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...

“Feature X is not working”

“Could you have a look at the log file, please?”

“The log file is empty!”

“Customer says, the log is empty…”

“Oh no!
How come?”

```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();
```

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.

```
        }
        r.close();
        w.close();
    }
}
```

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


A better world...

```
public class Lo

    private st

    public sta
        w = n
    }

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

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

}
```

Possible API violation on Writer w

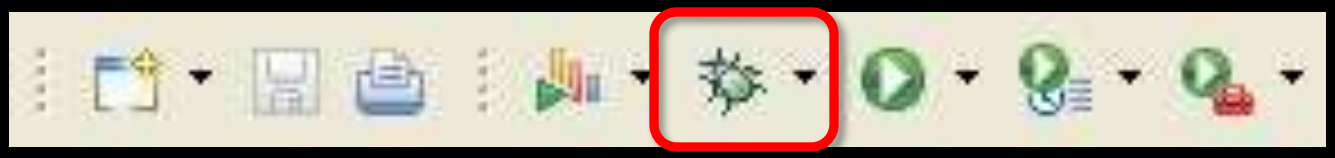
w implements type Writer, whose contract is violated.

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

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

Avoid the violation by making sure that w is not the closed state. - How?

I'm feeling lucky | Show API contract | Ignore



```
public class

    private

    public s
        w =
    }

    public static void 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


Non-local violations involve aliasing through long call chains

API contracts:
Client contracts that come along with an API.

Large software systems:
Complete Java programs
with up to hundreds of
thousands lines of code

Pure runtime monitoring

How does it work today?

```
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);
    }

}
```

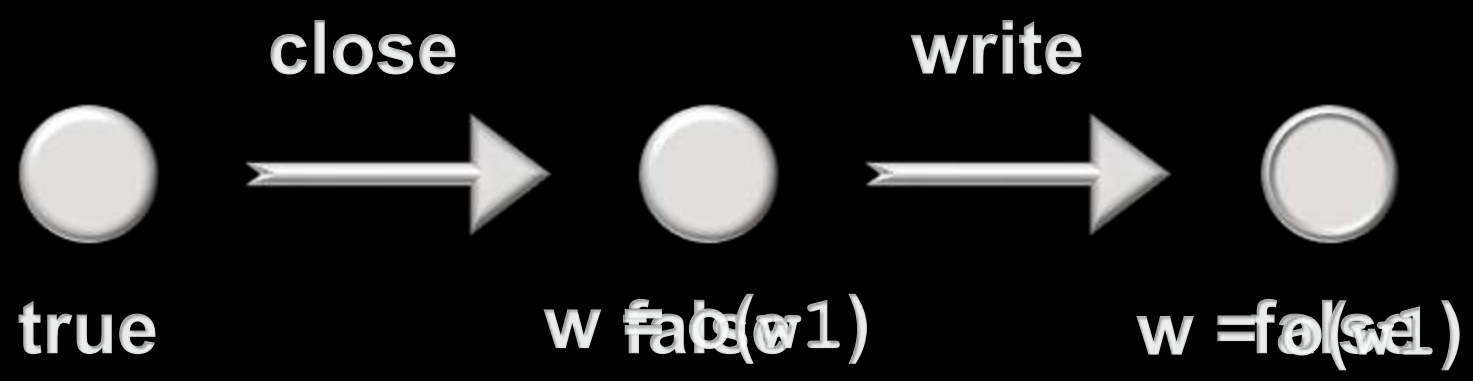
“no write after close”


