Type-State Checking using Dataflow Analysis

CS 6340

Motivation

Phase	Defect Removal Cost Multiplier
Requirements	**1**
Design	**3**
Code,Unit Test	**5**
Function/System Test	**12**
User Acceptance Test	**32**
Production	**95**


Technology Quarterly

SOFTWARE

Building a better bug-trap

Jun 19th 2003

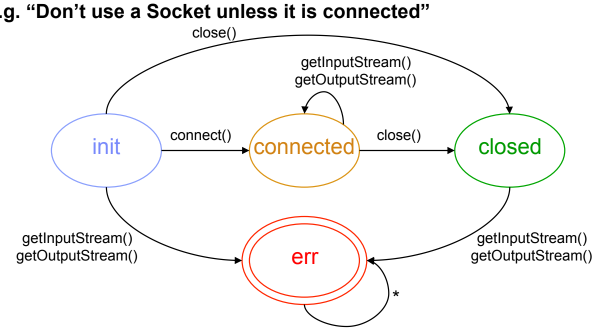
From *The Economist* print edition

People who write it are human first and programmers only second—in short, they make mistakes, lots of them. Can software help them write better software?

Typestate

- Application Trends
  - Increasing number of libraries and APIs
  - Non-trivial restrictions on permitted sequences of operations
- **Typestate:** Temporal safety properties, encoded as DFAs
- Apply to many libraries and APIs

e.g. "Don't use a Socket unless it is connected"



Goal

- **Typestate Verification: statically ensure that no execution of a Java program can transition to err**
  - Sound* (excluding concurrency)
  - Precise enough (reasonable number of false alarms)
  - Scalable
    - Handle programs of realistic size
    - Handle all Java features

* In the real world, some other caveats apply.

Challenge: Aliasing

```
void foo(Socket s, Socket t) {
  s.connect();
  t.getInputStream(); // potential error?
}
```

- **Strong Updates may be required**
  - Rules out solely flow-insensitive analysis

```
void foo(Socket s, Socket t) {
  s.connect();          // s MUST point to connected
  t = s;                // t MUST point to connected
  t.getInputStream();
}
```


Difficulties

```
class SocketHolder {  Socket s;  }
Socket makeSocket() { return new Socket(); // A }
open(Socket l) {
   l.connect();
}
talk(Socket s) {
  s.getOutputStream()).write("hello");
}
dispose(Socket s) { h.s.close(); }
main() {
  Set<SocketHolder> set = new HashSet<SocketHolder>();
  while(…) {
     SocketHolder h = new SocketHolder();
     h.s = makeSocket();
     set.add(h)
  } ;
  for (Iterator<SocketHolder> it = set.iterator(); …) {
     Socket g = it.next().s;
     open(g);
     talk(g);
     dispose(g);
  }
}
```

- **Flow-Sensitivity**
- **Interprocedural flow**
- **Context-Sensitivity**
- **Non-trivial Aliasing**
  - Destructive updates
- Path Sensitivity (ESP)
- Full Java Language
  - Exceptions, Reflection, ….
- Big programs

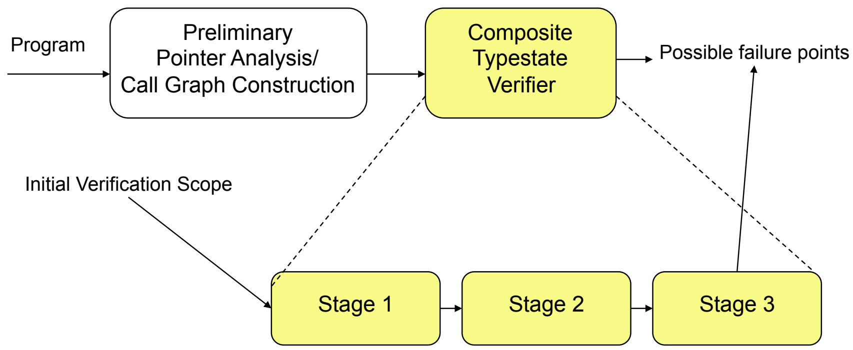
Our Approach

- **Flow-sensitive, context-sensitive abstract interpretation**
  - Abstract domains combine typestate and points-to
  - Techniques for inexpensive strong updates
    - *Uniqueness*
    - *Focus*
- **Staging**
  - Family of abstractions of varying cost/precision
  - Early stages reduce work for latter stages

