monitoring with state machines & regular expressions

CS 119

the propositional case
recall this property

$R_2$: An enumeration should not be propagated after the underlying vector has been changed.
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
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);
}
```
... 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 \rightarrow \text{state}_1

ds \downarrow
enum \rightarrow \text{state}_1

---

update create

ds \rightarrow \text{state}_1

ds \downarrow
enum \rightarrow \text{state}_1

---

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);
}
```
... 
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);
}

\[ ds \rightarrow state_2 \]
\[ \text{enum} \rightarrow state_1 \]

update  create  update  next
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());
    }
}
state machines
JavaMOP
this course

RuleR
this course

systems

commercial

First in Temporal Solutions

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.

The StateRover graphical programming environment supports statecharts, flowcharts, and advanced software design constructs for representing, building, and verifying complex software modules and classes.
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, `abc` matches `"abc"`, etc., but `[a-c]` matches only `"a"`, `"b"`, 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: `[abc\-z]` matches `"a"`, `"b"`, `"c"`, `"x"`, `"y"`, and `"z"`, as does `[a-c\-x\-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 `[\-abc]`.

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.

Matches what the \n 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, `abc*` matches `"abc"`, `"abc"`, `"abc"`, etc. `[xyz]` matches `"x"`, `"y"`, `"z"`, `"zx"`, `"zyx"`, `"xyzy"`, and so on. `\([ab]\)` matches `"a"`, `"b"`, `"abab"`, `"abab"`, and so on.

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

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

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

this course

extending
AspectJ

public aspect FaultSafeEnum {

pointcut vector_update() :
call(* Vector.add(..)) ||
call(* Vector.remove(..)) ||
call(* Vector.insertElementAt(..)) ||
call(* Vector.removeAt(..)) ||
call(* Vector.removeLast(..)) ||
call(* Vector.set(..));

tracematch(Vector ds, Enumeration e) {

create_enum after returning(e) : (call(* Vector.elements()) && target(ds)) || (call(Enumeration.new()) && args(ds));

call_next before : call(Object Enumeration.nextElement()) && target(e);

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!");
}
}
}
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:

\[(\text{open } (\text{read}^* + \text{write}^*) \text{ close})^*\]
state machine

```plaintext
define fsm:
  s0[open -> s1]
  S1[read -> s3
       write -> s2
       close -> s0]
  S2[write -> s2
       close -> s0]
  s3[read -> s3
       close -> s0]
```
Property:
A property $P$ over $A$ is a language: $P \subseteq A^*$

A monitor:
$\omega = \alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n$

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

but languages are usually infinite
from regular expressions to monitors via state machines

regular expression (RE)

→ step 1

deterministic finite state automaton (DFA)

→ step 2

monitor (M)
Step 2: DFA → 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
\( \sigma \): partial transition function
Step 2 : DFA → M

Given an automaton (DFA):

\[ \mathcal{A} = \{ Q, i \in Q, F \subseteq Q, \sigma: Q \times A \to Q \} \]

Extend \( \sigma \) to \( A^* \) as follows:

\[ \sigma^*: Q \times A^* \to Q \]

\[ \sigma^*(q, \epsilon) = q \]
\[ \sigma^*(q, \alpha) = \sigma(\alpha) \]
\[ \sigma^*(q, \omega\alpha) = \]
\[ \text{if } \sigma^*(q, \omega) \neq \bot \text{ then } \sigma(\sigma^*(q, \omega), \alpha) \text{ else } \bot \]
Step 2: DFA $\rightarrow$ M

Given an automaton (DFA):

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

Define monitor $M_A$:

$\omega = \alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n$

- $Y$ if $\sigma^*(i, \omega) \in F$
- $N$ if $\sigma^*(i, \omega) = \bot$
- $?$ otherwise
**Theorem**

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

\[ \varphi = (\text{green yellow red})^* \quad \overset{A(\varphi)}{\longrightarrow} \quad L_\varphi = \{\varepsilon, \text{green yellow red,}\ldots\} \]
syntax for extended regular expressions

\[ E ::= \emptyset \mid \epsilon \mid A \mid E \cdot E \mid E^* \mid E+E \mid E&E \mid \neg E \]