```
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);
    }

}
```

How to specify API contracts?

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:

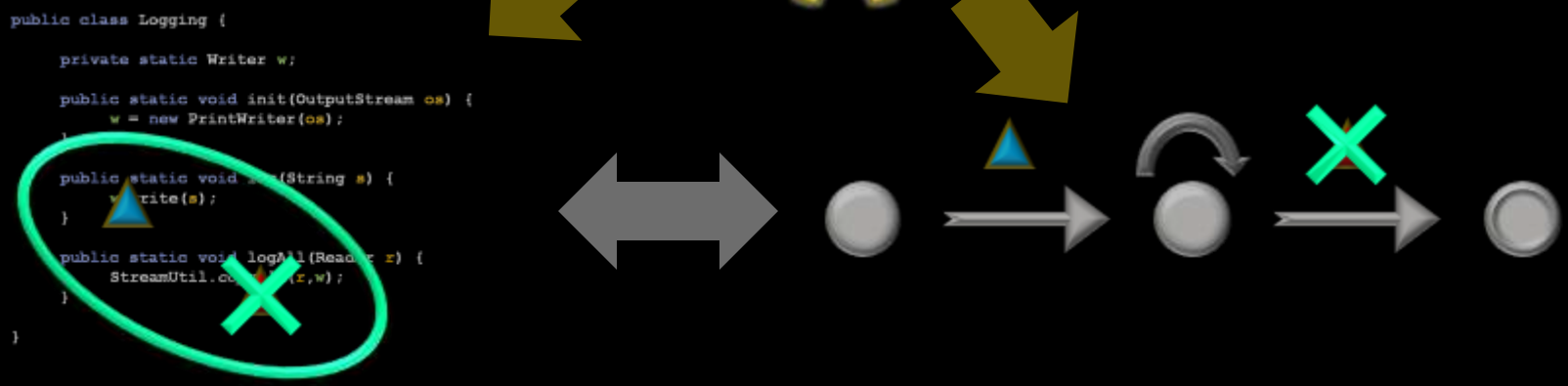
Potentially large runtime overhead

Problem 2:
No static guarantees

```
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”

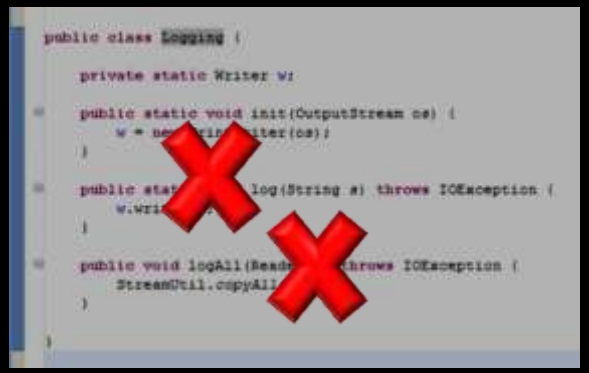
4 novel static program analyses

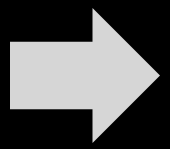


First static analysis:

“Quick Check”

close write





) = 0

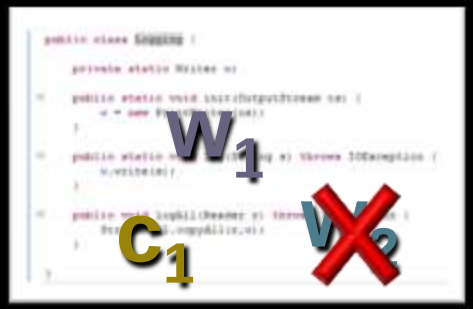
) = 2

Second static analysis:

“Consistent-shadows analysis”

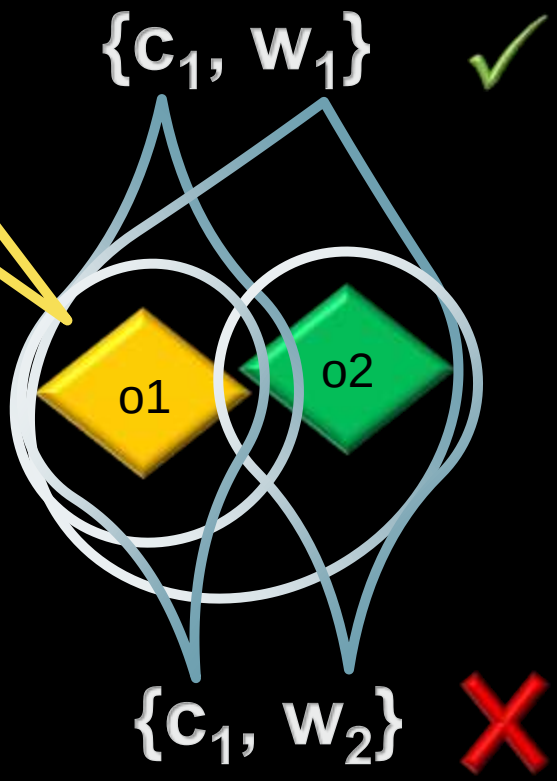
[close,write]

