Eric Bodden McGill University

Detecting non-local violations of API contracts in large software systems

```java
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r,w);
    }

}
```

One month later...

```java
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r,w);
    }

}
```

```
static void copyAll(Reader r, Writer w) {

    int buffer = 0;
    try {
        do {
            buffer = r.read();
        }
        r.close();
        w.close();
    }
}
```

void java.io.Writer.close() throws IOException

Closes the stream, flushing it first. Once the stream has been closed, further write() or flush() invocations will cause an IOException to be thrown. Closing a previously closed stream has no effect.

Throws:

Press 'F2' for focus.

One hour later...

```java
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r,w);
    }

}
```

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.io

Class PrintWriter

```
java.lang.Object
  └ java.io.Writer
      └ java.io.PrintWriter
```

public class **PrintWriter**
extends Writer

Print formatted representations of objects to a text-output stream. This class implements all of the print methods found in PrintStream. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.

Unlike the PrintStream class, if automatic flushing is enabled it will be done only when one of the println() methods is invoked, rather than whenever a newline character happens to be output. The println() methods use the platform's own notion of line separator rather than the newline character.

Methods in this class never throw I/O exceptions. The client may inquire as to whether any errors have occurred by invoking checkError().

Since:
JDK1.1

```
public class Lo

    private st

    public sta
        w = n
    }

    public static voi  log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r,w);
    }

}
```

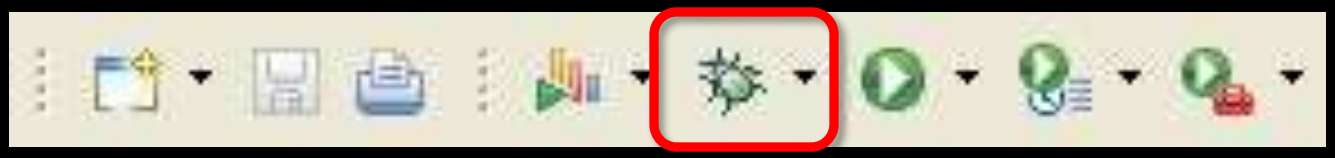
Possible API violation on Writer w

w implements type Writer, whose contract is violated.

At this line, w may be in state closed. – Why?

The operation write(String) is not allowed in that state.

I'm feeling lucky | Show API contract | Ignore



```
public class

    private

    public s
        w =
    }

    public static  d log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r,w);
    }

}
```

API violation on Writer w

w implements type Writer, whose contract is violated.

w is currently in state closed, caused by StreamUtil.copyAll(..)

The operation write(String) is not allowed in that state.

StreamUtil.copyAll(..) | Show API contract | Ignore

Detecting
non-local violations
of API contracts in
large software systems

Pure runtime monitoring

Current approach...

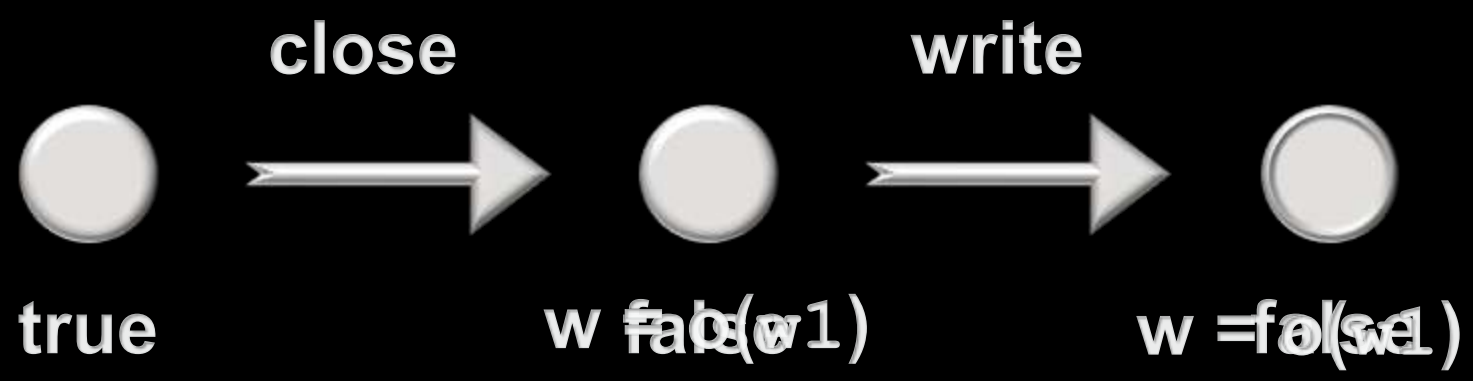
```
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r,w);
    }

}
```

