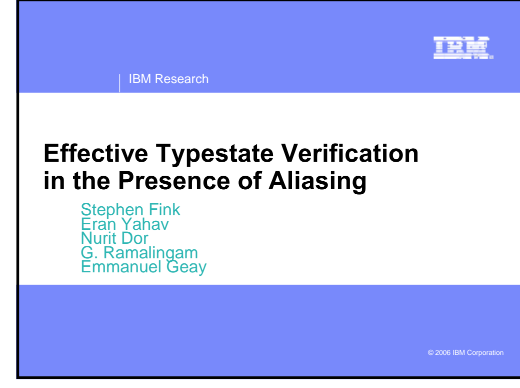
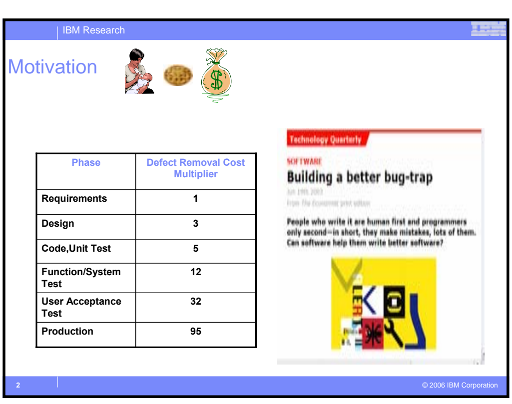
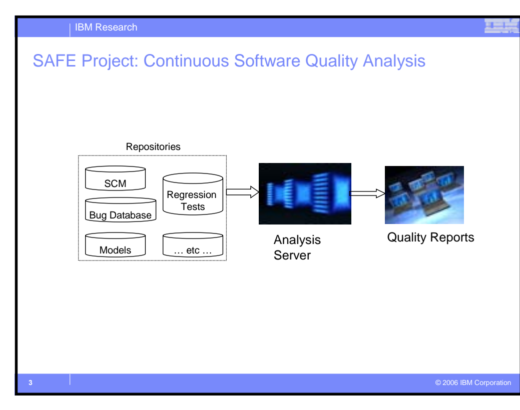
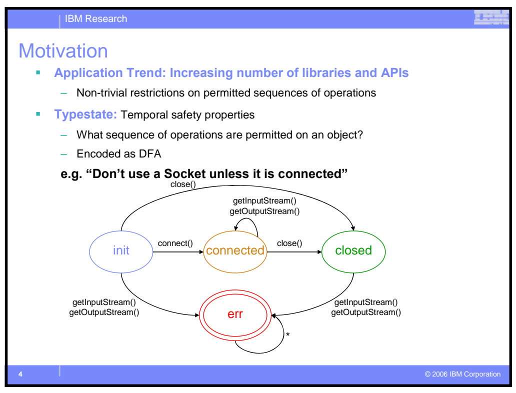
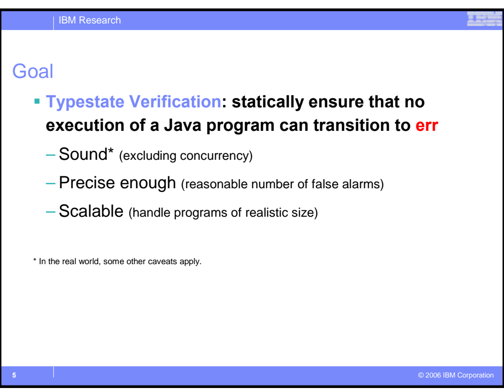


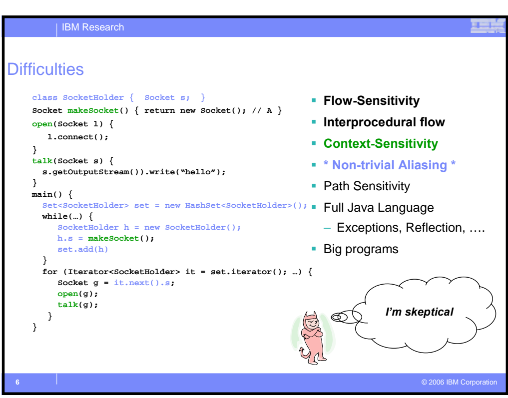


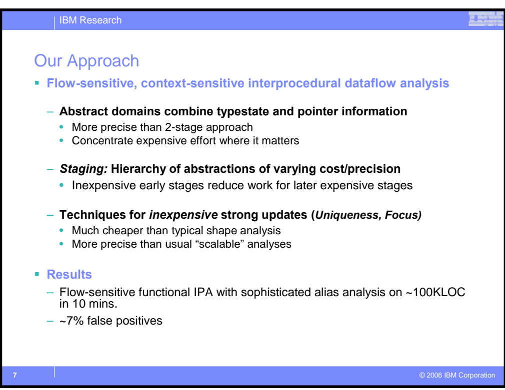
Phase	Defect Removal Cost Multiplier
Requirements	1
Design	3
Code,Unit Test	5
Function/System Test	12
User Acceptance Test	32
Production	95