[close,write]



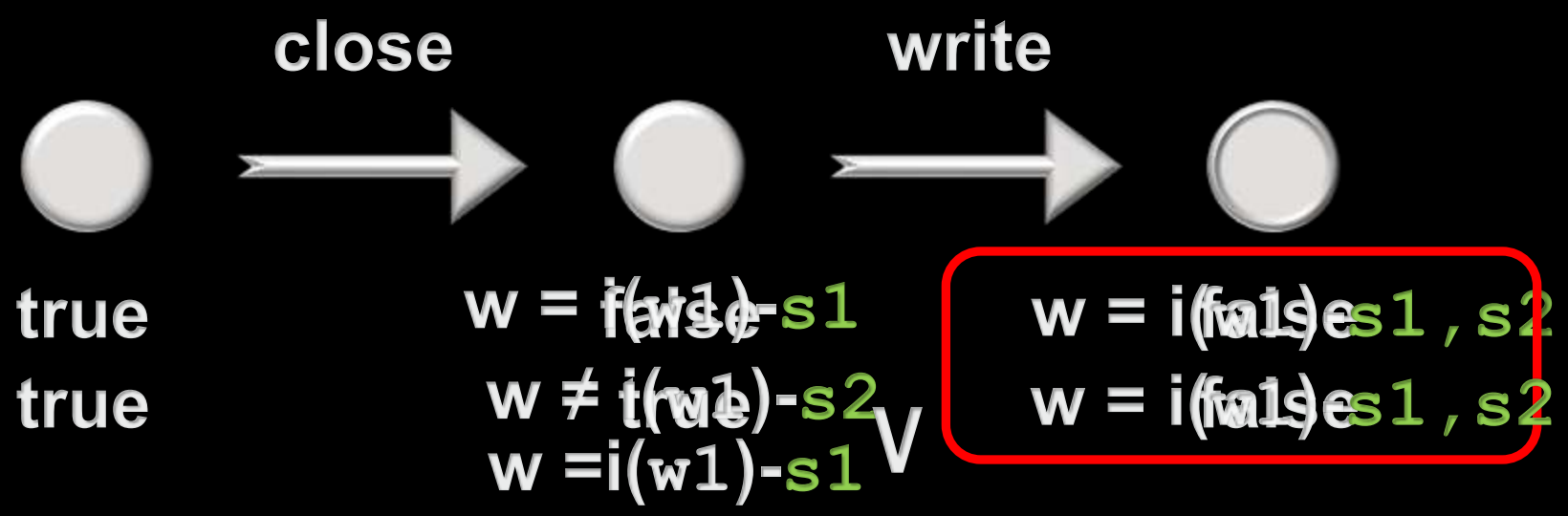
$\{c_1\} \times \{w_1, w_2\}$

```
new PrintWriter();
```