“no write after close”


```
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r,w);
    }

}
```

Current approach: Tracematches

```
tracematch(Writer w) {
   sym close after returning:
      call(* Writer.close()) && target(w);
   sym write before:
      call(* Writer.write(..)) && target(w);

   close write
   {
      System.out.println(
            "Writer "+w+" was closed!");
   }
}
```




```
{
    System.out.println(
        "Writer " + w + " was closed!");
}
```

```
w1.close();
w1.write("foo");
w1.write("foo");
```

Problem 1:

How to capture an API contract precisely and concisely?

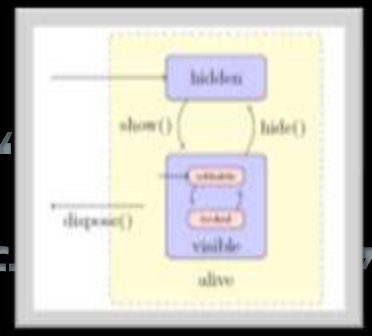
Problem 2:

Potentially large
runtime
overhead

Problem 3:
No static guarantees

novel static API specification language(s)

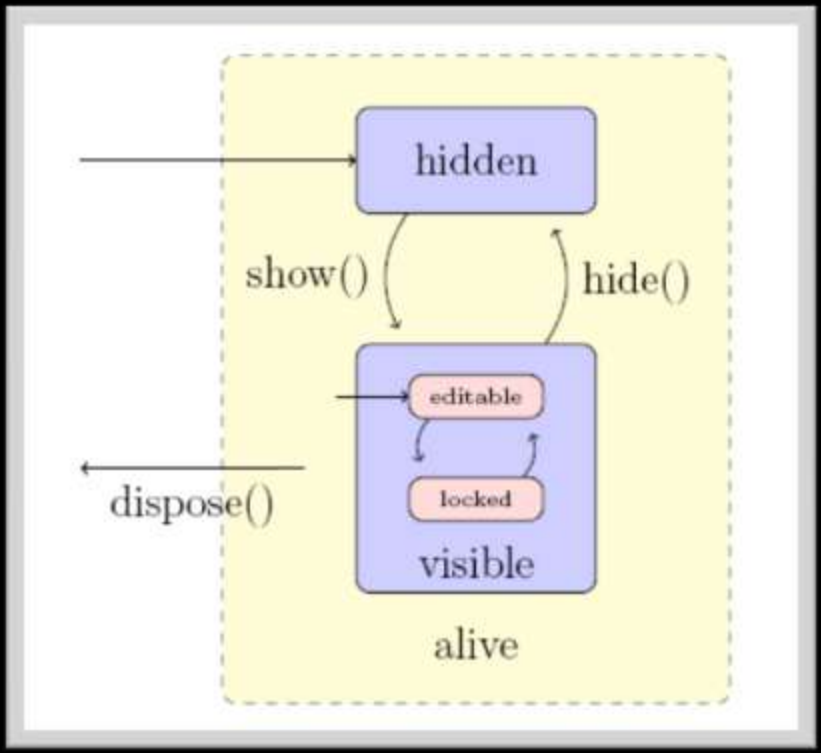
```
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r,w);
    }

}
```



“ af ”

4 novel static program analyses

```
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r,w);
    }

}
```


hidden
show()
hide()
editable
locked
visible
dispose()
alive

Current approach: Tracematches

```
tracematch(Writer w) {
   sym close after returning:
      call(* Writer.close()) && target(w);
   sym write before:
      call(* Writer.write(..)) && target(w);

   close write
   {
      System.out.println(
            "Writer "+w+" was closed!");
   }
}
```

Good:

Can reason about
single objects.

Even better:

Can reason about
combinations
of objects.

