An Overview of the Indus Framework for Analysis and Slicing of Concurrent Java Software

SAnToS Laboratory, Kansas State University, USA

http://indus.projects.cis.ksu.edu

Indus & Kaveri

Venkatesh Ranganath
Ganeshan Jayaraman
John Hatcliff

Slicing for Model Checking

Matt Dwyer
John Hatcliff
Matt Hoosier
VenkateshRanganath
Robby
Todd Walletine

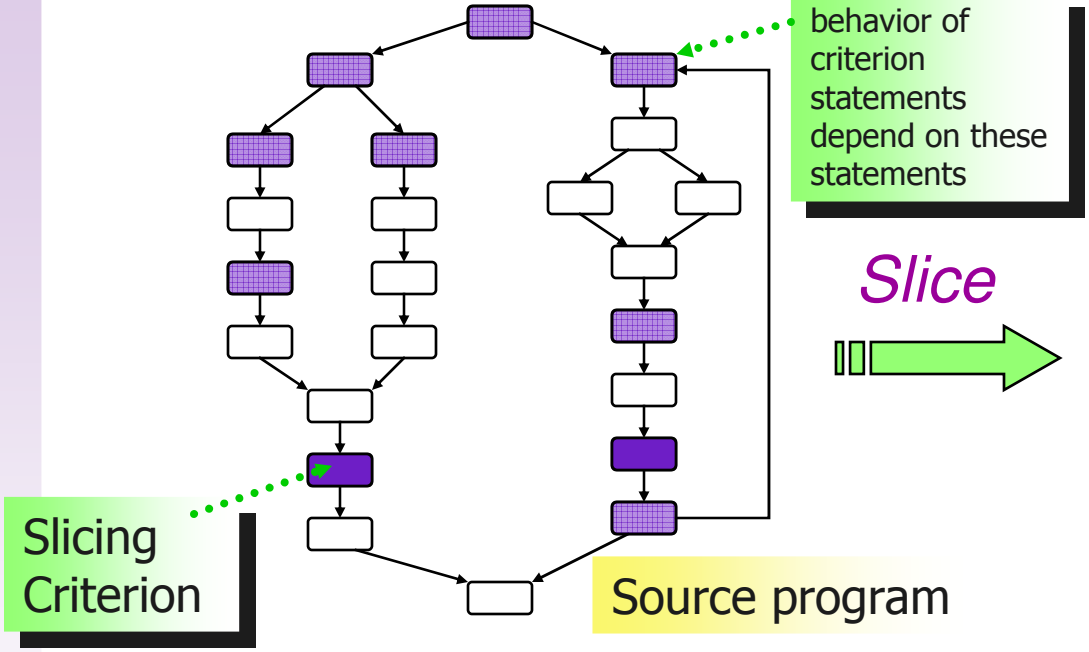
Support

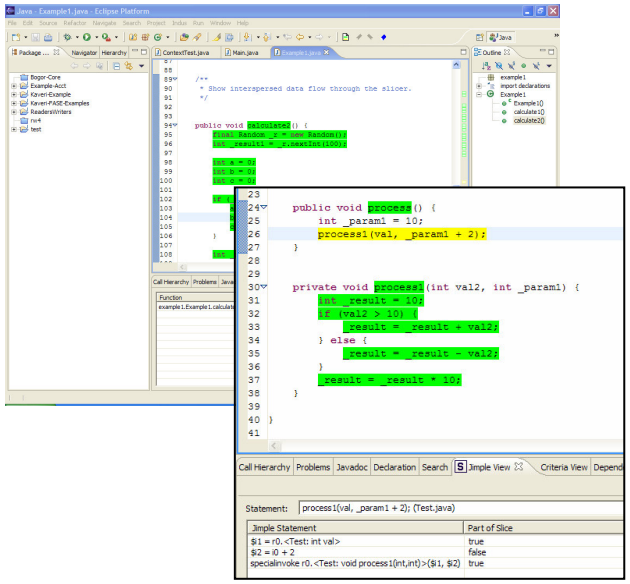
US Army Research Office
US Defense Advanced Research Projects Agency (DARPA)
Lockheed Martin
National Science Foundation

...thanks to Hongjun Zheng for building the initial prototype of our slicer!

