $\rightarrow$ DFA

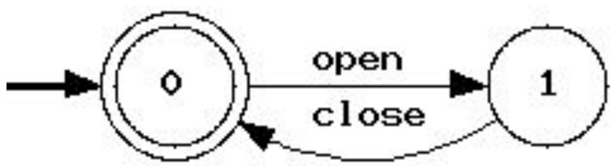
```
(open close)* + stop
```



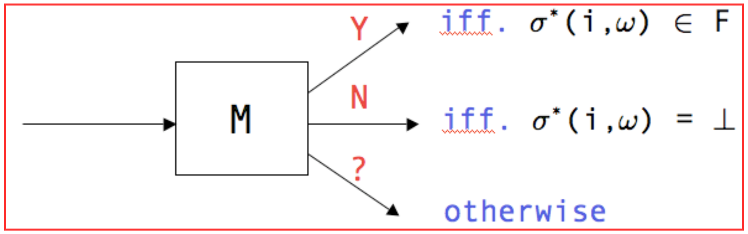
file example revisited



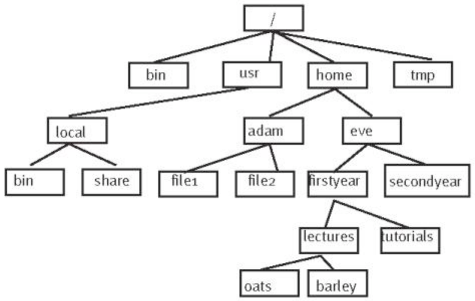
(open close)*



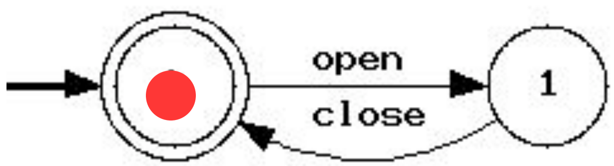
consider the trace:
open close open



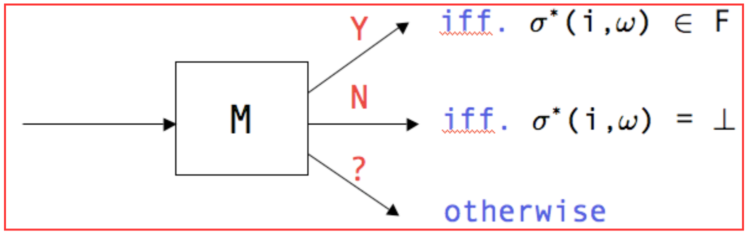
file example revisited



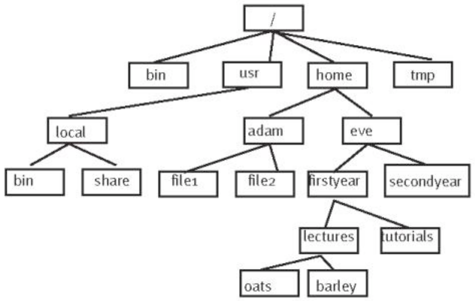
(open close)*



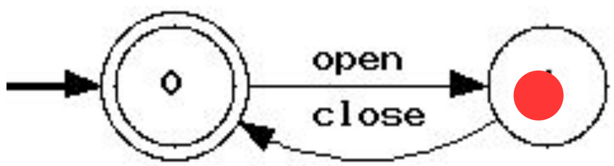
consider the trace:
open close open



file example revisited

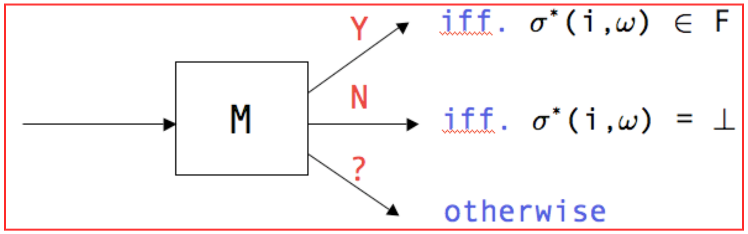


(open close)*



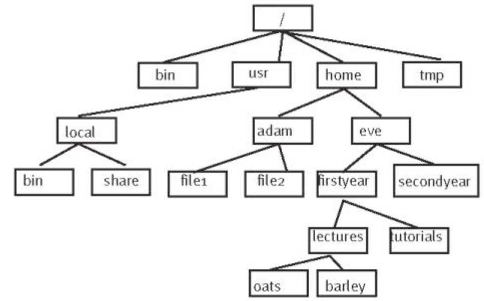
consider the trace:

open close open

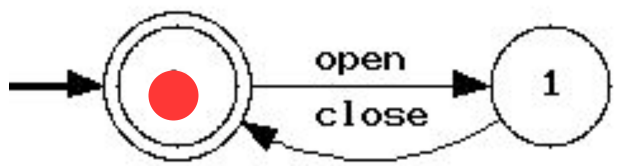


?