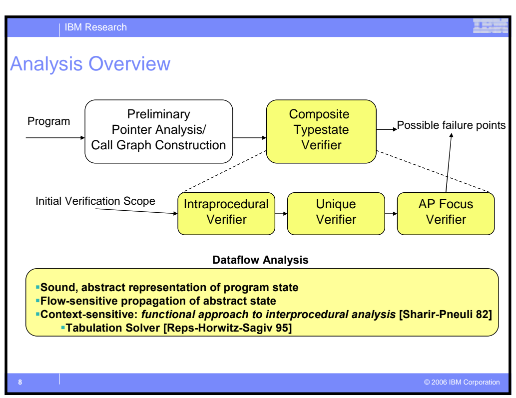


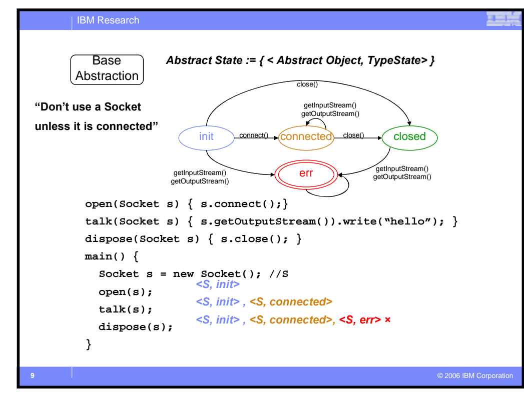

IBM Research
Motivation
Application Trend: Increasing number of libraries and APIs
Non-trivial restrictions on permitted sequences of operations
Typestate: Temporal safety properties
What sequence of operations are permitted on an object?
Encoded as DFA
e.g. "Don't use a Socket unless it is connected"
close()
getInputStream()
getOutputStream()
init
connect()
connected
close()
closed
getInputStream()
getOutputStream()
err
getInputStream()
getOutputStream()
*
4
© 2006 IBM Corporation


IBM Research
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)
* In the real world, some other caveats apply.
5
© 2006 IBM Corporation


IBM Research
Difficulties
class SocketHolder { Socket s; }
Socket makeSocket() { return new Socket(); // A }
open(Socket l) {
l.connect();
}
talk(Socket s) {
s.getOutputStream()).write("hello");
}
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);
}
}
Flow-Sensitivity
Interprocedural flow
Context-Sensitivity
* Non-trivial Aliasing *
Path Sensitivity
Full Java Language
Exceptions, Reflection, ….
Big programs
I'm skeptical
6
© 2006 IBM Corporation


IBM Research
Our Approach
Flow-sensitive, context-sensitive interprocedural dataflow analysis
Abstract domains combine typestate and pointer information
More precise than 2-stage approach
Concentrate expensive effort where it matters
Staging: Hierarchy of abstractions of varying cost/precision
Inexpensive early stages reduce work for later expensive stages
Techniques for inexpensive strong updates (Uniqueness, Focus)
Much cheaper than typical shape analysis
More precise than usual "scalable" analyses
Results
Flow-sensitive functional IPA with sophisticated alias analysis on ~100KLOC in 10 mins.
~7% false positives
7
© 2006 IBM Corporation


IBM Research
Analysis Overview
Program
Preliminary Pointer Analysis/ Call Graph Construction
Composite Typestate Verifier
Possible failure points
Initial Verification Scope
Intraprocedural Verifier
Unique Verifier
AP Focus Verifier
Dataflow Analysis
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]
8
© 2006 IBM Corporation


IBM Research
Base Abstraction
Abstract State := { < Abstract Object, TypeState> }
"Don't use a Socket unless it is connected"
init
connected
closed
err
connect()
close()
getInputStream()
getOutputStream()
open(Socket s) { s.connect();}
talk(Socket s) { s.getOutputStream()).write("hello"); }
dispose(Socket s) { s.close(); }
main() {
Socket s = new Socket(); //S
open(s);
talk(s);
dispose(s);
}
<S, init>
<S, init> , <S, connected>
<S, init> , <S, connected>, <S, err> ×
9
© 2006 IBM Corporation

Unique Abstraction

Abstract State := { < Abstract Object, TypeState, UniqueBit> }

- **“UniqueBit” ≈ “∃ exactly one concrete instance of abstract object”**
- **Allows 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);
}
```


© 2006 IBM Corporation

Unique Abstraction

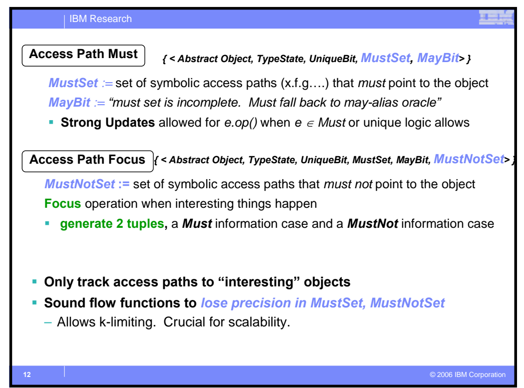
More than just singletons?