Why this is cool

- **Nifty abstractions**
  - Combined domain
    - More precise than 2-stage approach
    - Concentrate expensive effort where it matters
  - Parameterized hierarchy of abstractions
  - Relatively inexpensive techniques that allow precise aliasing
    - Much cheaper than shape analysis
    - More precise than usual "scalable" analyses
- **It works pretty well**
  - Techniques are complementary
  - Flow-sensitive functional IPA with sophisticated alias analysis on ~100KLOC in 20 mins.
    - Overapproximate inexpensive facts (distributive)
    - Underapproximate expensive facts (non-distributive)
  - <5% false warnings

Analysis Overview



- Sound, abstract representation of program state
- Flow-sensitive propagation of abstract state
- Context-sensitive: *functional approach to interprocedural analysis* [Sharir-Pneuli 82]
  - Tabulation Solver [Reps-Horwitz-Sagiv 95]
- ***Hierarchy of abstractions***

Base Abstraction

AS := { < Abstract Object, TypeState> }

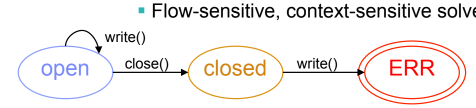
- **Two-Stage Approach**
  - First alias analysis, then typestate analysis
- ***Abstract Object*** := heap partition from preliminary pointer analysis
  - e.g. allocation site
- **Transfer functions**
  - Straightforward from instrumented concrete semantics
  - Rely on preliminary pointer analysis to determine typestate transitions
  - **No Strong Updates**
    - *{< I, T>} → { <I, T>, <I, δ(T)> }*
- **Works sometimes (75%)**

Base Abstraction

It works in some cases:

- Simple abstraction
- Flow-sensitive, context-sensitive solver

"Don't write to a closed PrintWriter"



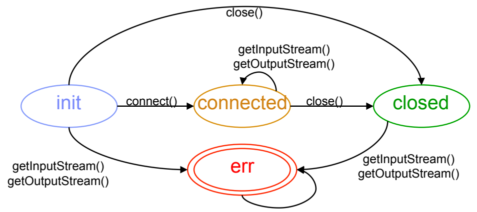

```
writeTo(PrintWriter w) { w.write(…); }
        <P, open>, <Q,open>, <Q, closed>, <Q, ERR>
        <P, open>, <Q,open>, <Q, closed>, <Q, ERR> ✓

main() {
  PrintWriter p = new PrintWriter(…); // P
        <P, open>
  PrintWriter q = new PrintWriter(..); // Q
        <P, open>, <Q,open>
  q.close();
        <P, open>, <Q,open>, <Q, closed>
  writeTo(q);
        <P, open>, <Q,open>, <Q, closed>, <Q, ERR>
  writeTo(p);
        <P, open>, <Q,open>, <Q, closed>, <Q, ERR>
  if (?) {
    p.close();
        <P, open>, <P,closed>, <Q, closed>, <Q,open>, <Q, ERR>
  else {
    writeTo(p);
        <P, open>, <Q,open>, <Q, closed>, <Q, ERR> ✓
    p.close();
        <P, open>, <P, closed>, <Q,open>, <Q, closed>, <Q, ERR>
  }
}
```

Base Abstraction

*Useless for **properties which require strong updates***

"Don't use a Socket unless it is connected"




```
open(Socket s) { s.connect();}
talk(Socket s)
   { s.getOutputStream()).write("hello"); }
dispose(Socket s) { s.close(); }
main() {
  Socket s = new Socket(); //S
                             <S, init>
  open(s);
                             <S, init> , <S, connected>
  talk(s);
                             <S, init> , <S, connected>, <S, err> ×
  dispose(s);
}
```