Good:

Semantics of regular expression matching well understood.

Bad:

Regular expressions bad at stating required events.

“always close after open”

“open close” ?

Bad:

Little reuse.

“no write after close”
and
“always close after open”

requires two tracematches!

Goal:
Make API specifications
precise and concise

Plan

Investigate a number of actual programs

Plan

Investigate a number of actual programs

What APIs do programs use?

Plan

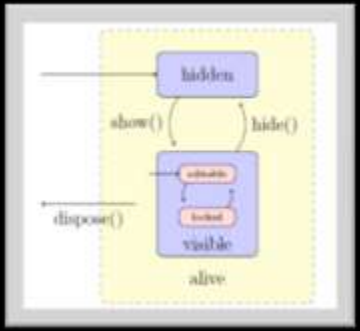
Investigate a number of actual programs

What mistakes do people make?

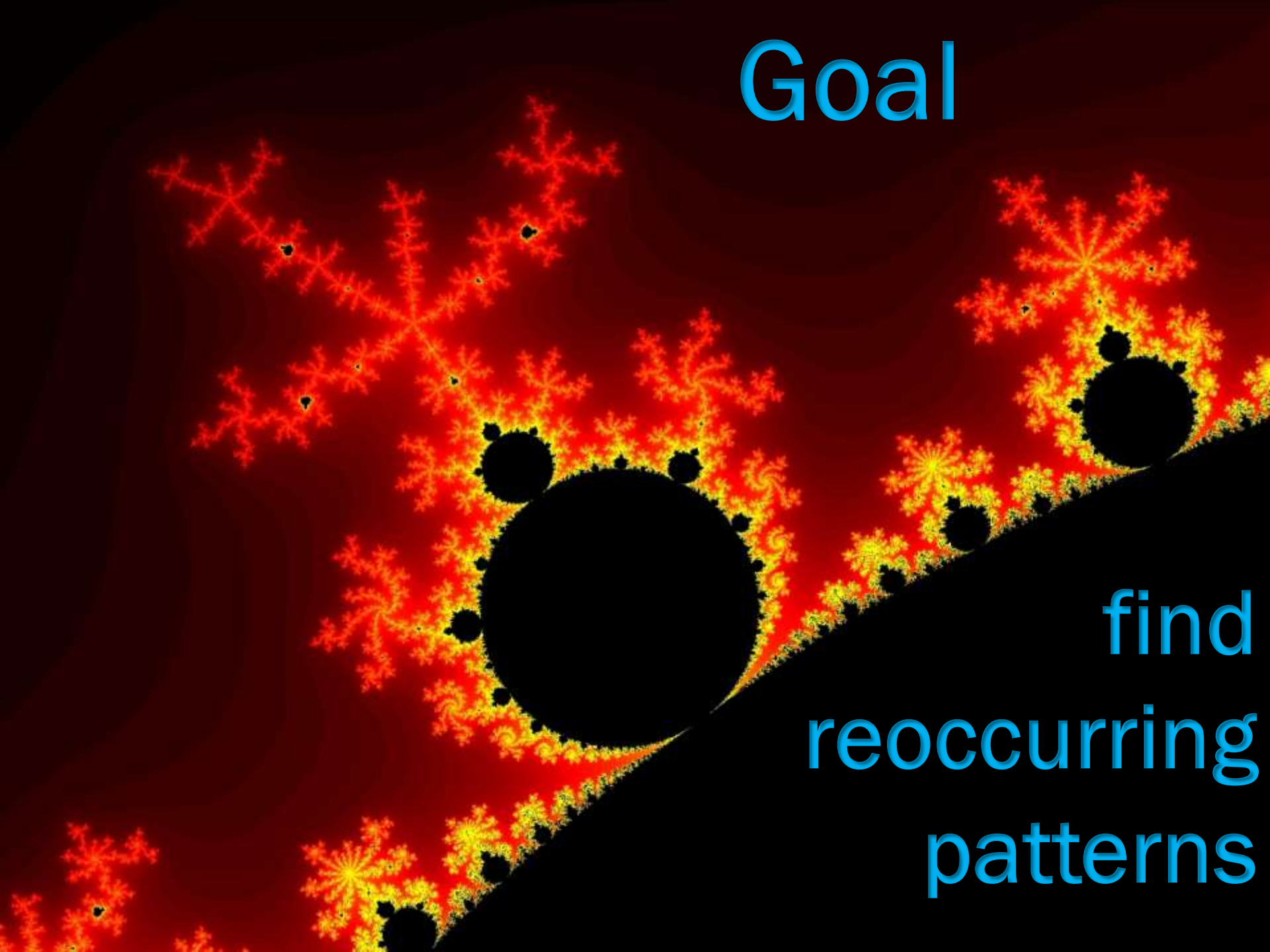
Plan

Investigate a number of actual programs

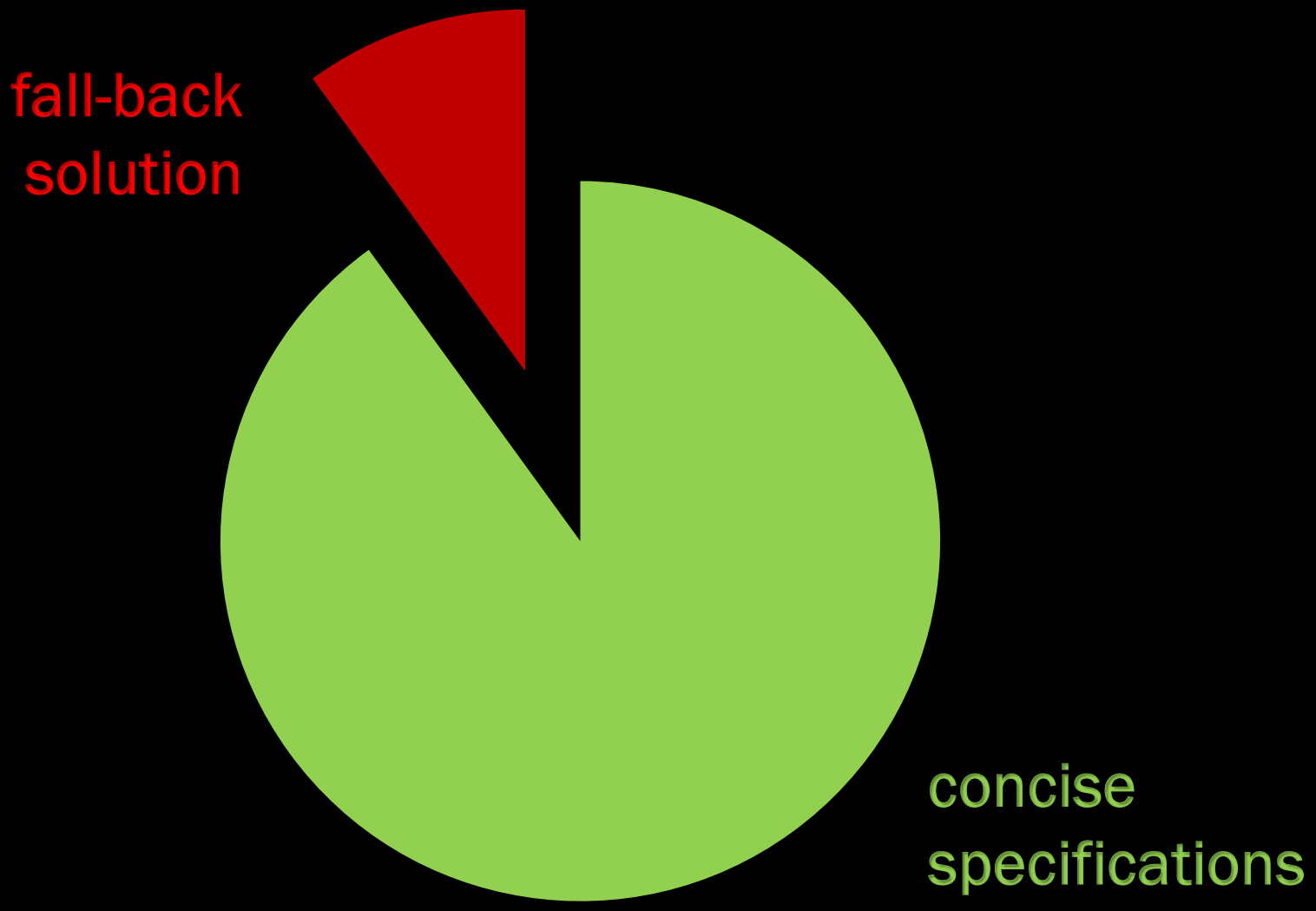
What specifications would have avoided those mistakes?