```
open(Socket s) { s.connect();}
talk(Socket s) { s.getOutputStream()).write(“hello”); }
dispose(Socket s) { s.close(); }
main() {
  while (…) {
    Socket s = new Socket(); //S      <S, closed, U>
                 <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

© 2006 IBM Corporation



Access Path Focus Abstraction → { < Abstract Object, TypeState, UniqueBit, MustSet, MayBit, MustNotSet> }

```
class SocketHolder {  Socket s;  }
Socket makeSocket() { return new Socket(); // A }
open(Socket t) {
   t.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);
   }
}
```

<A, init, ¬U, {}, May, {} >

<A, init, ¬U, {}, May, {¬ t} >, <A, connected, ¬U, {t}, May, {}>

<A, init, ¬U, {}, May , {¬g,¬s}>, <A, connected, ¬U, {g,s}, May, {}> ✔

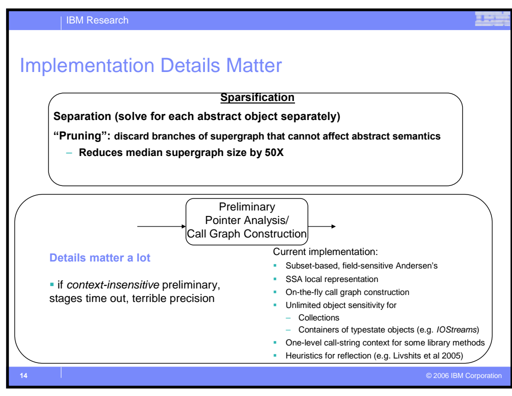
<A, init, U, {h.s}, ¬May, {}>

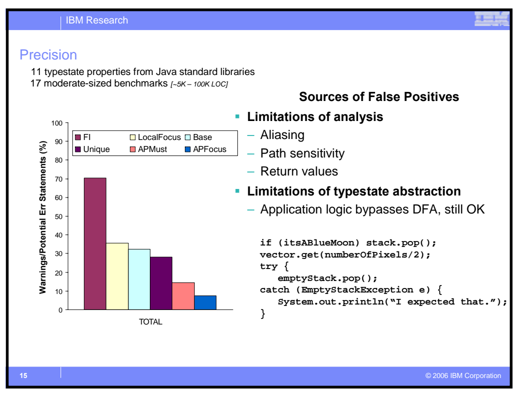
<A, init , U, {h.s}, May ,{}>

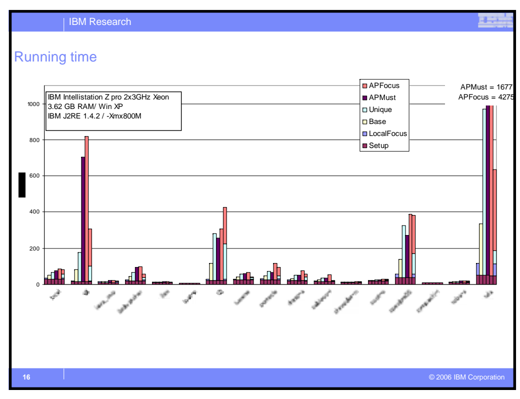
<A, init, ¬U, {}, May , {}>

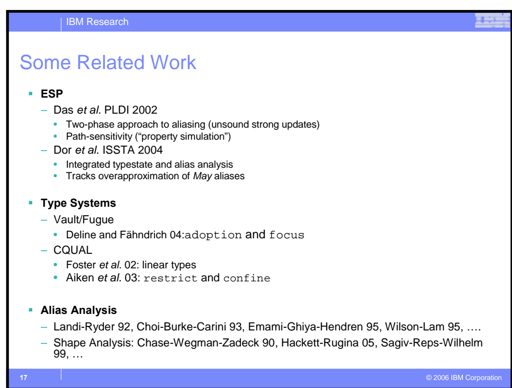
<A, init, ¬U, {}, May , {¬g}>, <A, connected, ¬U, {g}, May, {}>

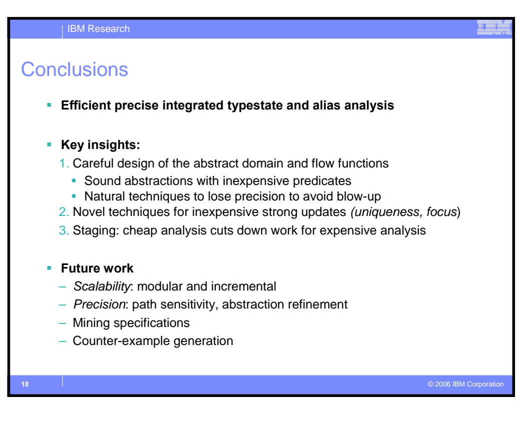
© 2006 IBM Corporation






IBM Research
Running time
IBM Intellistation Z pro 2x3GHz Xeon
3.62 GB RAM/ Win XP
IBM J2RE 1.4.2 / -Xmx800M
APFocus
APMust
Unique
Base
LocalFocus
Setup
APMust = 1677
APFocus = 4275
1000
800
600
400
200
0
16
© 2006 IBM Corporation


IBM Research
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 04: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, Hackett-Rugina 05, Sagiv-Reps-Wilhelm 99, …
17
© 2006 IBM Corporation


IBM Research
Conclusions
Efficient precise integrated typestate and alias analysis
Key insights:
1. Careful design of the abstract domain and flow functions
Sound abstractions with inexpensive predicates
Natural techniques to lose precision to avoid blow-up
2. Novel techniques for inexpensive strong updates *(uniqueness, focus)*
3. Staging: cheap analysis cuts down work for expensive analysis
Future work
Scalability: modular and incremental
Precision: path sensitivity, abstraction refinement
Mining specifications
Counter-example generation
18
© 2006 IBM Corporation