Program Slicing

Static Backwards Slicing

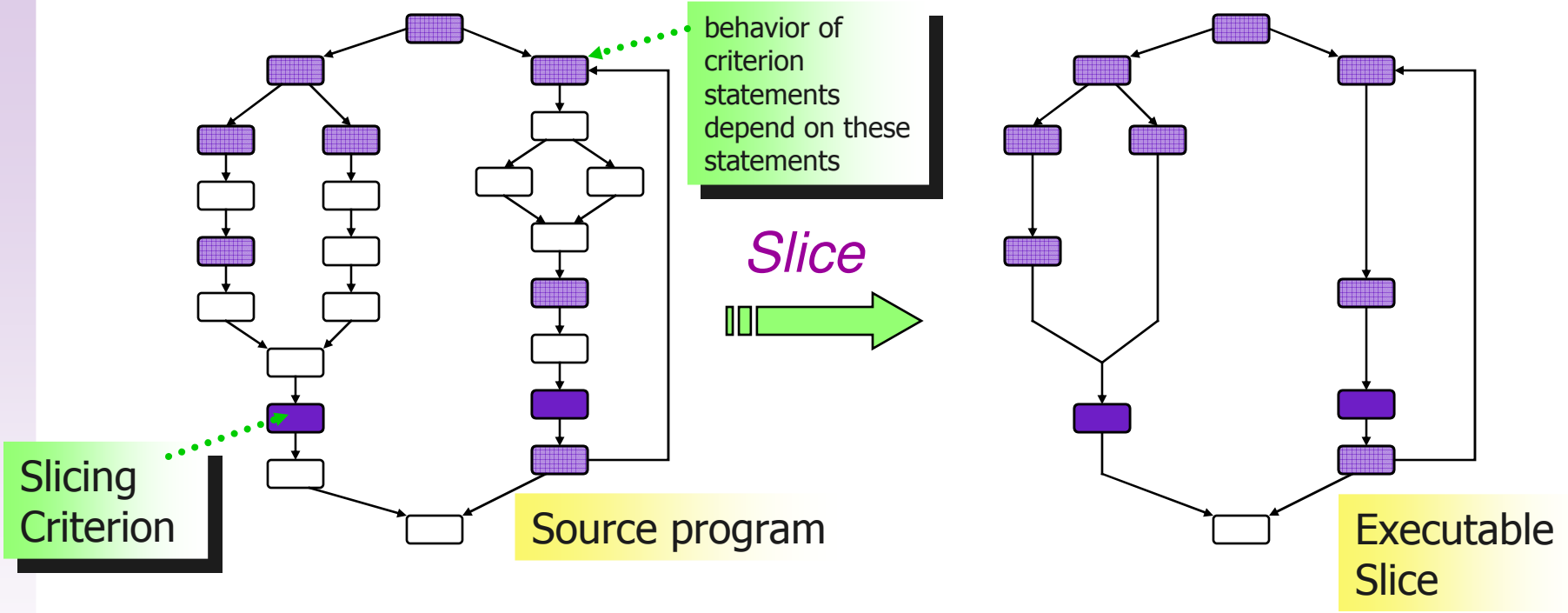




- slicing criterion statements are provided by the user
- backwards slicing automatically finds all statements that…
  - …the criterion statements depend on
  - …might influence the behavior of the slicing criterion

