Aspects for Trace Monitoring

The *abc* team:
http://aspectbench.org

Concurrent modification & enumeration

Enumeration: old version of Iterator, no guarantee of safety

THREAD 1:	THREAD 2:
Vector v;	
…	
Enumeration e = new MyEnum(v)	
…	
Elt a = (Elt) e.nextElement();	…
….	v.remove(b)
a = (Elt) e.nextElement();	…

undefined behaviour!

Catching the problem at runtime

add *stamp* field to:
- Vector
- Enumerations

add *source* field to:
- Enumerations

upon creation of an enumeration e over a vector v:

```
e.stamp = v.stamp; e.source = v;
```

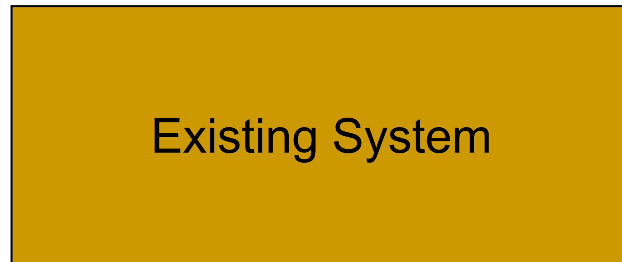
upon modification of a vector v:

```
v.stamp++;
```

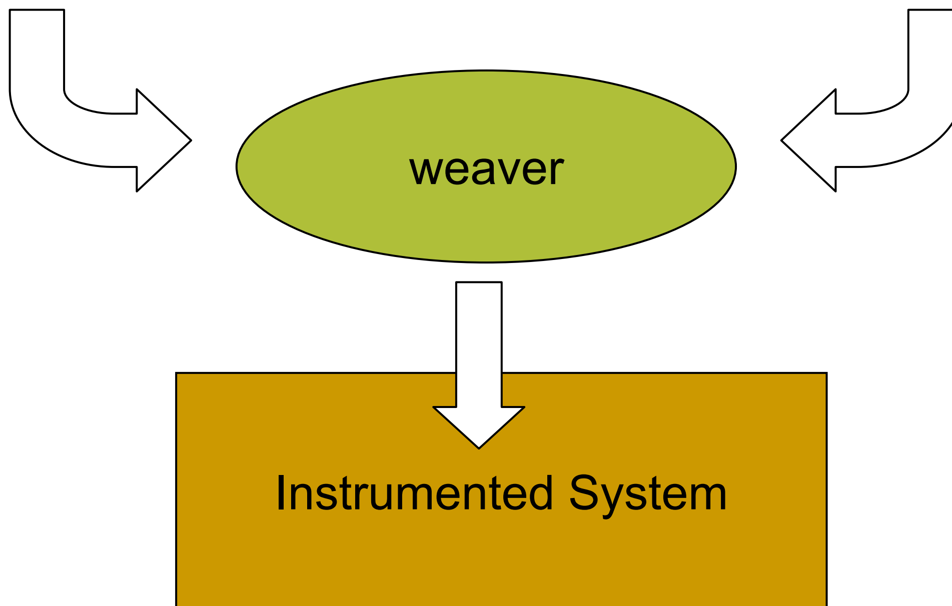
upon each nextElement on e:

```
if (e.source != null   &&   e.stamp != e.source.stamp)
        throw new ConcurrentModificationException();
```

Aspect-oriented Programming



Aspect
- injects members into classes
- intercepts events



AspectJ: extension of Java

Aspect for Safe Enum: new members

```
public aspect SafeEnum {

    private long Vector.stamp = 0;

  // introduce new members on every implementation of Enumeration
    private long Enumeration.stamp;
    private void Enumeration.setStamp(long n) { stamp = n; }
    private long Enumeration.getStamp() { return stamp; }

    private Vector Enumeration.source;
    private void Enumeration.setSource(Vector v) { vector = v;}
    private Vector Enumeration.getSource() {return vector;}

   // …. intercept creation, update and nextElement …
}
```