Goal



find reoccurring patterns



90 / 10

4 novel static program analyses

First static analysis:

“Quick Check”

close write

```
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public st id log(String s) {
        w.wri ;
    }

    public static void (Reader r) {
        StreamUtil.copy c,w);
    }

}
```

) = 0

) = 2

Second static analysis:

“Consistent-shadows analysis”

[close,write]

[close,write]

$\{c_1\} \times \{w_1, w_2\}$

```
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r,w);
    }

}
```

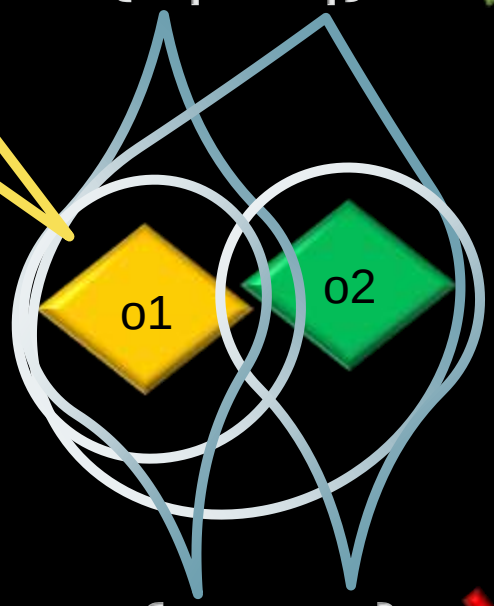
w_1

c_1

w_2

```
new PrintWriter();
```

$\{c_1, w_1\}$



$\{c_1, w_2\}$

Third static analysis:

“Flow-sensitive unnecessary shadows analysis”