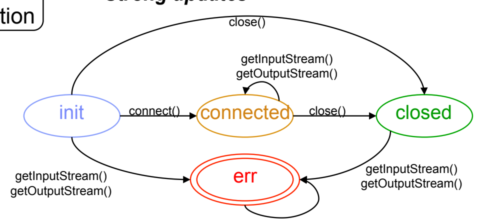

Unique Abstraction

AS := { < Abstract Object, TypeState, Unique> }

- **Transfer functions: Base Abstraction +**
  - Unique := true (U) when *creating* factoid at allocation site
  - Unique := false (¬U) when *propagating* factoid *through* an allocation site
- **Intuition: "Unique" ≈ "∃ exactly one concrete instance of abstract object"**
- **Strong Updates allowed for *e.op()* when**
  - Unique
  - *e* may point to exactly one abstract object
- **Works sometimes (80%)**

Unique Abstraction

Strong updates




```
open(Socket s) { s.connect();}
talk(Socket s) { s.getOutputStream()).write("hello"); }
dispose(Socket s) { s.close(); }
main() {
  Socket s = new Socket(); //S
                             <S, init, U>
  open(s);
                             <S, connected, U>
  talk(s);
                             <S, connected, U> ✓
  dispose(s);
}
```


Unique Abstraction

More than just singletons?

```
open(Socket s) { s.connect();}
talk(Socket s) { s.getOutputStream()).write("hello"); }
dispose(Socket s) { s.close(); }
main() {
  while (…) {
                                          <S, closed, U>
    Socket s = new Socket(); //S
                     <S, init, U>         <S, closed, ¬U> <S, init, ¬U >
    open(s);         <S, connected, U>    <S, closed, ¬U> <S, init, ¬U > <S, connected, ¬U>
    talk(s);         <S, connected, U>    <S,  err, ¬U> ×  ….
    dispose(s);      <S, closed, U>
  }
}
```

Live analysis to the rescue

- Preliminary live analysis oracle
- On-the-fly remove unreachable configurations

Unique Abstraction

What about aliasing?

```
class SocketHolder {Socket s; }
Socket makeSocket() { return new Socket(); // A }
open(Socket s) {
  s.connect();
}
talk(Socket s) {
  s.getOutputStream()).write("hello");
}
dispose(Socket s) { h.s.close(); }
main() {
  while(…) {
     SocketHolder h = new SocketHolder();
     h.s = makeSocket();       <A, init, U>
     Socket s = makeSocket();  <A, init , ¬U >
     open(h.s);
     talk(h.s);                <A, init, ¬U > <A, connected, ¬U >
     dispose(h.s);             <A, err, ¬U> × ....
     open(s);
     talk(s);
  }
}
```


Access Path Must Abstraction

AS := { < Abstract Object, TypeState, Unique, Must, May> }

- Unique Abstraction +
  - *Must :=* **set of symbolic access paths (x.f.g….) that *must* point to the object**
  - *May :=* **false iff *all* possible access paths appear in *Must* set**
- Flow functions
  - **Only track access paths to "interesting" objects**
    - Limits computational work dramatically
    - Less precise than shape analysis
  - **Always sound to discard Must set and set May := true**
    - Allows k-limiting. Crucial for scalability.
  - **Parameters**
    - *Width*: maximum cardinality of *Must* Set
    - *Depth*: maximum length of an individual access path
    - "*interesting" objects*: which objects to track precisely
      - currently: typestate objects
  - **Typestate transition for *e.op()* if (*e* ∈ *Must*) ∨ (*May* ∧ *mayPointTo(e,I)*)**
  - **Strong Updates**
    - allowed for ***e.op()*** when ***e* ∈ *Must*** or unique logic allows it
- Works sometimes (91%)