file example revisited

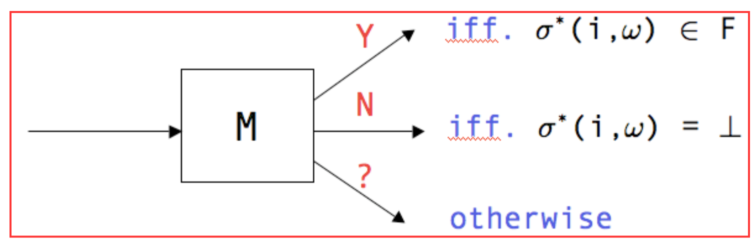


(open close)*



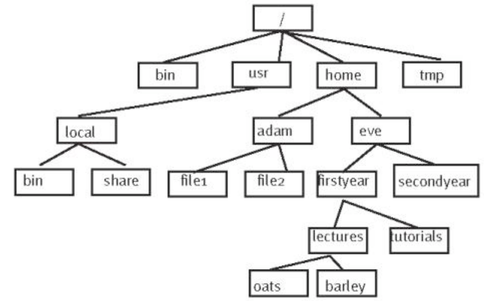
consider the trace:

open close open



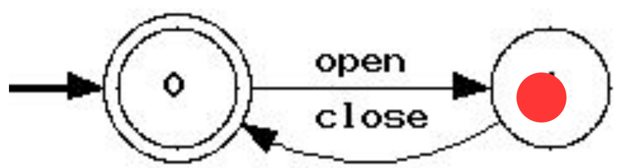
Y

file example revisited



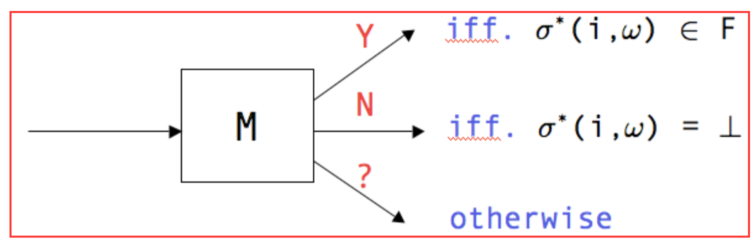
(open close)*

monitored as <u>safety</u> property $\Rightarrow$ no error



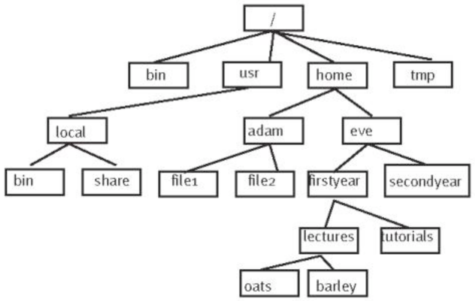
consider the trace:

open close open

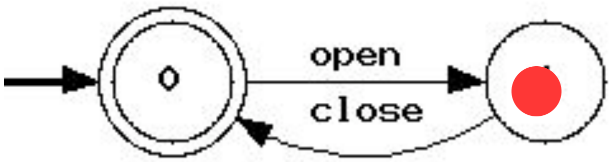


?

file example revisited



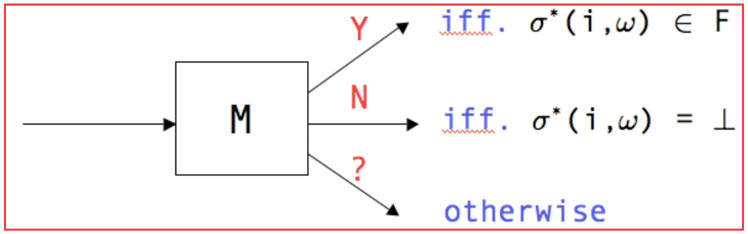
(open close)*



now, assume extra event

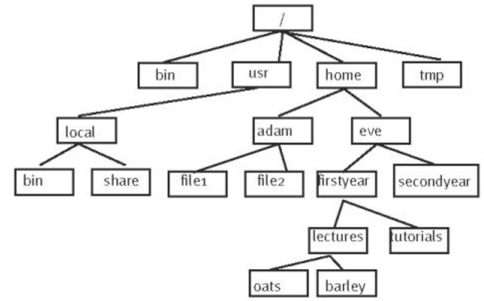
consider the trace:

open close open open



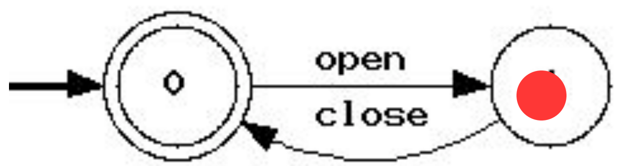
?

file example revisited



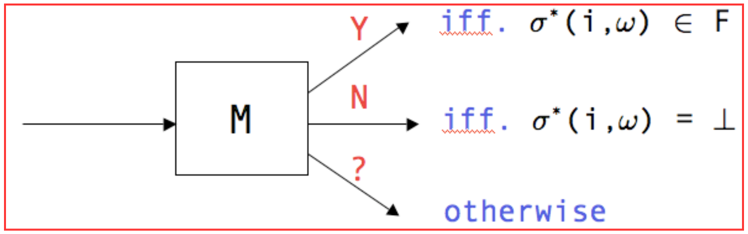
(open close)*

monitored as safety property ⇒ **error**



consider the trace:

open close open open



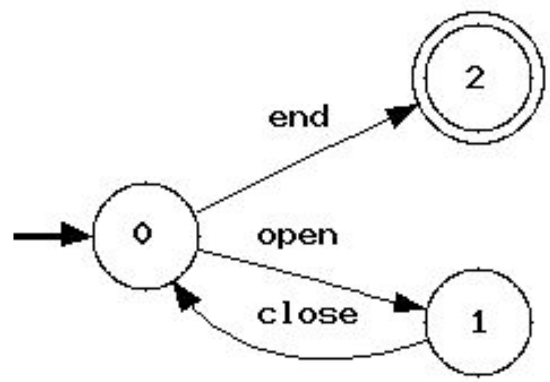
N

R1: A file should eventually be closed once opened.