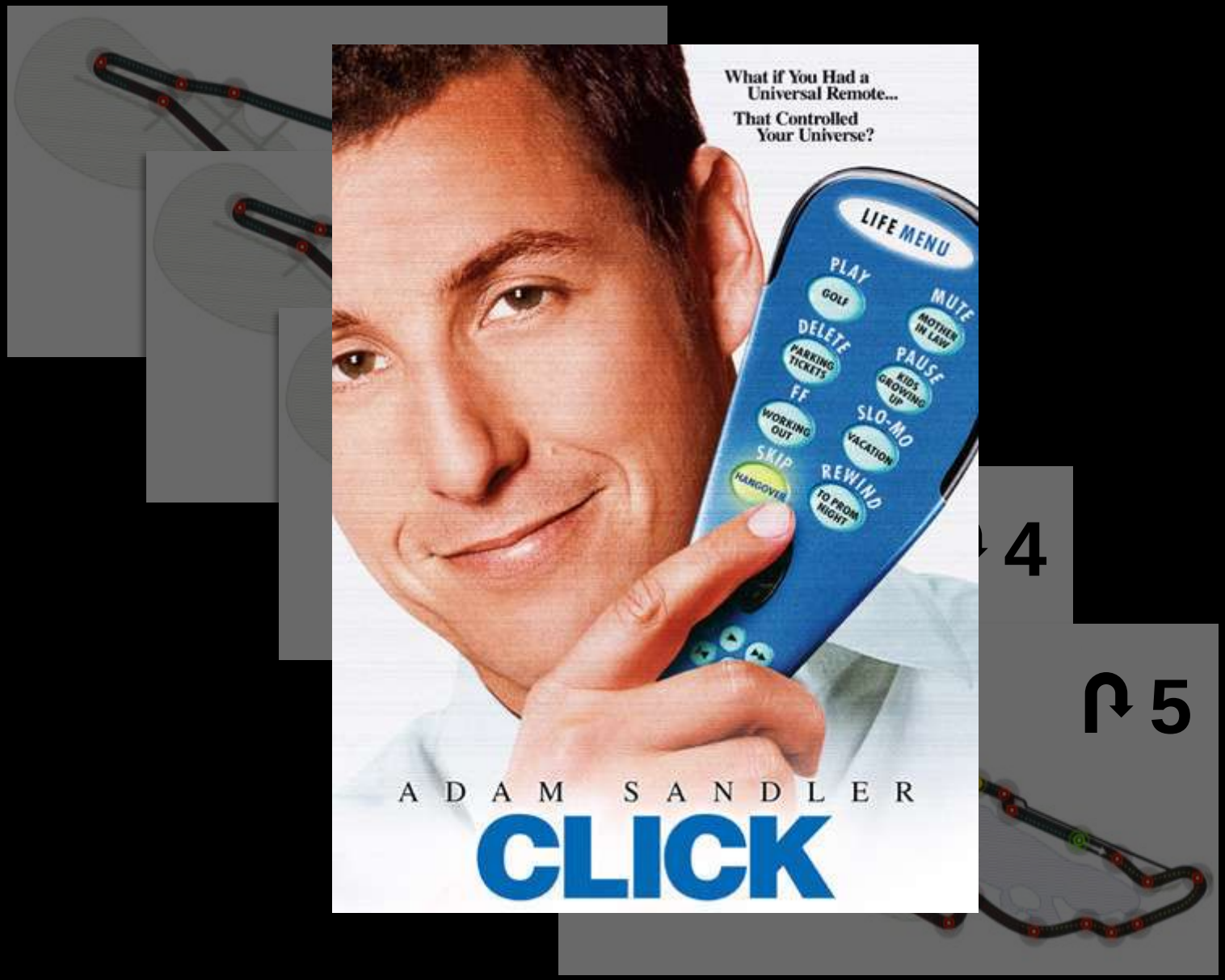
Third static analysis:

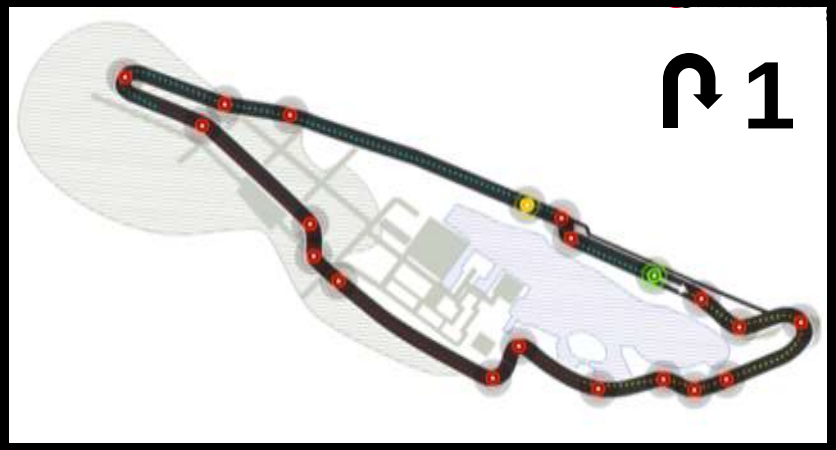
“Flow-sensitive unnecessary shadows analysis”