Aspect for Safe Enum: interceptions

```
synchronized after(Vector ds) returning (Enumeration e) :
   call(Enumeration+.new(..)) && args(ds) {
         e.setStamp(ds.stamp);
         e.setSource(ds);
}

synchronized after(Vector ds) :
   vector_update() && target (ds) {
         ds.stamp++;
}

synchronized before(Enumeration e) :
   call(Object Enumeration.nextElement()) && target(e) {
      if  (e.getSource() != null && e.getStamp() != e.getSource().stamp)
         throw new ConcurrentModificationException();
}
```

Aspect for Safe Enum: updates

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

Advantages of Aspects

- Easy, flexible instrumentation
- Benefits of high-level programming language
- Very small overheads (in this example)
- Widely available implementations, textbooks

Disadvantages of Aspects

- "That's not a *specification* of safe enumeration!"
- Syntactic nature of pointcuts
- No crisp semantics
- Loss of modular reasoning
- (I lied a little about weaving library classes)

this talk

Tracematches

match regular patterns

on sequences of before/after events

Autosave

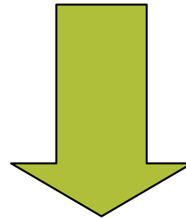
```
tracematch() {
        sym save after:
                call ( * Application.save() )
              || call ( * Application.autosave() );
        sym action after:
              call ( * Command.execute() );

        action [5]

        { Application.autosave(); }
}
```

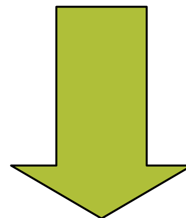
Filtering traces (no variables)

sequence of before/after events



remove all events that do not match a symbol, except last one

sequence of before/after events



match filtered trace with regular expression

if successful, run extra code

Failsafe enumeration over vectors

```
public aspect FailSafeEnum {

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

   tracematch(Vector ds, Enumeration e) {

      sym create_enum after returning(e) : call(Enumeration+.new(..)) && args(ds);
      sym call_next before : call(Object Enumeration.nextElement()) && target(e);
      sym update_source after : vector_update() && target(ds);

      create_enum   call_next*   update_source+   call_next

      { throw new ConcurrentModificationException(); }

   }
}
```

Filtering traces (with variables)

sequence of before/after events

filter for *every*
instantiation
of the variables

match filtered trace
with instance of
regular expression

if successful, run extra code

match filtered trace
with instance of
regular expression

if successful, run extra code

Sketch of operational semantics

- Run state machine alongside program
- States are tagged with constraints:
  - equality: ($v = o$)
  - inequality: if you decide to “skip” an event that might match with binding (v=o), henceforth you must assume ($v \neq o$)