can write a response property?

introduce a new end event
that **we know** will get emitted

(open close)* end



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.

java.util

Interface Iterator<E>

All Known Subinterfaces:
ListIterator<E>

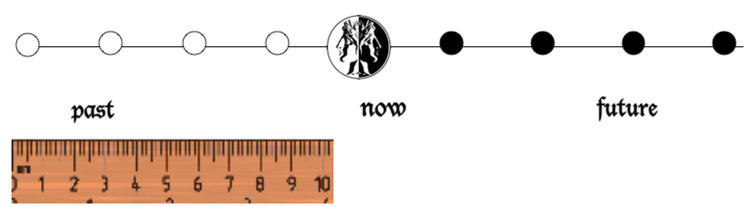
All Known Implementing Classes:
BeanContextSupport.BCSIterator, Scanner

R_1: There should be no two calls to next() without a call to hasNext() in between, on the same iterator .

(hasNext next)*

reg exp is too strong
should allow repeated hasNext calls

i) total trace semantics
ii) looking for violation



java.util
Interface Iterator<E>

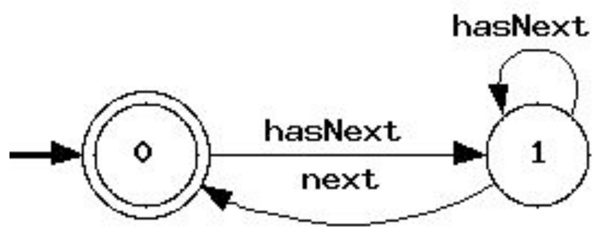
All Known Subinterfaces:
ListIterator<E>

All Known Implementing Classes:
BeanContextSupport.BCSIterator, Scanner

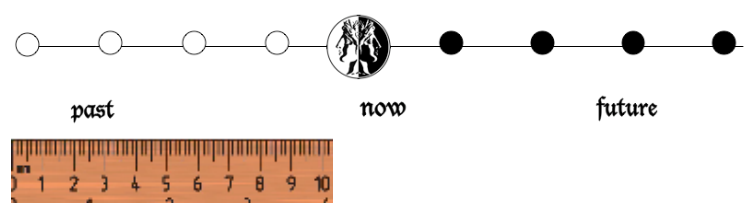
R_1: There should be no two calls to next() without a call to hasNext() in between, on the same iterator.

(hasNext hasNext* next)*

reg exp still too strong
should allow optional initial next call



i) total trace semantics
ii) looking for violation



java.util

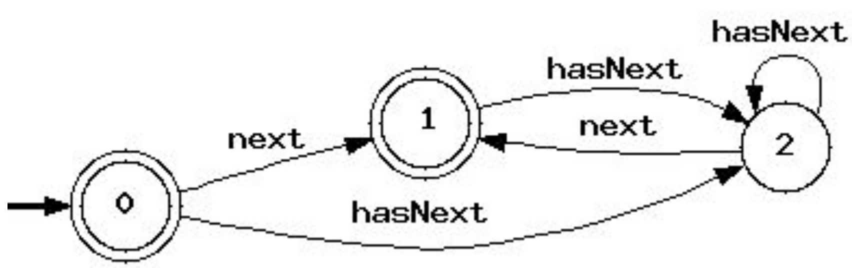
Interface Iterator<E>

All Known Subinterfaces:
ListIterator<E>

All Known Implementing Classes:
BeanContextSupport.BCSIterator, Scanner

R_1: There should be no two calls to next() without a call to hasNext() in between, on the same iterator .

(next + epsilon)(hasNext hasNext* next)*

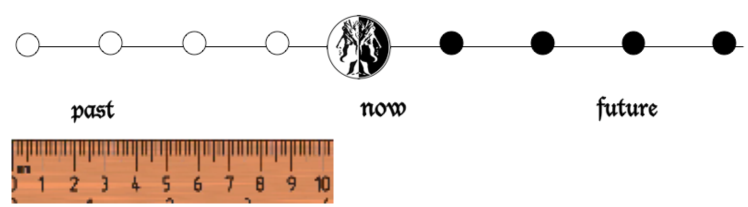


reg exp is too strong

should allow next not to be called

i) total trace semantics

ii) looking for violation



java.util

Interface Iterator<E>

All Known Subinterfaces:
ListIterator<E>

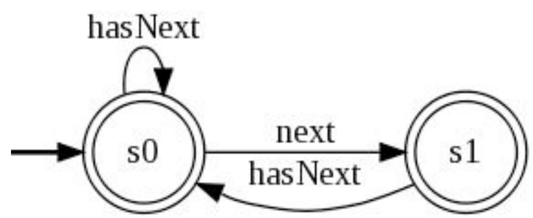
All Known Implementing Classes:
BeanContextSupport.BCSIterator, Scanner

R_1: There should be no two calls to next() without a call to hasNext() in between, on the same iterator.

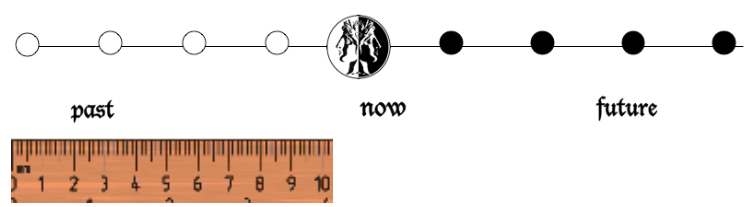
complex!?