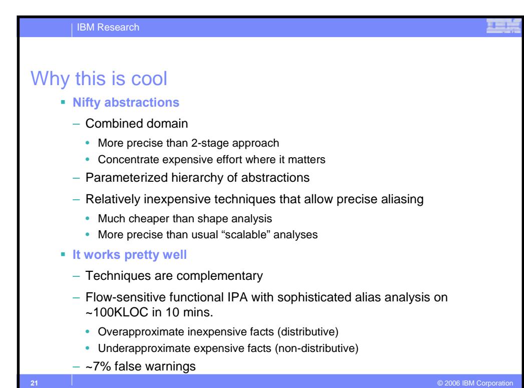
Challenge: Aliasing

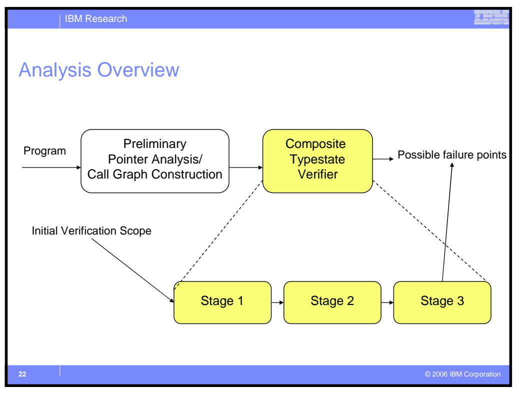
```
void foo(Socket s, Socket t) {
  s.connect();
  t.getInputStream(); // potential error?
}
```

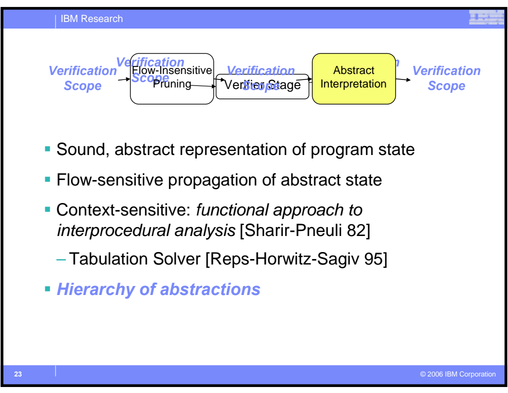
- **Strong Updates may be required**
  - Rules out solely flow-insensitive analysis

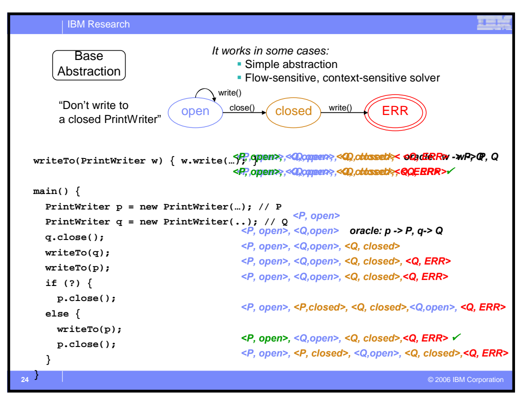
```
void foo(Socket s, Socket t) {
  s.connect();        // s MUST point to connected
  t = s;              // t MUST point to connected
  t.getInputStream();
}
```

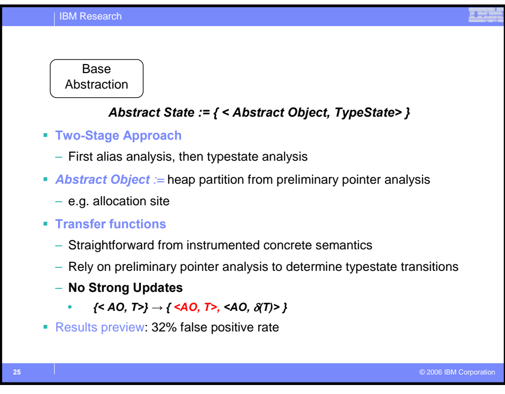

© 2006 IBM Corporation

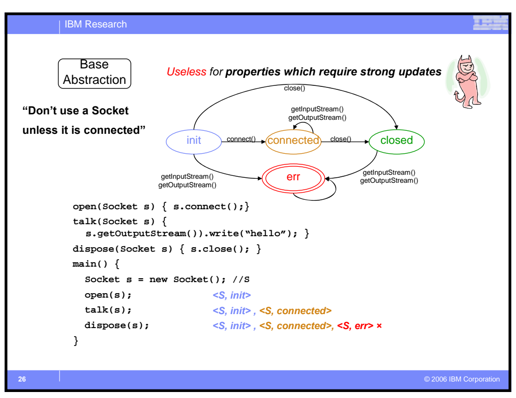


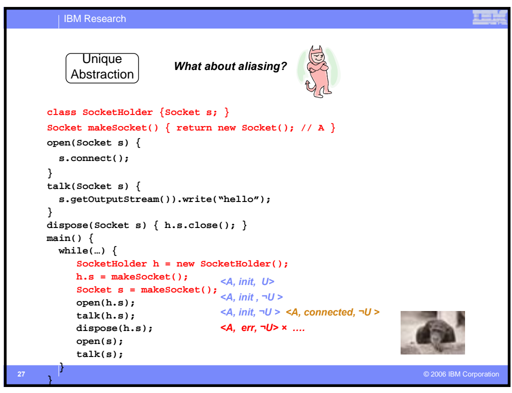

IBM Research
Analysis Overview
Program
Preliminary Pointer Analysis/ Call Graph Construction
Composite Typestate Verifier
Possible failure points
Initial Verification Scope
Stage 1
Stage 2
Stage 3
22
© 2006 IBM Corporation