Program Slicing

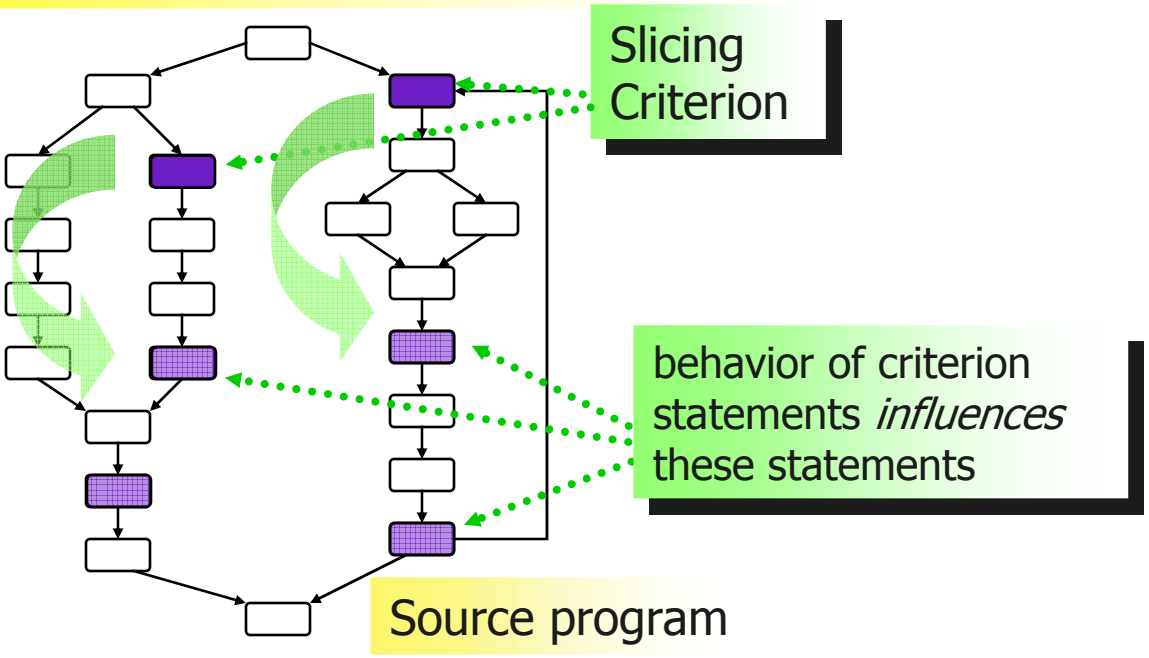
Static Backwards Slicing



- slicing criterion statements are provided by the user
- backwards slicing automatically finds all statements that…
  - …the criterion statements depend on
  - …might influence the behavior of the slicing criterion