(next + epsilon)(hasNext hasNext* (next+epsilon))*

ok



i) total trace semantics
ii) looking for violation



java.util

Interface Iterator<E>

All Known Subinterfaces:
ListIterator<E>

All Known Implementing Classes:
BeanContextSupport.BCSIterator, Scanner

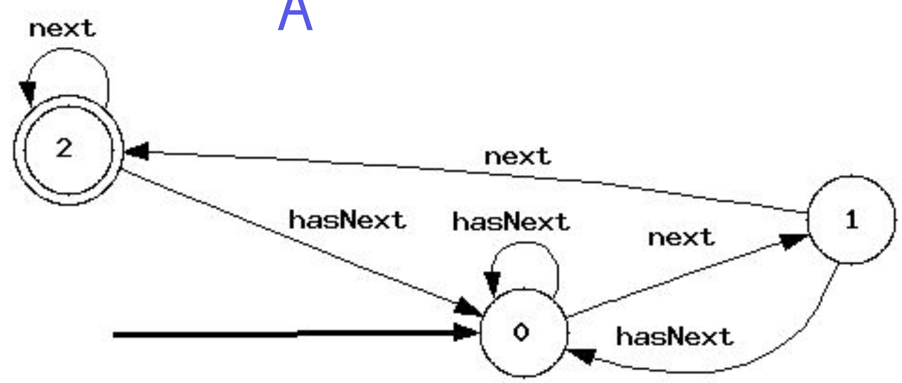
R_1: There should be no two calls to next() without a call to hasNext() in between, on the same iterator.

can we simplify further!?

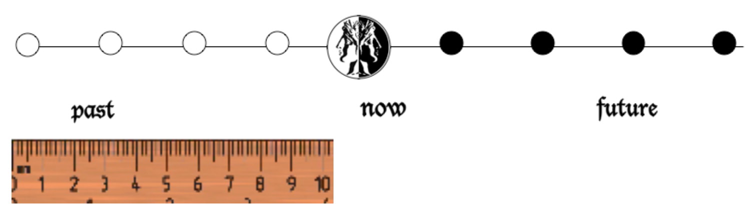
~empty next next

ok

A^*



i) total trace semantics
ii) looking for validation



java.util

Interface Iterator<E>

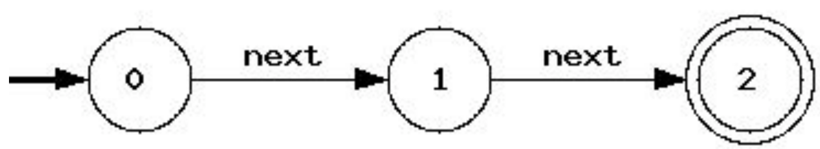
All Known Subinterfaces:
ListIterator<E>

All Known Implementing Classes:
BeanContextSupport.BCSIterator, Scanner

R_1: There should be no two calls to next() without a call to hasNext() in between, on the same iterator .