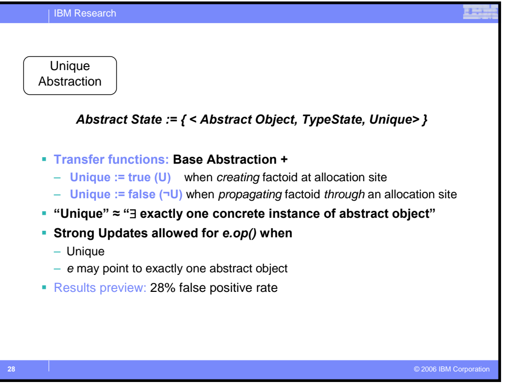

IBM Research
Verification Scope
Flow-Insensitive Pruning
Verifier Stage
Abstract Interpretation
Verification Scope
Sound, abstract representation of program state
Flow-sensitive propagation of abstract state
Context-sensitive: functional approach to interprocedural analysis [Sharir-Pneuli 82]
– Tabulation Solver [Reps-Horwitz-Sagiv 95]
Hierarchy of abstractions
23
© 2006 IBM Corporation

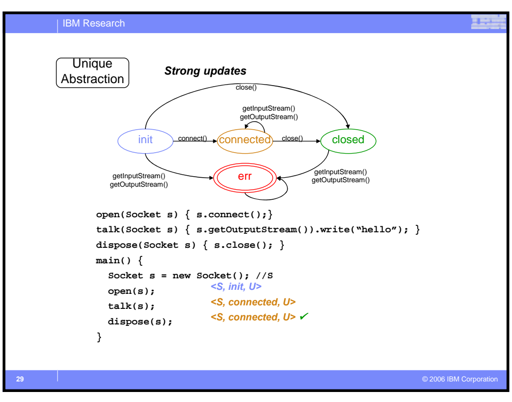

IBM Research
Base Abstraction
It works in some cases:
Simple abstraction
Flow-sensitive, context-sensitive solver
"Don't write to a closed PrintWriter"
open
write()
close()
closed
write()
ERR
writeTo(PrintWriter w) { w.write(…); }
main() {
PrintWriter p = new PrintWriter(…); // P
PrintWriter q = new PrintWriter(..); // Q
q.close();
writeTo(q);
writeTo(p);
if (?) {
p.close();
else {
writeTo(p);
p.close();
}
}
<P, open>
<P, open>, <Q,open>
oracle: p -> P, q-> Q
<P, open>, <Q,open>, <Q, closed>
<P, open>, <Q,open>, <Q, closed>, <Q, ERR>
<P, open>, <Q,open>, <Q, closed>,<Q, ERR>
<P, open>, <P,closed>, <Q, closed>,<Q,open>, <Q, ERR>
<P, open>, <Q,open>, <Q, closed>,<Q, ERR> ✓
<P, open>, <P, closed>, <Q,open>, <Q, closed>,<Q, ERR>
24
© 2006 IBM Corporation

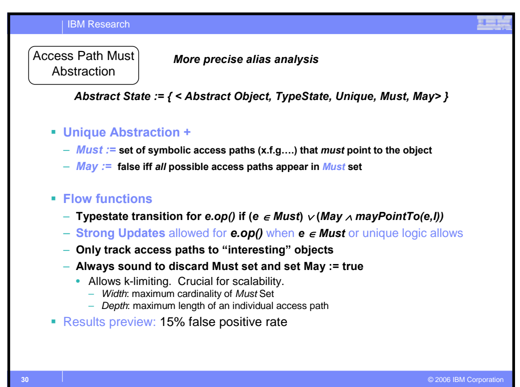

IBM Research
Base Abstraction
Abstract State := { < Abstract Object, TypeState> }
Two-Stage Approach
First alias analysis, then typestate analysis
Abstract Object := heap partition from preliminary pointer analysis
e.g. allocation site
Transfer functions
Straightforward from instrumented concrete semantics
Rely on preliminary pointer analysis to determine typestate transitions
No Strong Updates
{< AO, T>} → { <AO, T>, <AO, δ(T)> }
Results preview: 32% false positive rate
25
© 2006 IBM Corporation


IBM Research
Base Abstraction
Useless for properties which require strong updates
"Don't use a Socket unless it is connected"
init
connect()
connected
close()
closed
err
getInputStream()
getOutputStream()
open(Socket s) { s.connect();}
talk(Socket s) {
s.getOutputStream()).write("hello"); }
dispose(Socket s) { s.close(); }
main() {
Socket s = new Socket(); //S
open(s); <S, init>
talk(s); <S, init> , <S, connected>
dispose(s); <S, init> , <S, connected>, <S, err> ×
}
26
© 2006 IBM Corporation