Two BIG efficiency problems

- space leaks:
  - object bindings
  - partial matches

solved with static analysis of pattern

- indexing:
  - given event, find partial matches

solved with subtle data structure

The key to leak detection and prevention

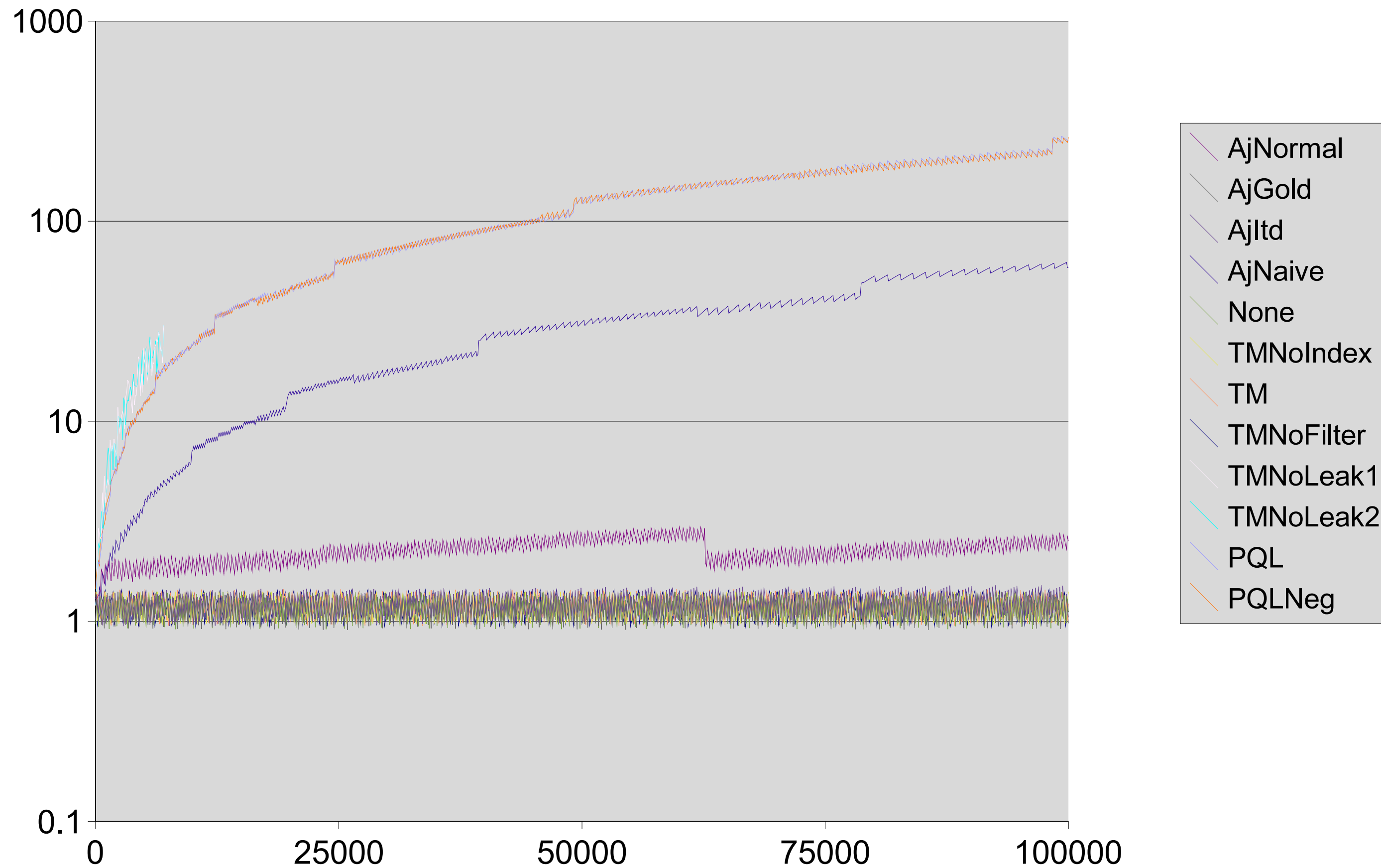
Classifying variable *v* on a state *s:*

Collectable:
all paths in automaton from *s* to a final state contain a transition that binds *v*
use weak references for bindings of v; when nullified, can also discard all disjuncts that contain these