next next

final

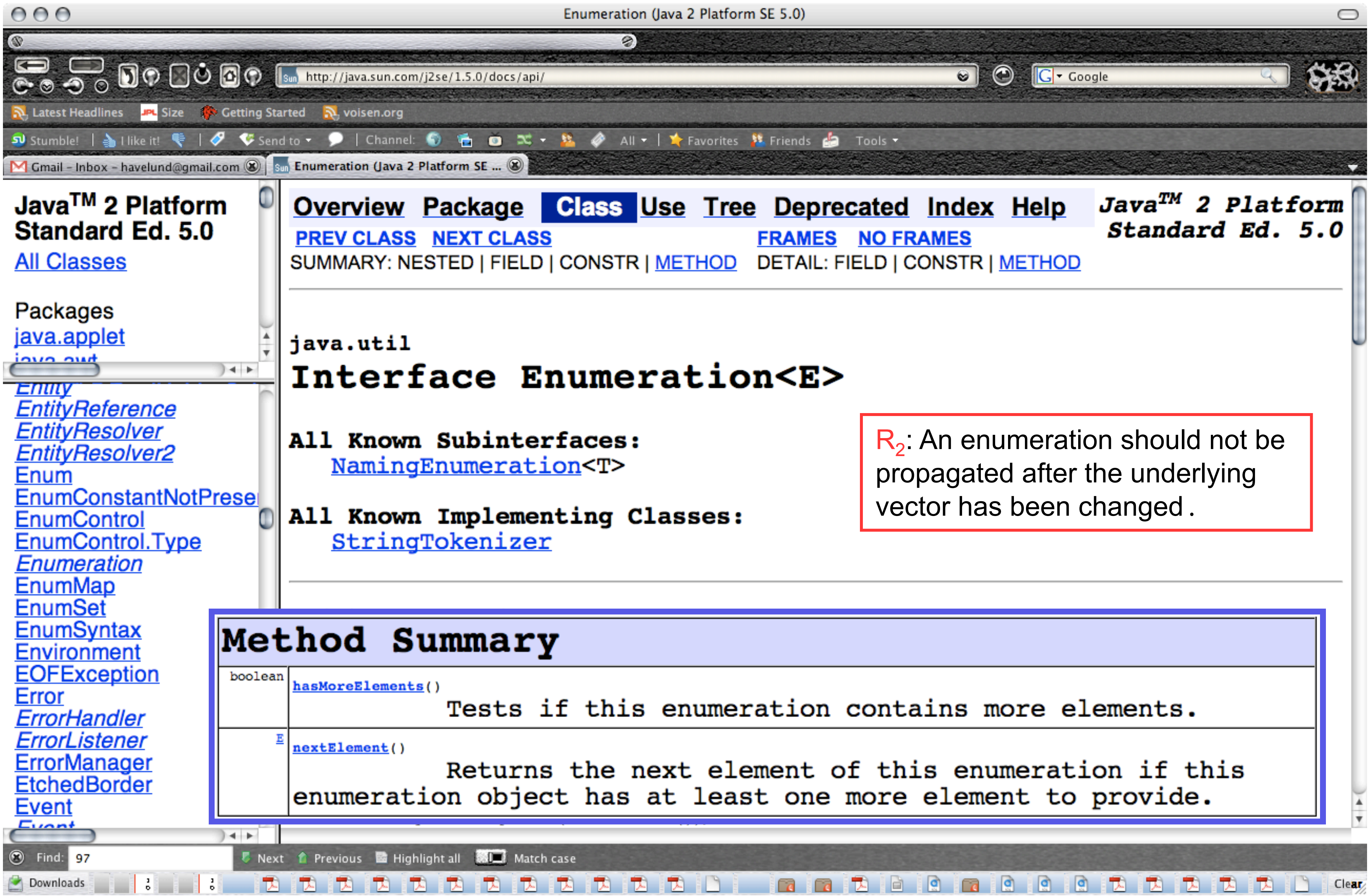


i) suffix trace semantics
ii) looking for validation

next examples

- suffix trace semantics
- validation (meaning an error)
- these interpretations appear the most convenient for regular expressions
- they yield the most succinct specifications
- most (if not all) existing systems for regular expressions support this interpretation
- this does not apply necessarily to other logic systems, such as for example temporal logic.

properties of Java library APIs



java.util

Interface Enumeration<E>

All Known Subinterfaces:
NamingEnumeration<T>

All Known Implementing Classes:
StringTokenizer

R_2: An enumeration should not be propagated after the underlying vector has been changed .

create next* updatesource updatesource* next

suffix validation

properties of Java library APIs

R_3: A collection should not be modified while it is a member of a hashset (don't change the hashcode).

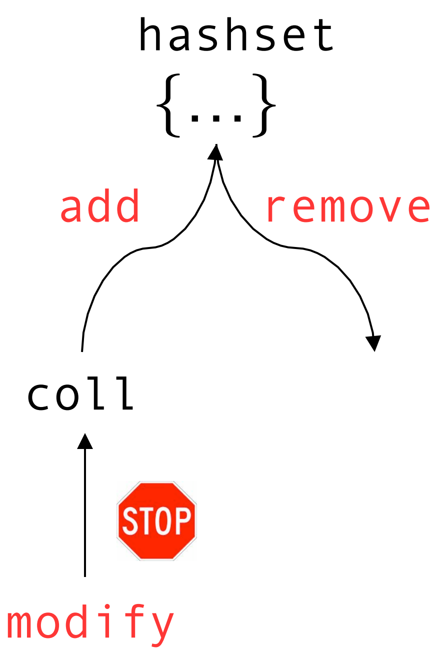
java.util
Class HashSet<E>

of

java.lang.Object
└ java.util.AbstractCollection<E>
  └ java.util.AbstractSet<E>
    └ java.util.HashSet<E>

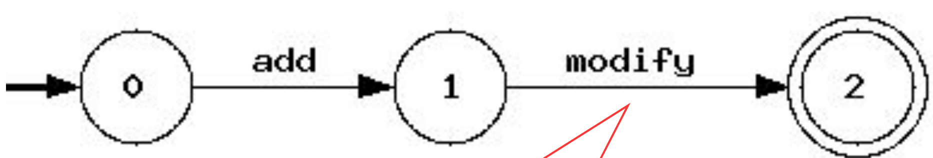
java.util
Interface Collection<E>

R_3: A collection should not be modified while it is a member of a hashset (don't change the hashcode).