→ Access Path Must Abstraction →

Better aliasing!

```
class SocketHolder {Socket s; }
Socket makeSocket() { return new Socket(); // A }
init(Socket t) {     <A, init , ¬U, {h.s, t}, ¬May>  <A, init, ¬U, {s}, ¬May >
  t.connect();       <A, connected , ¬U, {h.s, t}, ¬May>  <A, init, ¬U, {s}, ¬May >
}
talk(Socket u) {                 <A, connected , ¬U, {h.s, u}, ¬May>  <A, init, ¬U, {s}, ¬May >
  u.getOutputStream()).write("hello");
}                                <A, connected , ¬U, {h.s, u}, ¬May> ✓ …
dispose(Socket s) { h.s.close(); }
main() {
  while(…) {
     SocketHolder h = new SocketHolder();
     h.s = makeSocket();        <A, init, U, {h.s}, ¬May>
     Socket s = makeSocket();   <A, init , ¬U, {h.s}, ¬May>  <A, init, ¬U, {s}, ¬May >
     init(h.s);
     talk(h.s);                 <A, connected , ¬U, {h.s}, ¬May>  <A, init, ¬U, {s}, ¬May >
     dispose(h.s);
     init(s);
     talk(s);
  }
}
```


What about destructive updates?

→ Access Path Must Abstraction →

```
class SocketHolder {  Socket s;  }
Socket makeSocket() { return new Socket(); // A }
open(Socket l) {
   l.connect();
}
talk(Socket s) { s.getOutputStream()).write("hello"); }
dispose(Socket s) { h.s.close(); }
main() {
  Set<SocketHolder> set = new HashSet<SocketHolder>();
  while(…) {
     SocketHolder h = new SocketHolder();
     h.s = makeSocket();          <A, init, U, {h.s}, ¬May>
     set.add(h);                  <A, init , U, {h.s}, May>
  } ;
  for (Iterator<SocketHolder> it = set.iterator(); …) {
     Socket g = it.next().s;      <A, init, ¬U, {}, May >
     open(g);                     <A, init, ¬U, {}, May >, <A, connected, ¬U, {}, May >
     talk(g);                     <A, err, ¬U, {}, May > …
     dispose(g);
  }
}
```

Access Path Focus Abstraction

AS := { < Abstract Object, TypeState, Unique, Must, May, MustNot> }

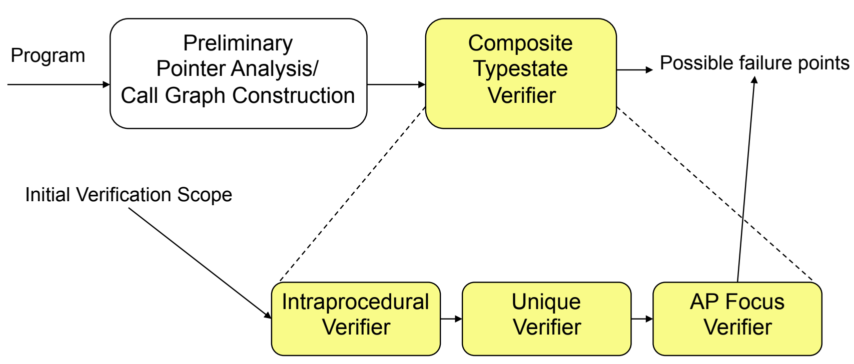
- **Access Path Must Abstraction +**
  - *MustNot* := set of symbolic access paths that *must not* point to the object
- **Flow functions**
  - **Focus** operation when “interesting” things happen
    - “materialization”, “focus”, “case splitting”
    - ***e.op()*** **on *< A, T, u, Must, May, MustNot>*, generate 2 factoids:**
      - ***< A, δ(T), u, Must U {e}, May, MustNot>***
      - ***< A, T, u, Must, May, MustNot U {e} >***
    - Interesting Operations
      - Typestate changes
      - Observable polymorphic dispatch
    - Allows k-limiting. Crucial for scalability
    - Allowed to limit exponential blowup due to focus
      - Current heuristic: discard *MustNot* before each focus operation
- **Works sometimes (95.6%)**