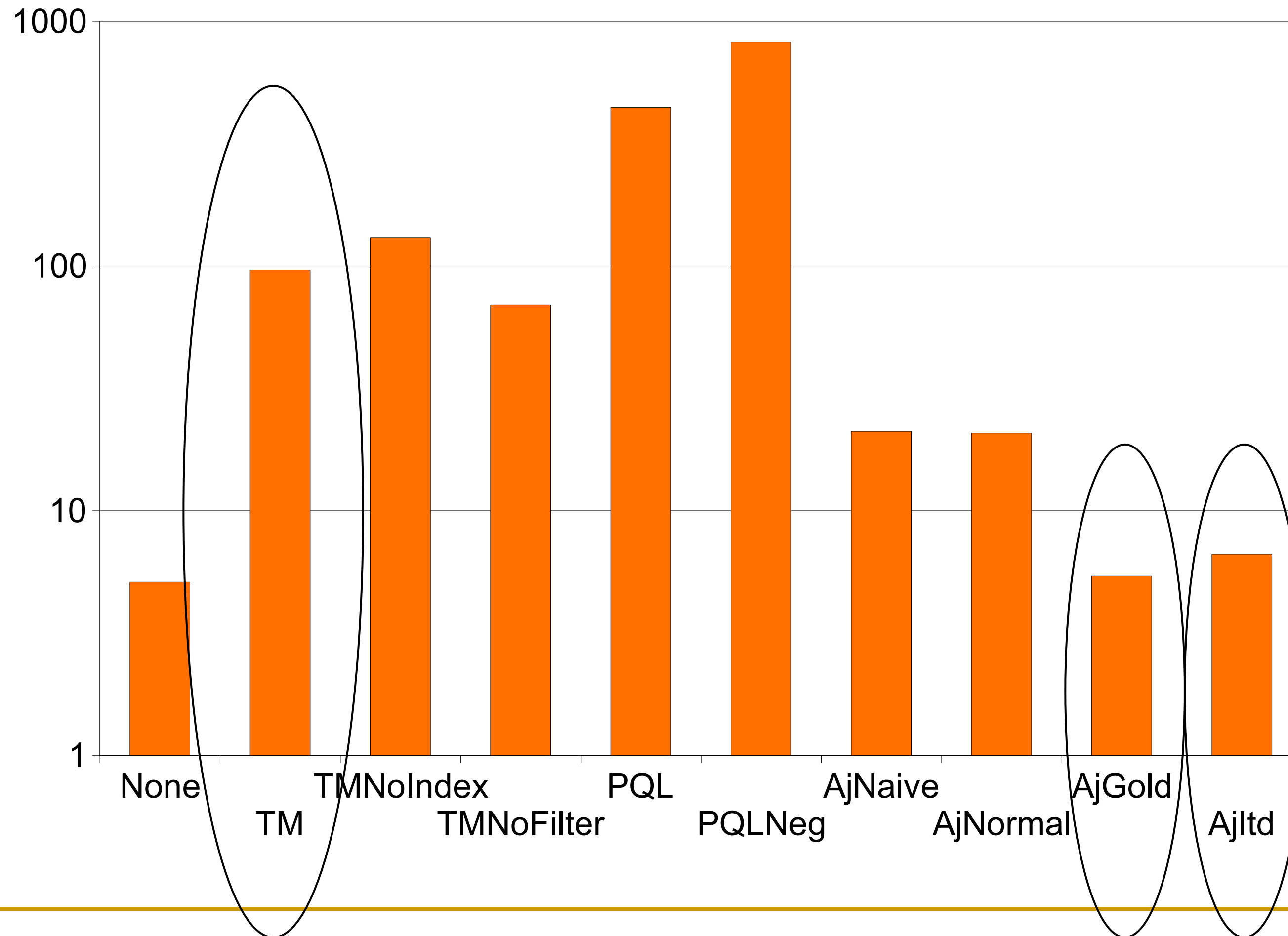
Weak:
Not collectable, but advice body does not mention *v*
use weak references for bindings of v; cannot discard containing disjuncts upon nullification

Strong:
Not collectable and not weak
use strong references for bindings of v

Performance: JHotDraw - memory



Performance: JHotDraw - time



More benchmarks

monitor	base	ksloc	none	aspectj	leak	noidx	TM
nulltrack	certrevsim	1.4	0.2	0.5	1.6	25.6	1.6
hashcode	aprove	438.7	345.0	458.9	>90m	>90m	845.1
observer	ajhotdraw	9.9	2.7	2.9	4.1	15.8	4.1
dbpooling	artificial	<0.1	70.0	4.5	5.0	4.8	4.9
luinmeth	jigsaw	100.9	13.6	18.0	21.9	20.9	22.4
lor	jigsaw	100.9	13.6	19.9	34.9	34.7	34.9
reweave	abc	51.2	4.5	5.4	9.1	9.0	8.7

Tracematch summary

- Surprisingly tricky to get right: proof of

  declarative semantics = operational semantics

  in OOPSLA ’05

- Part of extensible compiler abc

- Efficiency through combination of
  - leak prevention
  - indexing partial solutions

  in techreport abc-2006-1

Datalog Pointcuts

Specify semantic properties
instead of mere syntactic patterns

Syntax versus semantics

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

A call to any method of Vector
that may write to a memory location
that may be read
in some implementation of Enumeration.nextElement().