add	collection to hashset
remove	collection from hashset
modify	collection

add modify



note: without having seen a remove

suffix validation

limitations of regular expressions for specification

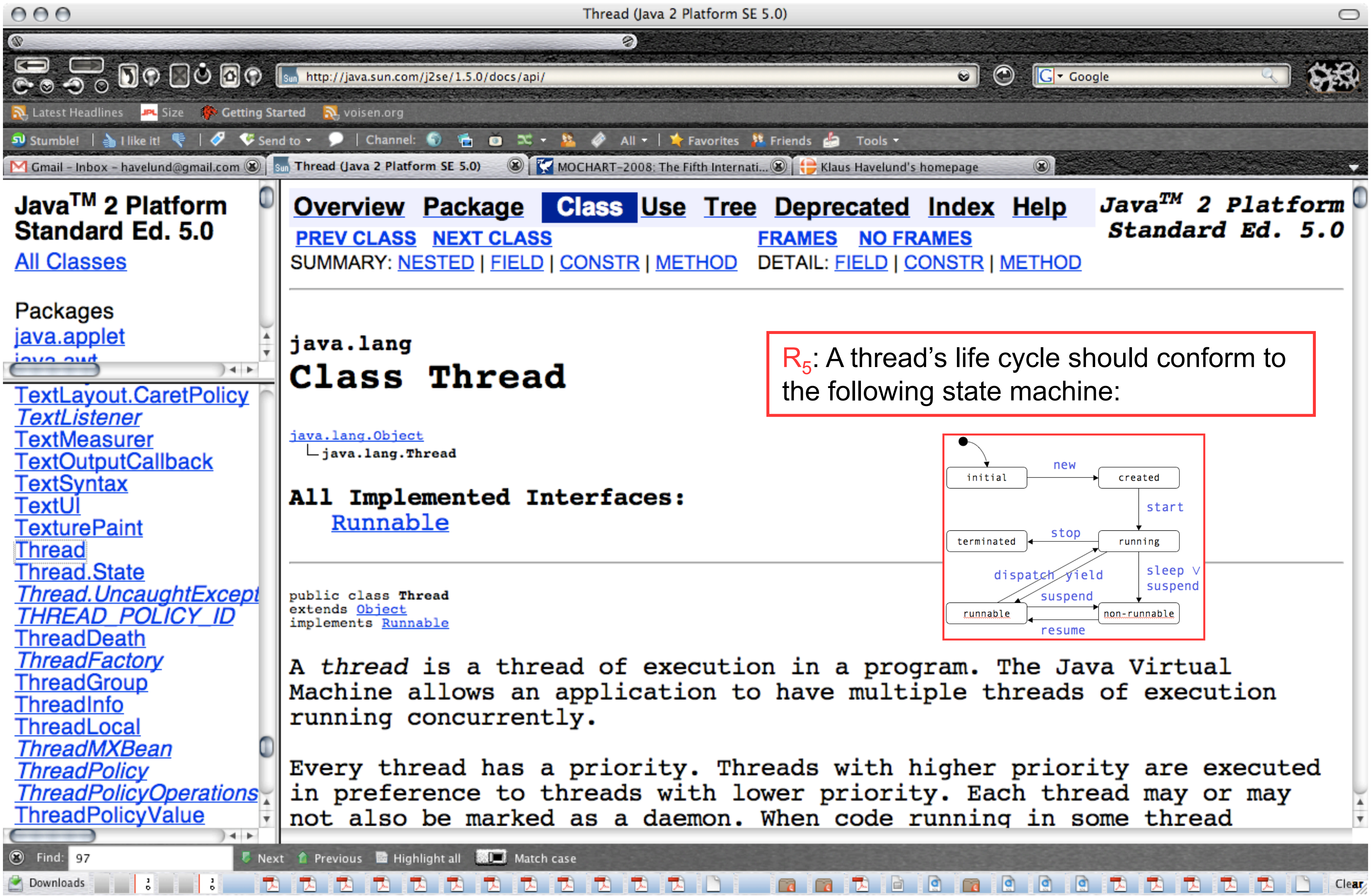
- regular expressions convenient for “brief” properties.
- less convenient on very state-full problems, where all good or bad behaviors must be formulated (state machines, see next slides).
- can only express regular properties. They cannot count an apriori unknown number of times, as required for the following property:

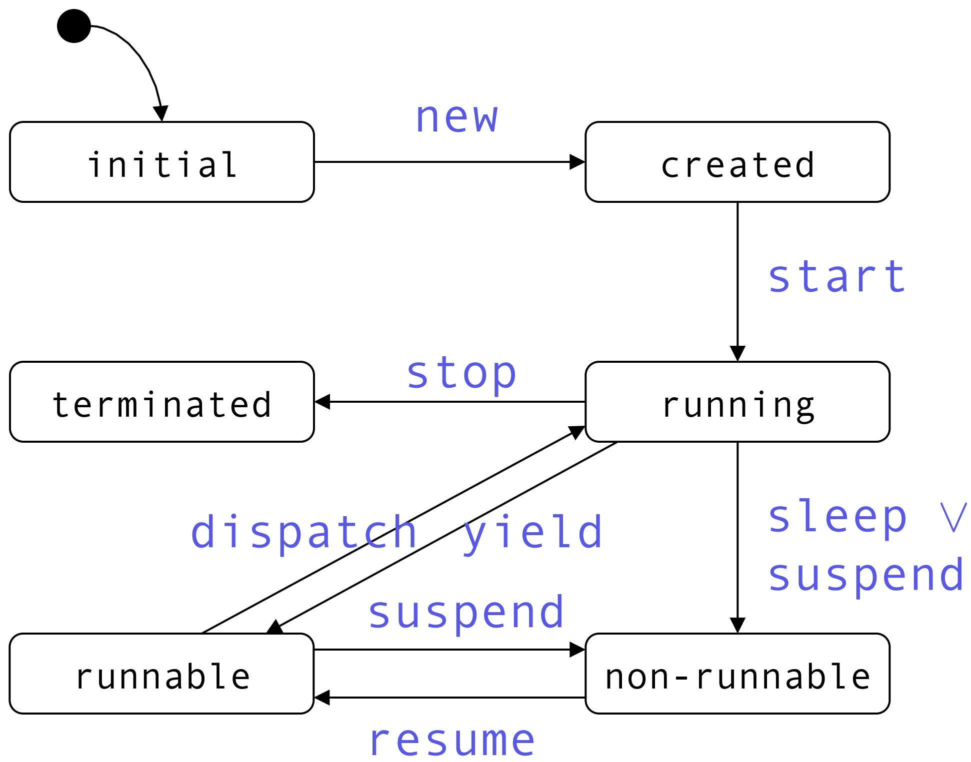
R_4: locks can be taken in a nested manner, but should be released in reverse order.