Program Slicing

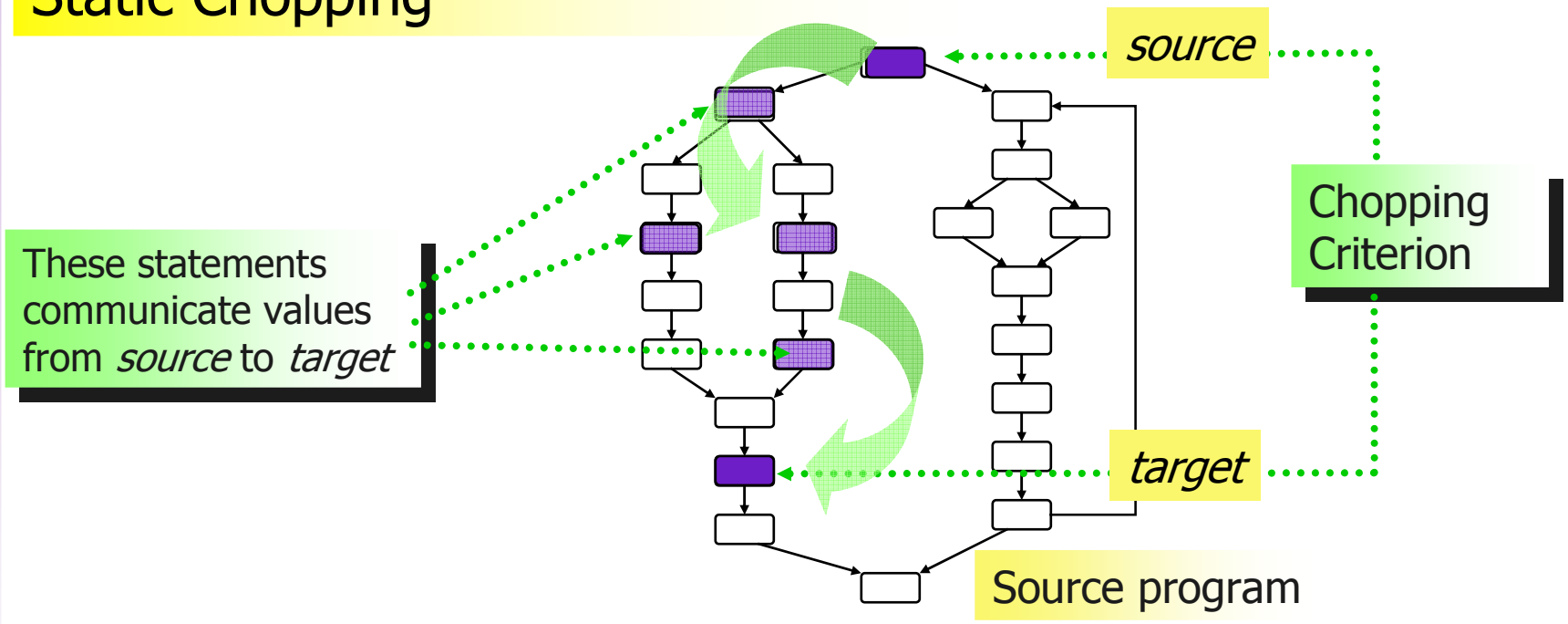
Static Forwards Slicing



- slicing criterion statements are provided by the user
- *forwards* slicing automatically finds all statements that…
  - …the criterion statements influence
  - …might depend on the behavior of the slicing criterion

Program Chopping

Static Chopping



- *source* and *target* chopping criterion statements are provided by the user
- *chopping* finds all statements that…
  - …contribute to control and data flows from the source to the target
- roughly equivalent to intersection of source forward slice and target backward slice

Applications

Program Understanding

If I change this statement, what pieces of the program are going to be affected?

```
class PowerOfTwo {
/**
 * A field that represent a power of two.
 */
protected static int powerOfTwo = 1;

/**
 * The main method.
 */
public static void main(String[] args) {

    //a counter that will give us the exponent of the powe
    int i = 0;

    //a loop to display the power of two.
    while (powerOfTwo < 32) {
        //print which power of two we are currently at.
        System.out.println("The current power of 2 is " +
        //compute the next power of two.
        Power(powerOfTwo);

                                    + " = " + powerOfTwo);
```

Forward slice with given statement as criteria

```
/*----------------------------------------------------------------------------
Prints a list of ODBC data sources and associated descriptions
----------------------------------------------------------------------------*/

                                    ODBCVER 0x0250                  // define ODBC API version 2.5
                                    OS2__
                                     "win2os2.h"

                                    sql.h>
                                    sqlext.h>

                                    in(int argc, char *argv[])

                                    HENV    henv;                   // environment handle
                                    RETCODE rc;                     // error return code
                                    UCHAR   dsName[SQL_MAX_DSN_LENGTH+1];   // name of the data sourc
                                    SWORD   dsNameLen;              // holds length of data
                                    UCHAR   dsDesc[257];            // description of data
                                    SWORD   dsDescLen;              // holds length of dat

                                    rc = SQLAllocEnv(&henv);        // allocate the ODB
                                    if (rc == SQL_ERROR)
                                    {
                                        printf("Error allocating environment\n");
                                        return -1;
                                    }

                                    //  fetch all the data sources defined on this system
                                    while (
                                            SQLDataSources(henv,SQL_FETCH_NEXT,dsName,
                                            (SWORD) (SQL_MAX_DSN_LENGTH), &dsNameLen, dsDesc,  256, &dsDes
                                            != SQL_NO_DATA_FOUND
                                        )
                                    {
                                        printf("DSN: %s  Desc: %s\n",dsName, dsDesc);
                                    }

                                    return 0;
}
```

Where are the values that flow into this statement coming from?

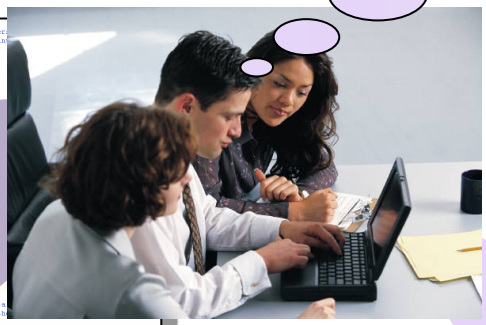
Backward slice with given statement as criteria

"Studies of software maintainers have shown that approximately 50% of their time is spent in the process of understanding the code that they are to maintain". Jussi Koskinen "Software Maintenance Costs"

Applications

Program Sub-setting

This code is bloated. How can we limit the functionality to only what we need?