IBM Research
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); <A, init, ¬U > <A, connected, ¬U >
talk(h.s);
dispose(h.s); <A, err, ¬U> × ….
open(s);
talk(s);
}
}
27
© 2006 IBM Corporation


IBM Research
Unique Abstraction
Abstract State := { < 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
"Unique" ≈ "∃ exactly one concrete instance of abstract object"
Strong Updates allowed for e.op() when
Unique
e may point to exactly one abstract object
Results preview: 28% false positive rate
28
© 2006 IBM Corporation


IBM Research
Unique Abstraction
Strong updates
close()
getInputStream()
getOutputStream()
init
connect()
connected
close()
closed
getInputStream()
getOutputStream()
err
getInputStream()
getOutputStream()
open(Socket s) { s.connect();}
talk(Socket s) { s.getOutputStream()).write("hello"); }
dispose(Socket s) { s.close(); }
main() {
Socket s = new Socket(); //S
open(s);
talk(s);
dispose(s);
}
<S, init, U>
<S, connected, U>
<S, connected, U> ✓
29
© 2006 IBM Corporation


IBM Research
Access Path Must Abstraction
More precise alias analysis
Abstract State := { < 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
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
Only track access paths to "interesting" objects
Always sound to discard Must set and set May := true
Allows k-limiting. Crucial for scalability.
Width: maximum cardinality of Must Set
Depth: maximum length of an individual access path
Results preview: 15% false positive rate
30
© 2006 IBM Corporation

Access Path Focus Abstraction

Abstract State := { < 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**
    - Typestate changes
    - Observable changes in access path predicates
  - ***e.op()*** **on *< A, T, u, Must, May, MustNot>*, generate 2 factoids:**
    1. ***< A, δ(T), u, Must U {e}, May, MustNot>***
    2. ***< A, T, u, Must, May, MustNot U {e} >***
- Results preview: 7% false positive rate

© 2006 IBM Corporation

Access Path Must Abstraction → ***Better aliasing!***

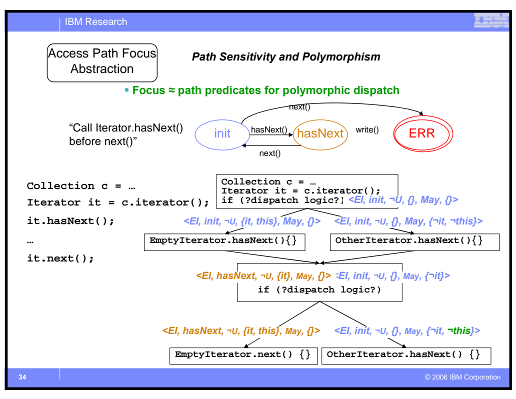
```
class SocketHolder {Socket s; }
Socket makeSocket() { return new Socket(); // A }
init(Socket t) {
  t.connect();           <A, init , ¬U, {h.s, t}, ¬May>  <A, init, ¬U, {s}, ¬May >
}                        <A, connected , ¬U, {h.s, t}, ¬May>  <A, init, ¬U, {s}, ¬May >
talk(Socket u) {                       <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();
     Socket s = makeSocket();  <A, init,  U, {h.s}, ¬May>
     init(h.s);                <A, init , ¬U, {h.s}, ¬May>  <A, init, ¬U, {s}, ¬May >
     talk(h.s);                <A, connected , ¬U, {h.s}, ¬May>  <A, init, ¬U, {s}, ¬May >
     dispose(h.s);
     init(s);
     talk(s);
  }
}
```

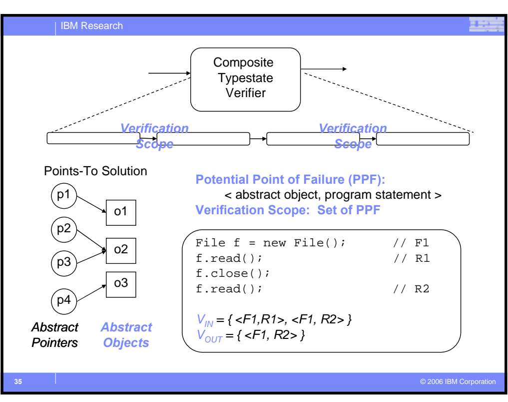

© 2006 IBM Corporation

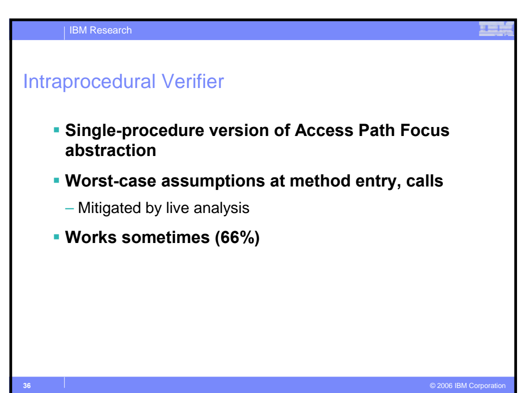
Access Path Must Abstraction → ***What about destructive updates?***