close write

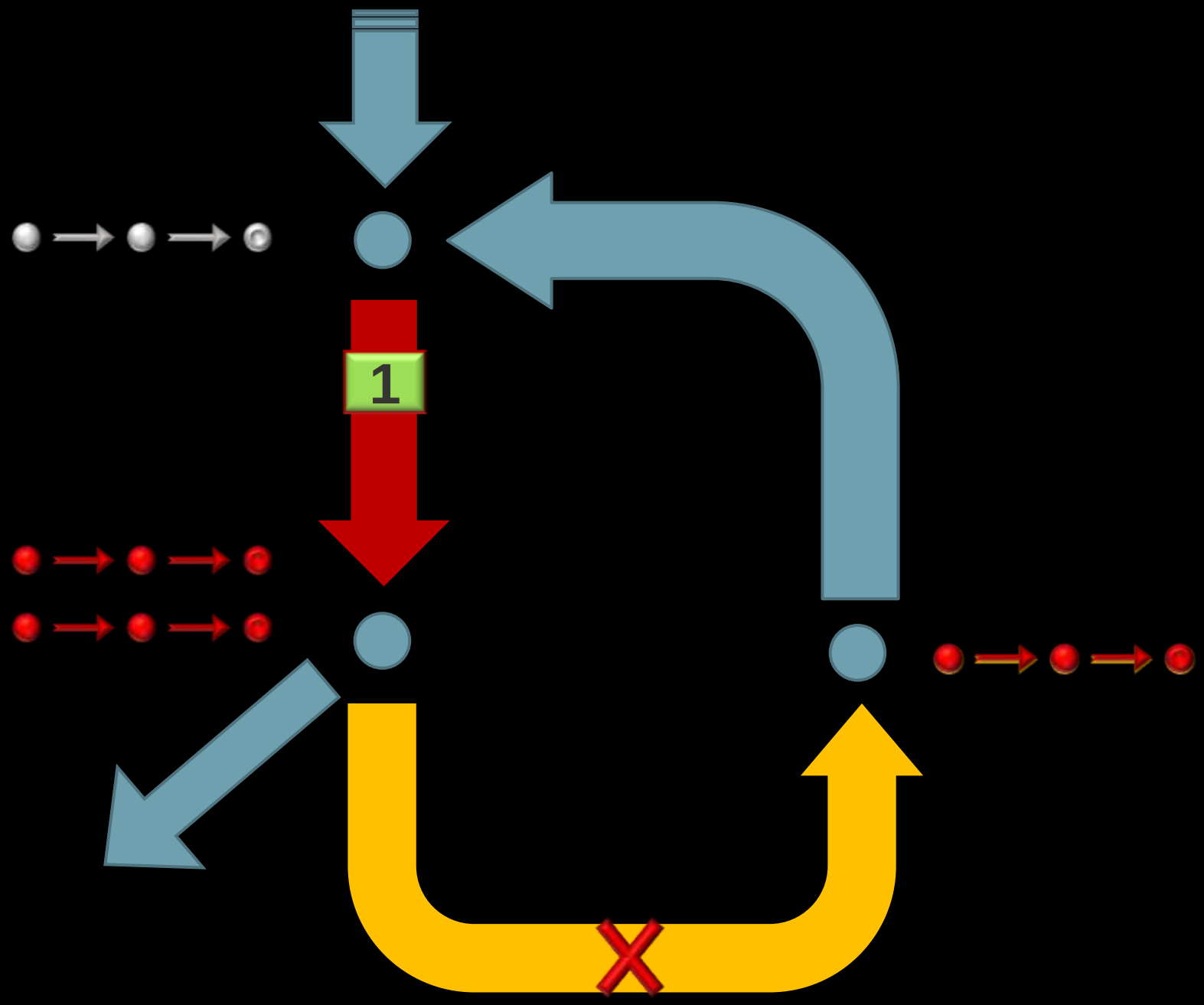
true w = i(w1)-s1 false w = i(w1)-s1,s2

true w ≠ i(w1)-s2 false w = i(w1)-s1,s2

w =i(w1)-s1 ✓

```
w1.close();          //s1

w1.write("foo");     //s2

w1.write("foo");     //s3
```

4th static analysis:
“run-once loop optimization”


1

Results

|.....|.....|.....|.....|.....|.....|.....|.....|.....|.....|.