Backward slice on desired modules to obtain desired subset of functionality

Applications

Impact Analysis

My regression tests take 20 hours to run. How can I limit the tests to only those that exercise the changed code?

relevant tests

change

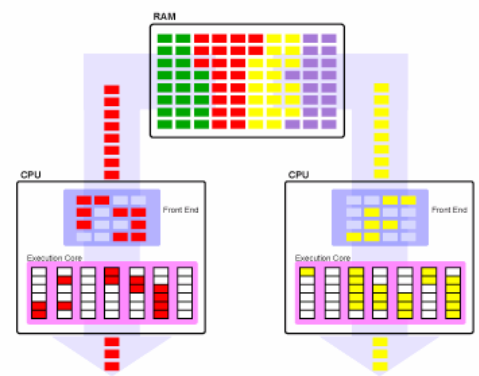
Backward slice with changed statements as criteria

Test suite

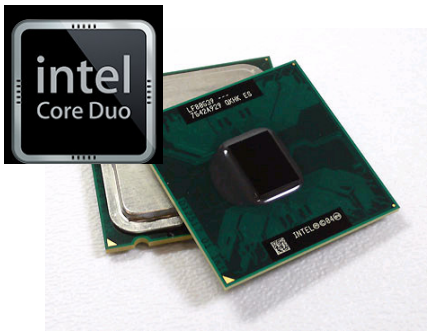
Current Trends

Slicing in a multi-threaded context?

- Significant increase in use of multi-threading due to languages with built-in threading primitives such as Java and C#



- Dramatic increase in use of concurrency in near future due to proliferation of multi-core processors