Access Path Focus Abstraction

Recover from destructive updates

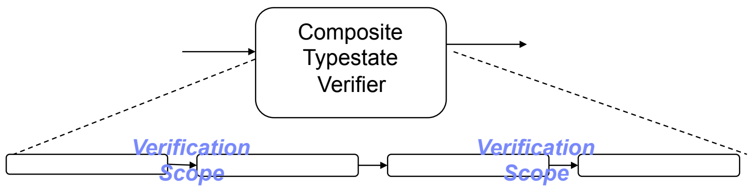
```
class SocketHolder {  Socket s;  }
Socket makeSocket() { return new Socket(); // A }
open(Socket t) {
                         <A, init, ¬U, {}, May, {} >
   t.connect();
                         <A, init, ¬U, {}, May, {¬t} >, <A, connected, ¬U, {t}, May, {}>
}
talk(Socket s) {         <A, init, ¬U, {}, May, {¬g,¬s}>, <A, connected, ¬U, {g,s}, May, {}>
  s.getOutputStream()).write("hello");   ✓
}
dispose(Socket s) { h.s.close(); }
main() {
  Set<SocketHolder> set = new HashSet<SocketHolder>();
  while(…) {
     SocketHolder h = new SocketHolder();
     h.s = makeSocket();          <A, init, U, {h.s}, ¬May, {}>
     set.add(h);                  <A, init, U, {h.s}, May, {}>
  }
  for (Iterator<SocketHolder> it = set.iterator(); …) {
     Socket g = it.next().s;
     open(g);                     <A, init, ¬U, {}, May, {}>
     talk(g);                     <A, init, ¬U, {}, May, {¬g}>, <A, connected, ¬U, {g}, May, {}>
     dispose(g);
  }
}
```

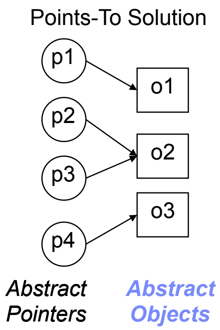

Analysis Overview



Intraprocedural Verifier

- **Single-procedure version of Access Path Focus abstraction**
- **Worst-case assumptions at method entry, calls**
  - Mitigated by live analysis
- **Works sometimes (66%)**





Potential Point of Failure (PPF):
< abstract object, program statement >
Verification Scope: Set of PPF

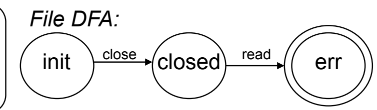
```
File f = new File();        // F1
f.read();                   // R1
f.close();
f.read();                   // R2
```

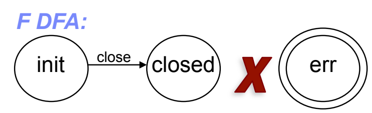
V_{IN} = { <F1,R1>, <F1, R2> }
V_{OUT} = { <F1, R2> }

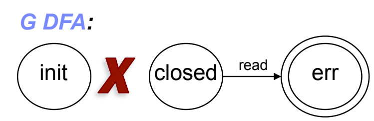
Flow-Insensitive Pruning

- From alias oracle, build typestate DFA for each abstract object
- Prune verification scope by DFA reachability
- It works sometimes (30%)

```
File f = new File();      // F
f.close();
File g = new File();      // G
g.read();
```

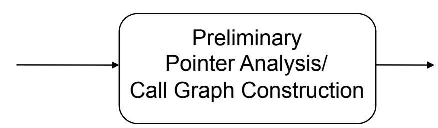






Sparsification

- **Separation (solve for each abstract object separately)**
- **"Slicing": discard branches of supergraph that cannot affect abstract semantics**
  - Identify program variables that might appear k-limited access path
    - K-step reachability from typestate objects from prelim. pointer analysis
  - Identify call graph nodes that might
    - modify these variables
    - cause typestate transitions (depends on incoming verification scope)
  - Discard any nodes that cannot (transitively) affect abstract interpretation
  - **Reduces median supergraph size by 50X**