Program is a relational database

typeDecl (RefType T, String N, Boolean IsIntf, Package P)
    T has name N in package P, IsIntf indicates T interface or not
implements (Class C, Interface I)
    C implements interface I
methodDecl (Method M, String N, RefType C, Type T)
    M has name N, is declared in C, and has return type T
callShadow (Shadow S, Method M, RefType T)
    S is a call to M with static receiver type T
executionShadow (Shadow S, Method M)
    S is the body of M
getShadow (Shadow S, Field F, RefType T)
    S is get of F with static receiver type T
contains (Element P, Element C)
    C is lexically contained in P
…

Datalog for method pattern

```
vector_update_method(Method M) :-
        // M is a method in a class V named ‘Vector’
        typeDecl(V,’Vector’,_),
        methodDecl(M,_,V,_),

        // N is a method named ‘nextElement’ in an
        // implementation I of the Enumeration interface
        typeDecl(E,’Enumeration’,_),
        implements(I,E),
        methodDecl(N,’nextElement’,I,_),

        // N may read field F (possibly via a chain of calls)
        mayRead(N,F),

        // M may write field F
        mayWrite(M,F).
```

Datalog for *mayRead*

```
mayRead(Method M,Field F) :-
        callContains(M,G),
        getShadow(F,G).

callContains(Method M, Shadow G) :-
        mayCall*(M,M'),              // static call chain from M to M'
        executionShadow(E,M'),       // body of M' is E
        contains+(E,G),              // it contains a shadow G

mayCall*(Method X, Method Z) :- X=Z, method(X) .
mayCall*(Method X, Method Z) :- mayCall(X,Y),mayCall*(Y,Z).

contains+(Element X, Element Z) :- contains(X,Z).
contains+(Element X, Element Z) :- contains(X,Y),contains+(Y,Z).
```

Alternative surface syntax

Reduce elegant pattern notation to datalog queries with rewrite rules, *e.g.*

aj2dl(call(methconstrpat), C, S)

→

X^R^(*methconstrpat2dl*(methconstrpat, C, R, X), callShadow(S,X,R))

90 rules suffice to reduce the complete AspectJ pointcut language to Datalog; Implemented exactly as shown above, in concrete syntax, using Stratego

the first rigorous semantics of AspectJ pointcut matching

But is Datalog execution feasible?

- CodeQuest:
  - optimising compiler from Datalog to SQL
  - facts held in relational database
- XSB:
  - optimising native compiler for tabled Prolog
  - facts held in memory

Execution times of > 6 hours

Replace pointcut matcher in *ajc*?

project	***ksloc***	***ajc***	***Populate + index (PI)***	***Aggregate Query (AQ)***	***Ratio = (PI + AQ + AJC) / AJC***
weka	10	5.49	6.29	6.62	3.35
jhotdraw	21	5.05	6.33	10.71	4.37
reweave	51	19.89	11.67	19.95	2.59
jigsaw	101	32.25	15.61	30.66	2.43

all times in seconds

Summing up

aspects: useful for testing and runtime verification

- AspectJ popular extension of Java
- two compilers ajc (industrial use); abc (research)

tracematches: regular patterns over full computation history

- optimising implementation included with abc
- alternative LTL patterns in J-Lo (another abc extension)

datalog pointcuts:

- balance between static and dynamic analysis
- concise pattern syntax via user-specified rewrite rules
- coming up shortly in new abc release

Website with papers and downloads

- http://aspectbench.org
- Source and executables for abc compiler
  - Full AspectJ 1.2 language (1.5 coming soon)
  - Built on Soot and Polyglot
  - Easy to put in new analyses, optimisations, language features
- Mailing lists for users and developers
- Bugzilla for reporting and tracking bugs
- Tutorial slides on how to extend abc:
  - We’re happy to help you get started
  - Links to other groups who are extending abc
- Benchmark set for tracematches
  - More RV benchmarks very, very welcome