Slicing Multi-Threaded Programs

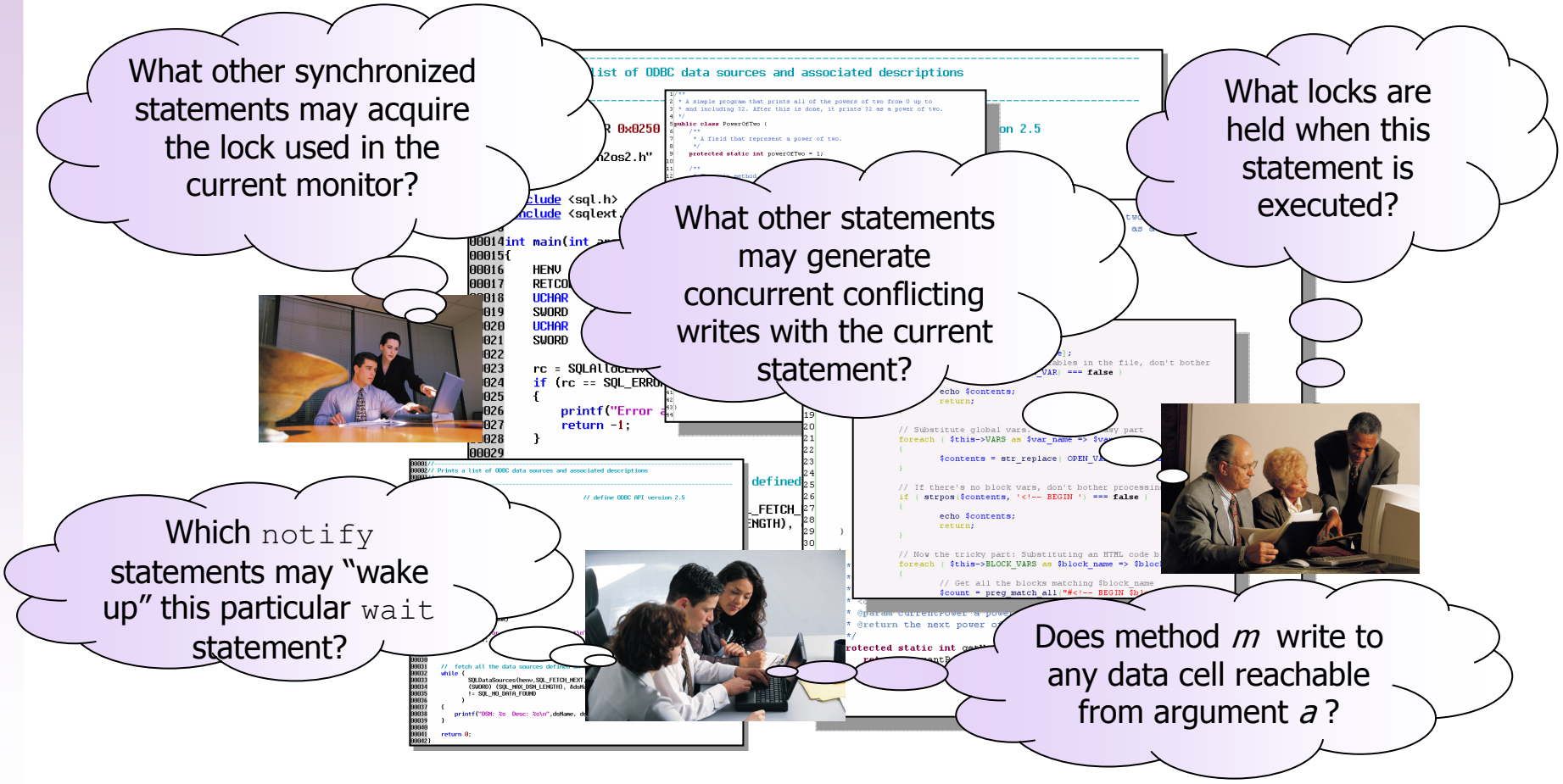
Questions…

- What are the basic notions of dependence required for multi-threaded Java programs?
- What sort of tool infrastructure is needed to calculate slices and present the results to developers?
- What optimizations and engineering decisions are necessary for scaling slicing?
- What are some non-conventional applications of slicing that arise in a multi-threaded setting?

Let's move beyond queries like: "what other statements does my current statement depend on?"

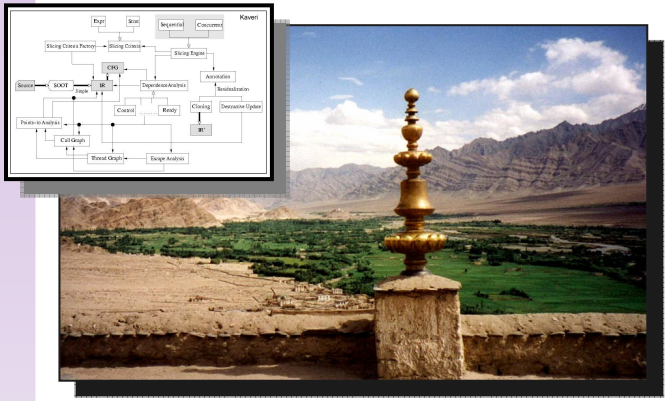
Beyond Traditional Slicing Questions

Mining the results of wide-ranging static analysis for concurrency



…we are interested in going well beyond than the usual slicing questions

Indus



Indus provides a rich collection of program analyses and transformations for Java programs (full Java)

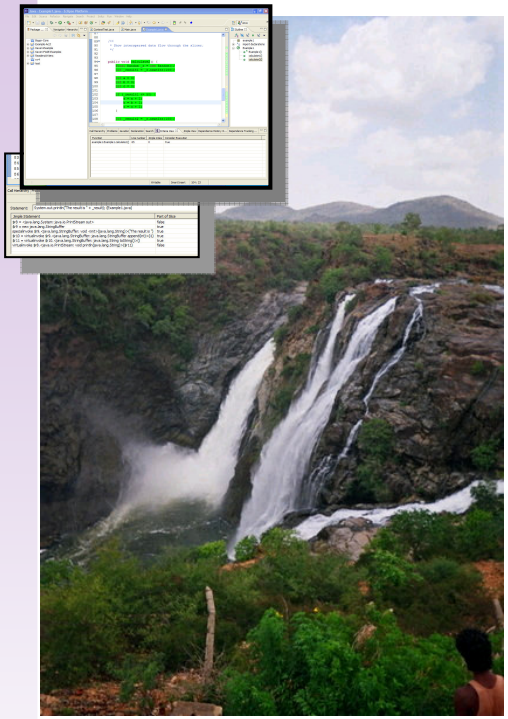
I. Platform for static analyses such as points-to analysis, escape analysis, and dependence analyses,