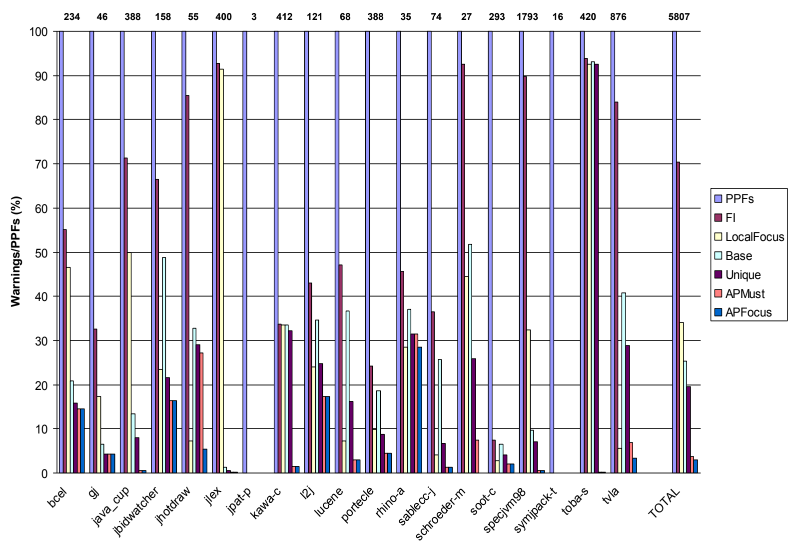


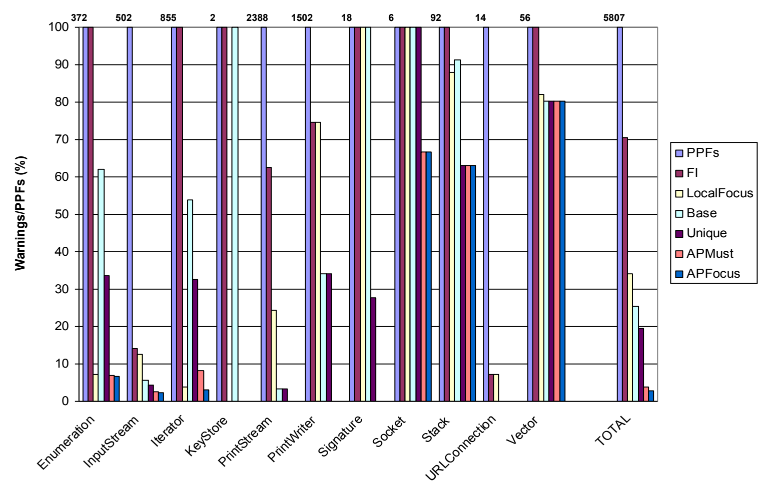
- Typestate verifiers rely on **call graph, fallback alias oracle**
- Current implementation: flow-insensitive, partially context-sensitive pointer-analysis
  - Subset-based, field-sensitive Andersen's
  - SSA local representation
  - On-the-fly call graph construction
  - Unlimited object sensitivity for
    - Collections
    - Containers of typestate objects (e.g. IOStreams)
  - One-level call-string context for some library methods
    - Arraycopy, clone, …
  - Heuristics for reflection (e.g. Livshits et al 2005)
- Details matter a lot
  - e.g. context-insensitive preliminary: later stages time out, terrible precision