$\texttt{lock}^n\ \texttt{release}^n$

```
lock
  lock
    lock
    release
  release
release
```

properties of Java library APIs

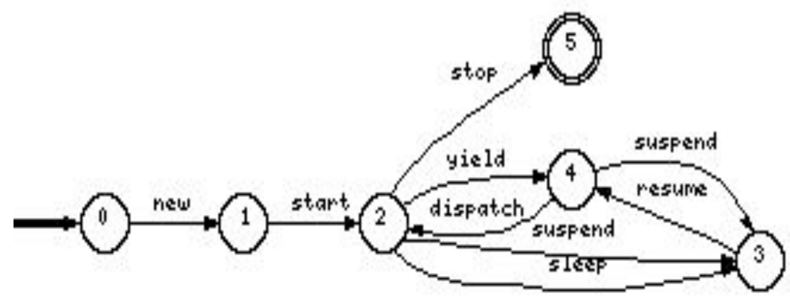




as regular exp

RE not suited for complex state

example: thread's life cycle



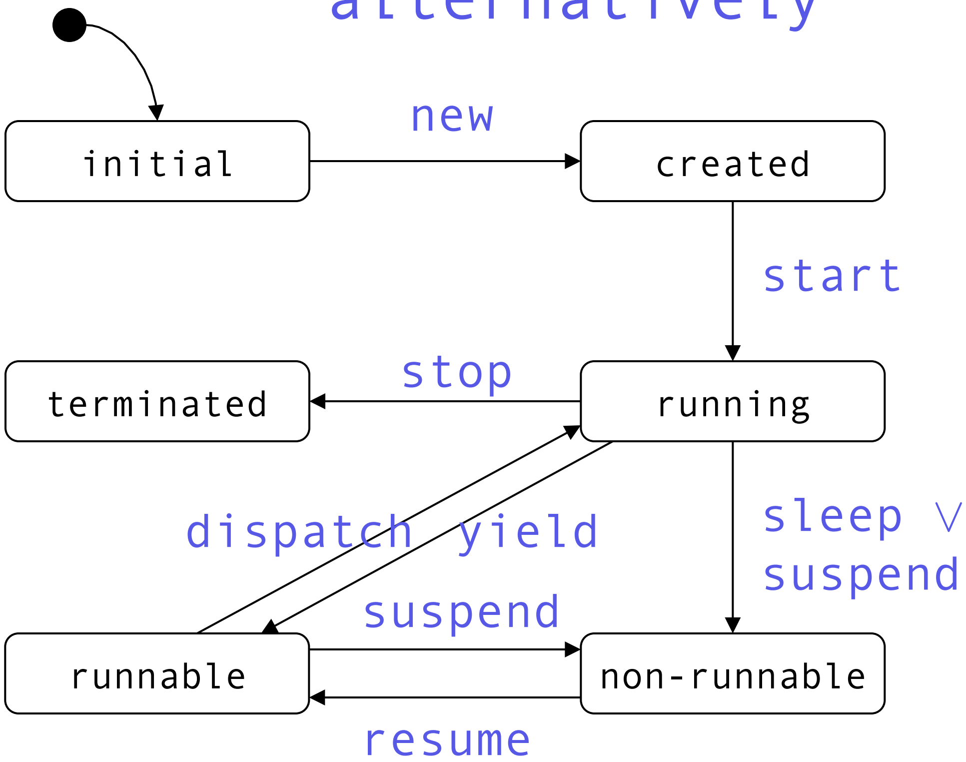
new start
((
(yield (suspend resume)* dispatch) +
((sleep+suspend) (resume suspend)* resume dispatch)
))*
stop

total trace
track violation

tool

what if one can stop in every state?