examples  
\[ A = \{a,b,c\} : \text{extended with negation} \]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>aab</td>
<td>{aab}</td>
</tr>
<tr>
<td>(ab)*</td>
<td>{\epsilon, ab, ababab, \ldots}</td>
</tr>
<tr>
<td>(a+b)* &amp; \neg(ab)*</td>
<td>words of randomly interleaved a’s and b’s but not any words of form: ababab \ldots</td>
</tr>
<tr>
<td></td>
<td>{a, aa, abba, bbbb, \ldots}</td>
</tr>
<tr>
<td>Expression</td>
<td>Semantics</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
</tr>
<tr>
<td>$L(\emptyset)$</td>
<td>${}$</td>
</tr>
<tr>
<td>$L(\epsilon)$</td>
<td>${\epsilon}$</td>
</tr>
<tr>
<td>$L(\alpha)$</td>
<td>${\alpha}$</td>
</tr>
<tr>
<td>$L(E_1 \cdot E_2)$</td>
<td>${\omega_1 \omega_2 \mid \omega_1 \in L(E_1) \land \omega_2 \in L(E_2)}$</td>
</tr>
<tr>
<td>$L(E^*)$</td>
<td>${\omega_1 \omega_2 \ldots \omega_i \ldots \omega_n \mid \omega_i \in L(E)}$</td>
</tr>
<tr>
<td>$L(E_1 + E_2)$</td>
<td>$L(E_1) \cup L(E_2)$</td>
</tr>
<tr>
<td>$L(E_1 &amp; E_2)$</td>
<td>$L(E_1) \cap L(E_2)$</td>
</tr>
<tr>
<td>$L(\neg E)$</td>
<td>$A^* \setminus L(E)$</td>
</tr>
</tbody>
</table>
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 $\varepsilon$-automata ($\varepsilon$-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: \( \text{RE} \rightarrow \text{DFA} \)

DFA → minimized DFA

**minimize:**
delete nodes that do not lead to final node

Start
RE $\rightarrow \epsilon$-NFA

\[ r_1 + r_2 \]

\[ r_1 r_2 \]

\[ r^* \]
example derivation

RE → $\epsilon$-NFA

(open close)* + stop
example derivation

$\epsilon$-NFA $\rightarrow$ 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

\[ \text{iff. } \sigma^*(i, \omega) \in F \]
\[ \text{iff. } \sigma^*(i, \omega) = \bot \]
otherwise
file example revisited

(open close)*

consider the trace:
open close open
file example revisited

(open close)* monitored as safety property ⇒ no error

consider the trace:
open close open

\[ \text{iff. } \sigma^*(i, \omega) \in F \]
\[ \text{iff. } \sigma^*(i, \omega) = \perp \]
otherwise
file example revisited

(open close)∗

now, assume extra event

consider the trace:
open close open open open
file example revisited

(\text{open close})^* \quad \text{monitored as safety property} \Rightarrow \text{error}

consider the trace:
open close open open open
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

R1: 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<>  

All Known Implementing Classes:

- beanContextSupport.ResultSetIterator, Scanner

---

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

\[(\texttt{hasNext next})^*\]

reg exp is too strong should allow repeated \texttt{hasNext} calls

---

i) total trace semantics

ii) looking for violation
java.util

**Interface Iterator<E>**

All Known Subinterfaces:
- `ListIterator<>`

All Known Implementing Classes:
- `BeanContextSupport.MSIterator`, `Scanner`

---

R₁: There should be no two calls to `next()` without a call to `hasNext()` in between, on the same iterator.

---

(reg exp still too strong should allow optional initial next call)

---

(hasNext hasNext* next)*

---

i) total trace semantics

ii) looking for violation
i) total trace semantics
ii) looking for violation

\[(next + \text{epsilon})(\text{hasNext hasNext}^* \text{ next})^*\]

\(R_i: \text{ There should be no two calls to } \text{next()} \text{ without a call to } \text{hasNext()} \text{ in between, on the same iterator.}\)

reg exp is too strong should allow next not to be called
java.util

`iterator Iterator<E>`

All Known Subinterfaces:

- `ListIterator<E>`

All Known Implementing Classes:

- `BeanContextSupport.ISIterator`, `Scanner`

---

**R₁:** 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))`
```

---

i) **total trace semantics**

ii) **looking for violation**
i) total trace semantics
ii) looking for validation
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

R$_2$: An enumeration should not be propagated after the underlying vector has been changed.
java.util

**Interface Enumeration**<E>

All Known Subinterfaces:
  - NamingEnumeration<T>

All Known Implementing Classes:
  - StringTokenizer

R₂: An enumeration should not be propagated after the underlying vector has been changed.

create next* updatesource updatesource* next
properties of Java library APIs

R₃: A collection should not be modified while it is a member of a hashset (don’t change the hashcode).
**R₃**: 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

**note**: without having seen a remove
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:

\[ R_4: \text{locks can be taken in a nested manner, but should be released in reverse order.} \]

\[
\text{lock}^n \text{ release}^n
\]
properties of Java library APIs

The thread's life cycle should conform to the following state machine:

A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread
RE not suited for complex state

example: thread's life cycle

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

stop
RE not suited for complex state

example: thread's life cycle

alternatively

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

as regular exp

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:

\[ \forall (f:\text{File}) \bullet (\text{open}(f) \; \text{close}(f))^{*} \]
try writing FSMs & EREs
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 choose 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 pmenard@cs.uiuc.edu

Choose an example:
- ComplexPattern2
- FileOperations
- HasNext
- HashSet
- InterruptFix
- SafeConversionSpeed
- SafeCounterModify
- SafeFile
- SafeIterator
- SafeMapIterator
- SafeSyncCollection
- SafeSyncMap

HasNext

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.

```plaintext
fn:
  s0[
    open -> s1
  ]
  s1[
    close -> s0
  ]
```

Below is an image representation of the monitor:

```
(\text{open close})^*
```

- s0
- s1

open

close


end