102 program/tracematch combinations

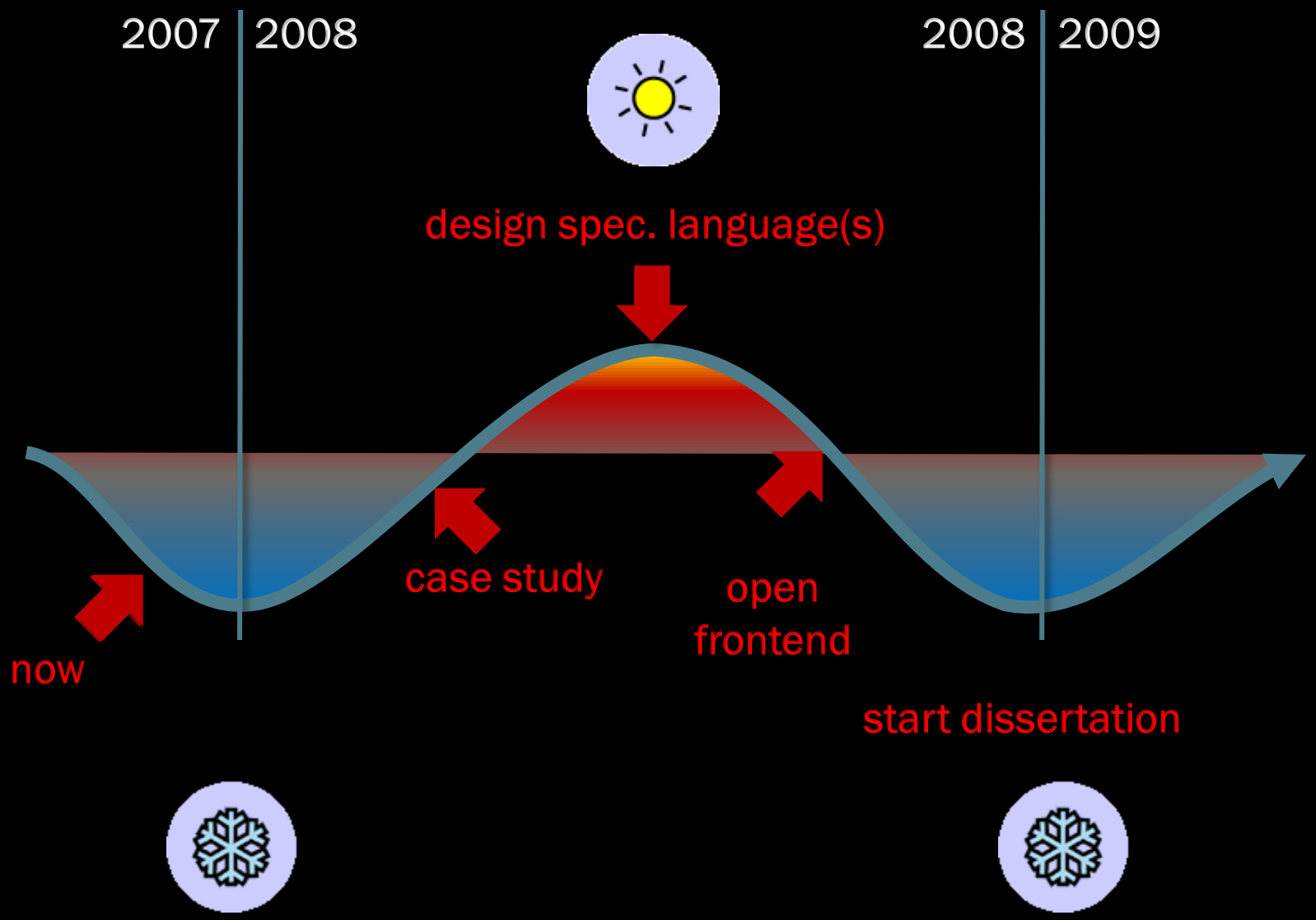
static guarantees in 77 cases

Found 5 programs with bugs or questionable code.

less than 10 shadows in 12 cases

less than 10% overhead in 9 cases

Timeline for completion




E=mc2

Jython / Reader (1/2)

```
private final void FillBuff() {
        ...
        try {
                if ((i = inputStream.read(...)) == -1) {
                        inputStream.close();
                        throw new java.io.IOException();
                }
                else
                        maxNextCharInd += i;
                return;
        }
        ...
}
```

Jython / Reader (2/2)

```
static String getLine(BufferedReader reader, int line)
{
    if (reader == null)
        return "";
    try {
        String text=null;
        for(int i=0; i < line; i++) {
            text = reader.readLine();
        }
        return text;
    } catch (IOException ioe) {
        return null;
    }
}
```

Jython / hasNext (delegate)

```
public Iterator iterator() {
    return new Iterator() {
        Iterator i = list.iterator();
        public void remove() {
            throw new UnsupportedOperationException();
        }
        public boolean hasNext() {
            return i.hasNext();
        }
        public Object next() {
            return i.next();
        }
    };
}
```

pmd / HasNext (old version)

```
private List markUsages(IDataFlowNode inode) {
        ...
        for (Iterator k = ((List)entry.getValue())
                .iterator();k.hasNext();) {
                addAccess(k, inode);
        }
        ...
}

...

private void addAccess(Iterator k, IDataFlowNode inode)
{
        NameOccurrence occurrence =
                (NameOccurrence) k.next();
        ...
}
```

pmd / HasNext (fixed version)

```
private List markUsages(IDataFlowNode inode) {
        ...
        for (NameOccurrence occurrence: entry.getValue())
        {
                addAccess(occurrence, inode);
        }
        ...
}

...

private void addAccess(NameOccurrence occurrence,
                       IDataFlowNode inode) {
        ...
}
```

Benchmark programs

antlr	hsqldb
bloat	jython
chart	lucene
eclipse	pmd
fop	xalan

pentaho	scimark

Tracematches

ASyncIteration	HasNextElem
FailSafeEnum	LeakingSync
FailSafeIter	Reader
HashMap	Writer
HasNext	…

+ program specific tracematches