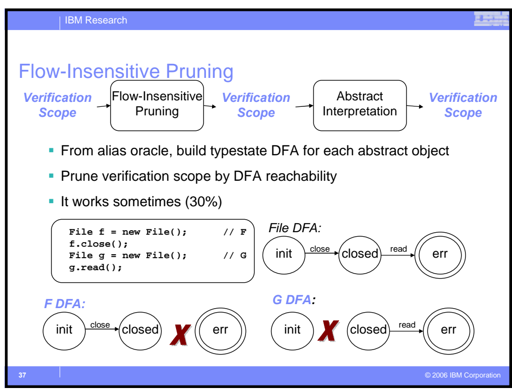
```
class SocketHolder {  Socket s;  }
Socket makeSocket() { return new Socket(); // A }
open(Socket l) {
   l.connect();
}
talk(Socket s) { s.getOutputStream()).write("hello"); }
dispose(Socket s) { h.s.close(); }
main() {
  Set<SocketHolder> set = new HashSet<SocketHolder>();
  while(…) {
     SocketHolder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

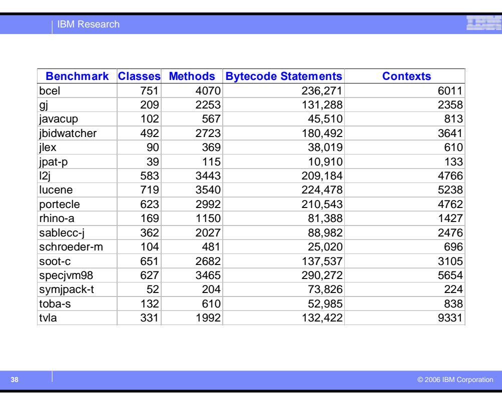

© 2006 IBM Corporation


IBM Research
Access Path Focus Abstraction
Path Sensitivity and Polymorphism
Focus ≈ path predicates for polymorphic dispatch
"Call Iterator.hasNext() before next()"
init
hasNext
ERR
Collection c = …
Iterator it = c.iterator();
it.hasNext();
…
it.next();
if (?dispatch logic?)
EmptyIterator.hasNext(){}
OtherIterator.hasNext(){}
EmptyIterator.next() {}
OtherIterator.hasNext() {}
34
© 2006 IBM Corporation

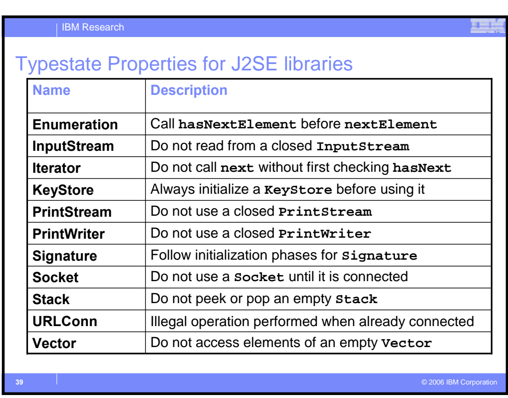

IBM Research
Composite Typestate Verifier
Verification Scope
Verification Scope
Points-To Solution
p1
p2
p3
p4
o1
o2
o3
Abstract Pointers
Abstract Objects
Potential Point of Failure (PPF):
< abstract object, program statement >
Verification Scope: Set of PPF
File f = new File(); // F1
f.read(); // R1
f.close();
f.read(); // R2
V_IN = { <F1,R1>, <F1, R2> }
V_OUT = { <F1, R2> }
35
© 2006 IBM Corporation


IBM Research
Intraprocedural Verifier
Single-procedure version of Access Path Focus abstraction
Worst-case assumptions at method entry, calls
– Mitigated by live analysis
Works sometimes (66%)
36
© 2006 IBM Corporation

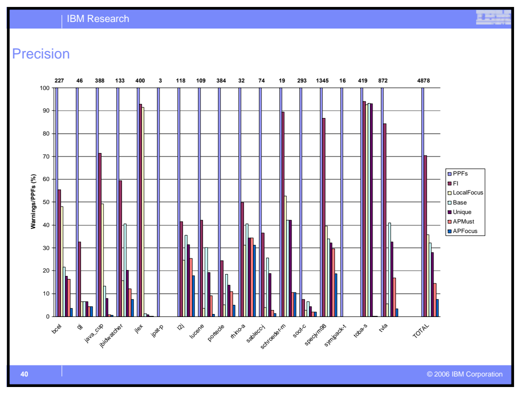


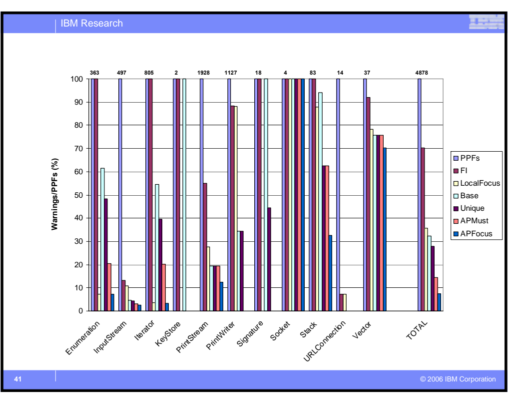


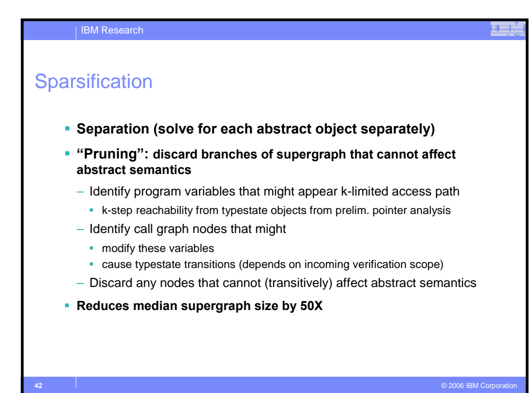
Benchmark	Classes	Methods	Bytecode Statements	Contexts
bcel	751	4070	236,271	6011
gj	209	2253	131,288	2358
javacup	102	567	45,510	813
jbidwatcher	492	2723	180,492	3641
jlex	90	369	38,019	610
jpat-p	39	115	10,910	133
l2j	583	3443	209,184	4766
lucene	719	3540	224,478	5238
portecle	623	2992	210,543	4762
rhino-a	169	1150	81,388	1427
sablecc-j	362	2027	88,982	2476
schroeder-m	104	481	25,020	696
soot-c	651	2682	137,537	3105
specjvm98	627	3465	290,272	5654