Typestate Properties for J2SE libraries

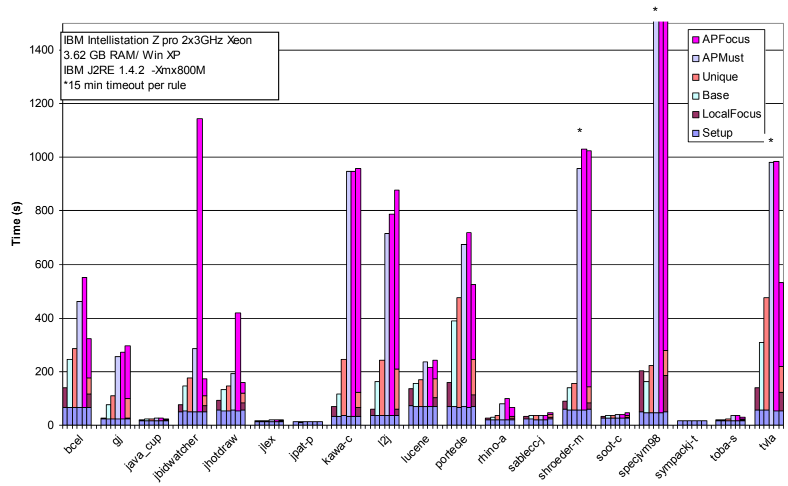
Name	Description
Enumeration	Call `hasNextElement` before `nextElement`
InputStream	Do not read from a closed `InputStream`
Iterator	Do not call `next` without first checking `hasNext`
KeyStore	Always initialize a `KeyStore` before using it
PrintStream	Do not use a closed `PrintStream`
PrintWriter	Do not use a closed `PrintWriter`
Signature	Follow initialization phases for `Signature`
Socket	Do not use a `Socket` until it is connected
Stack	Do not peek or pop an empty `Stack`
URLConn	Illegal operation performed when already connected
Vector	Do not access elements of an empty `Vector`

Benchmark	Classes	Methods	Bytecode Statements	Contexts
bcel	1,723	7,130	1,474,264	13,725
gj	230	2,362	139,305	2,521
java_cup	123	661	53,296	990
jbidwatcher	1,182	4,994	1,029,507	7,030
jhotdraw	1,688	6,337	1,400,640	11,203
jlex	111	473	44,736	776
jpat-p	64	225	17,783	269
kawa-c	612	3,027	141,527	3,438
l2j	838	4,247	877,077	6,438
lucene	1,783	6,694	1,474,334	12,576
portecle	1,800	6,737	1,481,249	13,430
rhino-a	196	1,293	92,225	1,645
sablecc-j	391	2,144	96,201	2,747
schroeder-m	1,459	5,215	1,367,432	9,682
soot-c	665	2,764	144,554	3,272
specjvm98	965	4,673	979,159	8,152
symjpack-t	74	305	80,508	351
toba-s	163	760	65,415	1,169
tvla	346	2,077	139,474	12,874





Running time



Limitations

- **Limitations of analysis (~50%)**
  - Aliasing
  - Path sensitivity
  - Return values

```
if (!stack.isEmpty()) stack.pop();
vector.get(vector.size()-1);
```

Not always straightforward (encapsulation)

```
if (!foo.isAnEmptyFoo()) foo.popFromAStack();
```

- **Limitations of typestate abstraction (~50%)**
  - Application logic bypasses DFA, still OK

```
if (itsABlueMoon) stack.pop();
vector.get(numberOfPixels/2);
try {
   emptyStack.pop();
catch (EmptyStackException e) {
   System.out.println("I expected that.");
}
```


Some related work

- **ESP**
  - Das *et al.* PLDI 2002
    - Two-phase approach to aliasing (unsound strong updates)
    - Path-sensitivity ("property simulation")
  - Dor *et al.* ISSTA 2004
    - Integrated typestate and alias analysis
    - Tracks overapproximation of *May* aliases
- **Type Systems**
  - Vault/Fugue
    - Deline and Fähndrich: `adoption` and `focus`
  - CQUAL
    - Foster *et al.* 02: linear types
    - Aiken *et al.* 03: `restrict` and `confine`
- **Alias Analysis**
  - Landi-Ryder 92, Choi-Burke-Carini 93, Emami-Ghiya-Hendren 95, Wilson-Lam 95, ….
  - Shape Analysis: Chase-Wegman-Zadeck 90, Hacket-Rugina 05, TVLA (Sagiv-Reps-Wilhelm), …