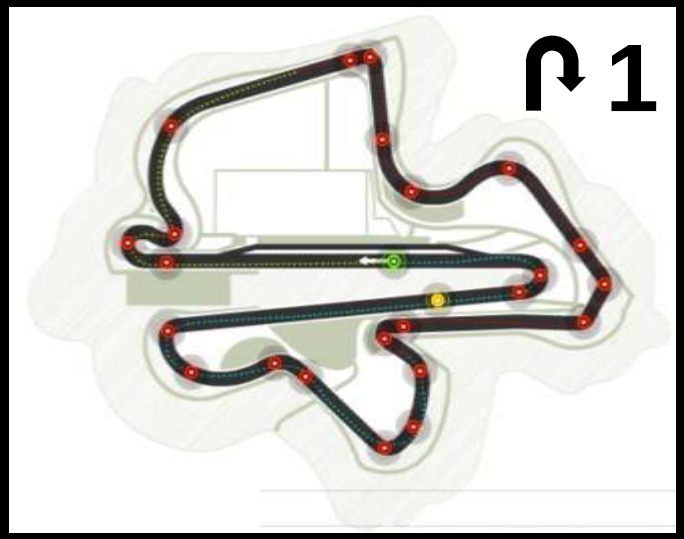


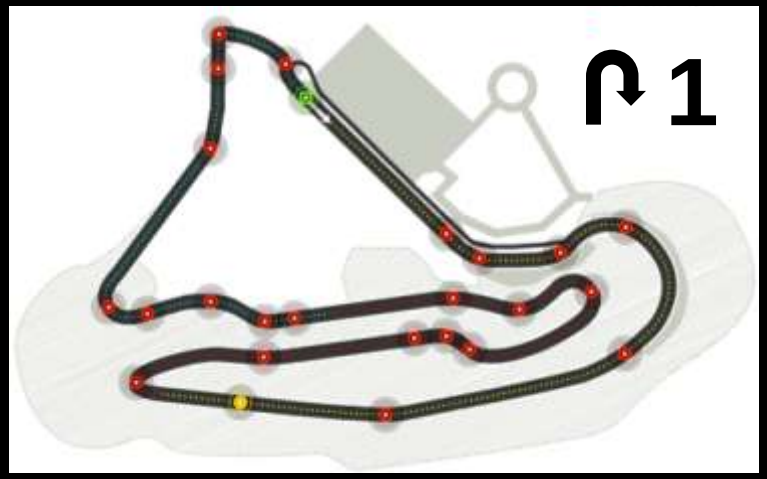

```
w1.close();          //s1

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

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

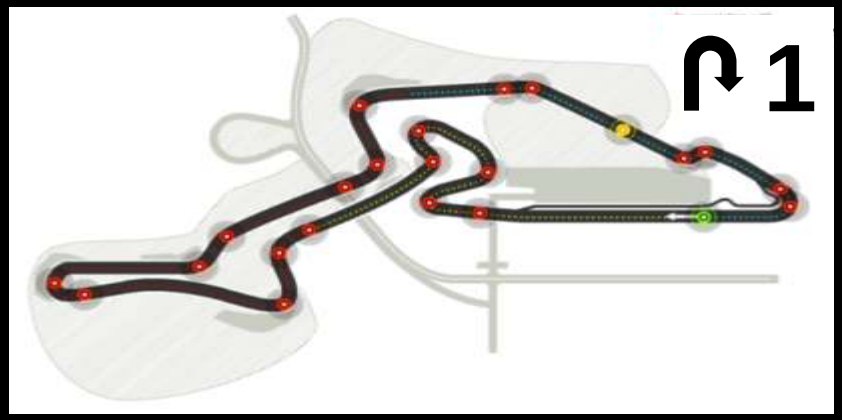
4th static analysis:
“run-once loop optimization”

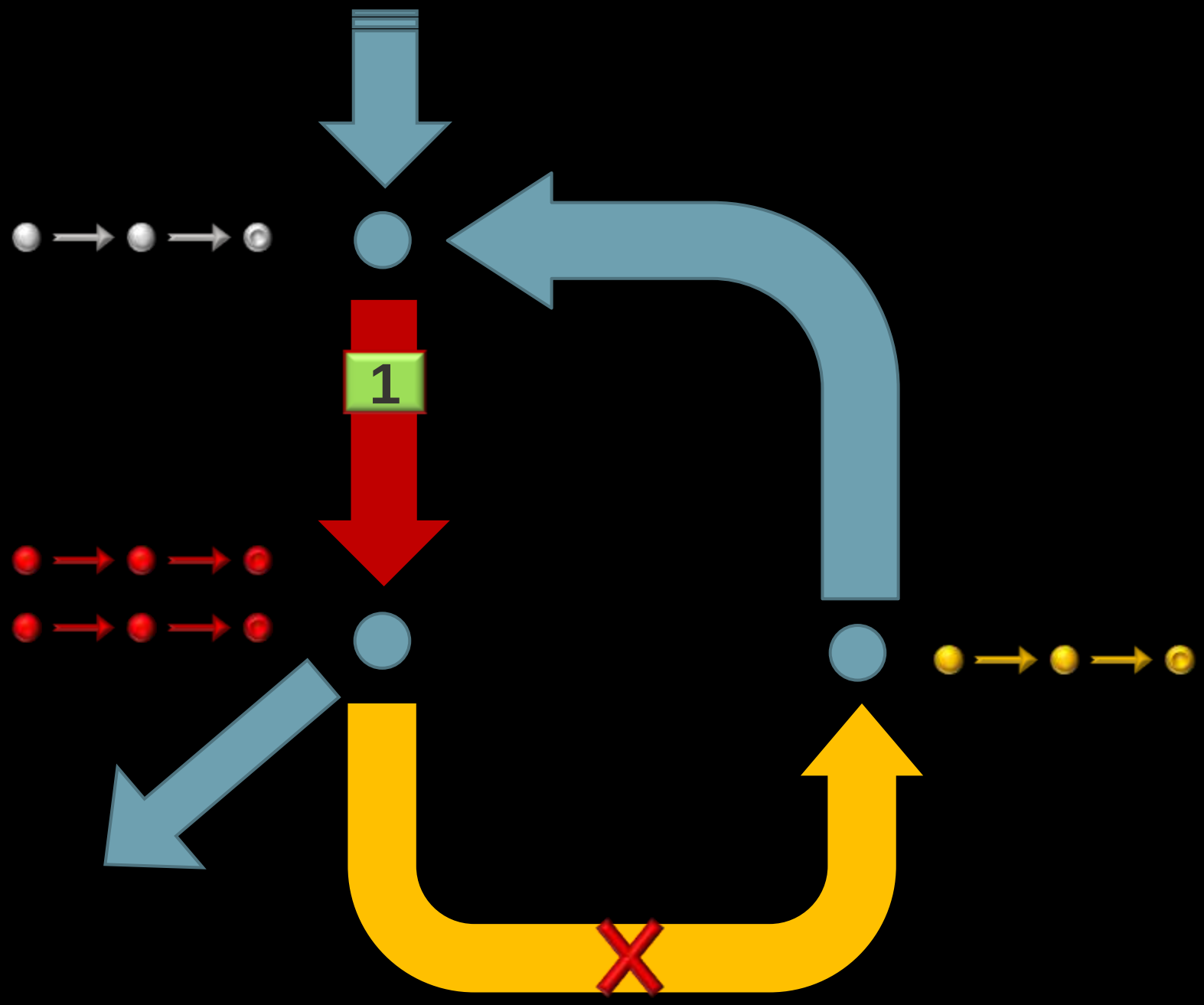

What if You Had a Universal Remote...
That Controlled Your Universe?
LIFE MENU
PLAY
GOLF
MUTE
MOTHER IN LAW
DELETE
PARKING TICKETS
PAUSE
KIDS GROWING UP
FF
WORKING OUT
SLO-MO
VACATION
SKIP
HANGOVER
REWIND
TO FROM NIGHT
ADAM SANDLER
CLICK
4
5


Ꞁ 1


Ꞁ 1


Ꞁ 1


Ꞁ 1


1

Results

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

102 program/tracematch combinations

static guarantees in 77 cases

Found 5 programs (out of 12) with bugs or questionable code.

less than 10 shadows in 12 cases

less than 10% overhead in 9 cases


621

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) {
        ...
}
```

pmd / HasNext (isEmpty())

```
protected NameDeclaration
      findVariableHere(NameOccurrence occurrence) {
      if (occurrence.isThisOrSuper() ||
          occurrence.getImage().equals(className)) {
            if (variableNames.isEmpty() &&
                  methodNames.isEmpty()) {
                  return null;
            }
            if (!variableNames.isEmpty()) {
                  return variableNames.keySet().
                        iterator().next();
            }
            return methodNames.keySet().
                  iterator().next();
      }
      ...
}
```