symjpack-t	52	204	73,826	224
toba-s	132	610	52,985	838
tvla	331	1992	132,422	9331

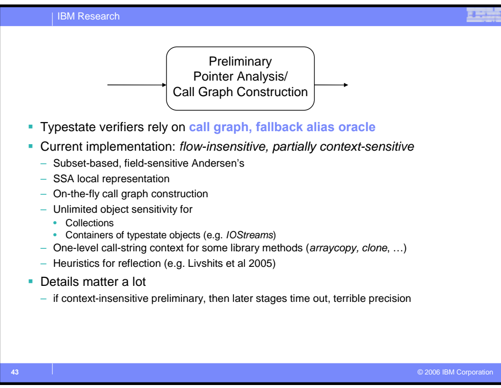


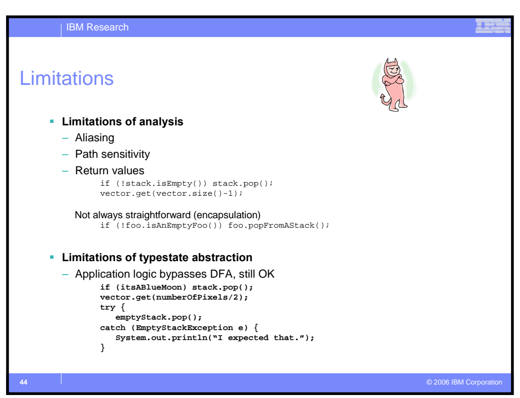
Name	Description
Enumeration	Call `hasNextElement` before `nextElement`
InputStream	Do not read from a closed `InputStream`
Iterator	Do not call `next` without first checking `hasNext`
KeyStore	Always initialize a `KeyStore` before using it
PrintStream	Do not use a closed `PrintStream`
PrintWriter	Do 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`

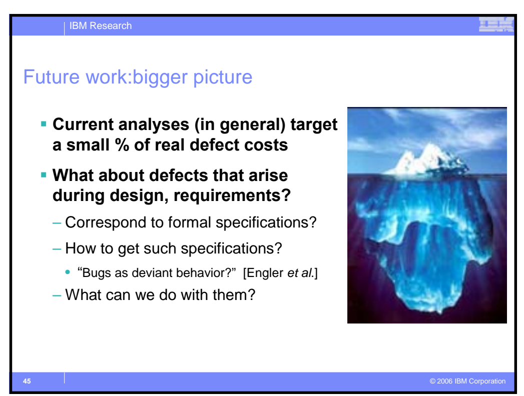

IBM Research
Precision
227 46 388 133 400 3 118 109 384 32 74 19 293 1345 16 419 872 4878
Warnings/PPFs (%)
PPFs
FI
LocalFocus
Base
Unique
APMust
APFocus
40
© 2006 IBM Corporation


IBM Research
363 497 805 2 1928 1127 18 4 83 14 37 4878
Warnings/PPFs (%)
Enumeration
InputStream
Iterator
KeyStore
PrintStream
PrintWriter
Signature
Socket
Stack
URLConnection
Vector
TOTAL
PPFs
FI
LocalFocus
Base
Unique
APMust
APFocus
41
© 2006 IBM Corporation


IBM Research
Sparsification
Separation (solve for each abstract object separately)
"Pruning": 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 semantics
Reduces median supergraph size by 50X
42
© 2006 IBM Corporation


IBM Research
Preliminary
Pointer Analysis/
Call Graph Construction
Typestate verifiers rely on call graph, fallback alias oracle
Current implementation: flow-insensitive, partially context-sensitive
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
if context-insensitive preliminary, then later stages time out, terrible precision
43
© 2006 IBM Corporation


IBM Research
Limitations
Limitations of analysis
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
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.");
}
44
© 2006 IBM Corporation


IBM Research
Future work:bigger picture
Current analyses (in general) target a small % of real defect costs
What about defects that arise during design, requirements?
Correspond to formal specifications?
How to get such specifications?
"Bugs as deviant behavior?" [Engler et al.]
What can we do with them?
45
© 2006 IBM Corporation