alternatively



RE not suited for complex state

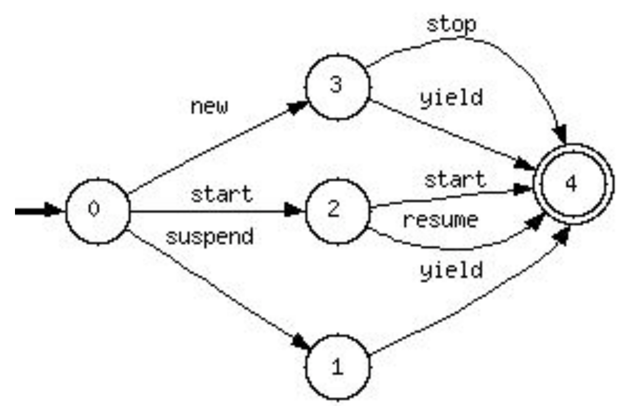
example: thread's life cycle

as regular exp

(new stop) +
(new yield) +
(start start) +
(start resume) +
(suspend yield)
+ …

suffix trace
track validation

bad scenarios



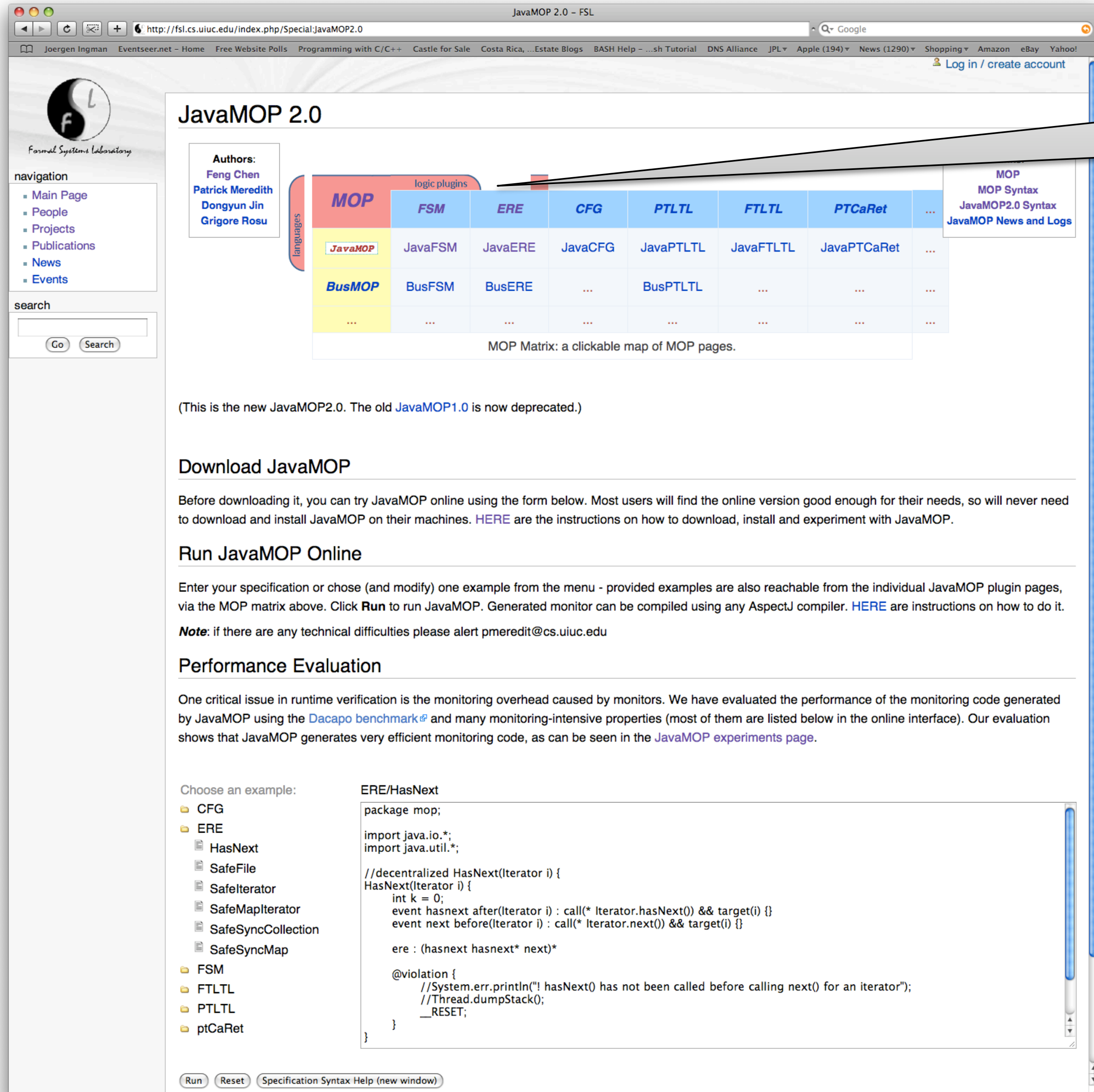
next time

- state machines and regular expressions in JavaMOP, a framework for writing logic-based monitors:

  http://fsl.cs.uiuc.edu/index.php/Special:JavaMOP2.0

- data parameterization:

  $\forall$ (f:File) • (open(f) close(f))*



try writing
FSMs & EREs

MOP Extended Regular Expression (ERE) Plugin – FSL

http://fsl.cs.uiuc.edu/index.php/Special:EREPlugin

This page allows one to synthesize online monitors from extended regular expressions (ERE), using the ERE plugin for MOP available for download below. The generated monitors are automata-based; see links in the top-right box for details and syntax. These monitors for ERE specifications are language-independent and can be used in various language instances of MOP, as well as in other monitoring applications not necessarily based on MOP. Simply chose one of the examples below from the list on the left, or write your own in the text box, then click run.

Download: MOP_ERE_Plugin.zip

Note: if there are any technical difficulties please alert pmeredit@cs.uiuc.edu

Choose an example:

- ComplexPattern2
- FileOperations
- HasNext
- HashSet
- InterruptFix
- SafeConversionSpeed
- SafeCounterModify
- SafeFile
- SafeIterator
- SafeMapIterator
- SafeSyncCollection
- SafeSyncMap

HasNext

```
event open
event close
ere: (open close)*
```

Run Reset

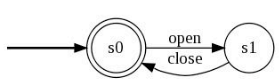
Number of generated monitors (since 14 December 2008): 792 (MOP), 164 (ERE), 48 (Language-neutral ERE)

The textbox below shows the generated monitor; see ERE Plugin Output Syntax (opens new window) for its precise syntax and meaning.
Below the textbox is a more easily read graph representation of it.

```
fsm:
 !s0[
  open -> s1
 ]
 s1[
  close -> s0
 ]
```

Below is an image representation of the monitor:

(open close)*



end