- … Object-flow Analysis (OFA)
- … Escape Analysis
- … Side-Effect Analysis
- … Monitor Analysis
- … Safe Lock Analysis
- … Dependence Analyses

II. Transformations such as program slicing and program specialization via partial evaluation

- … Backwards and Forwards Static Slicing
- … Generation of executable slices
- … Slice restriction using scope specifications and call paths
- … Property-aware slicing

Kaveri



Kaveri provides a featureful Eclipse-based UI front-end for Indus integrated with the Eclipse Java development environment

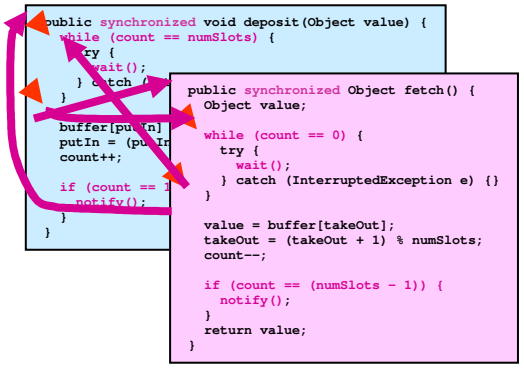
I. Configuring, invoking and visualizing results of Indus slicing

II. Scripting framework for flexible querying of underlying analysis APIs

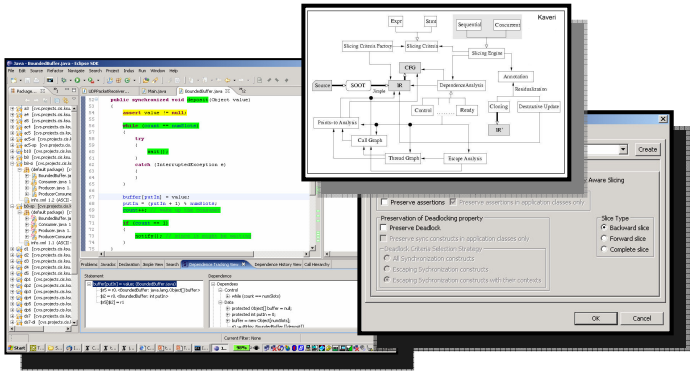
"According to Hindu tradition, the Kaveri river cleanses all sins."

Outline

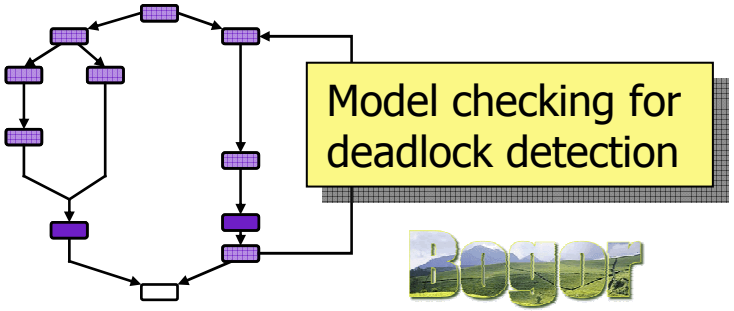
I. Program Dependences for Java



II. Indus & Kaveri



III. Application to Verifying Concurrent Java Programs



IV. Analysis optimization and mining analysis results

Dependence

Various notions of Dependences [Hatcliff et. al.:SAS99]

- Intra-thread data dependence
- Control dependence
- Interference dependence
- Synchronization dependence
- Divergence dependence
- Ready dependence

The usual stuff…

Dependences related to threading & locking…

Dependences for preserving liveness properties…

…dealing with all of these in the presence of, e.g., pointers, exceptions, etc.

Bounded Buffer Example

```
class BoundedBuffer {
  protected int numSlots = 0;          /* size of buffer */
  protected Object[] buffer = null;    /* buffer array */
  protected int putIn = 0;             /* next empty slot */
  protected int takeOut = 0;           /* next item */
  protected int count = 0;             /* number items */

  public BoundedBuffer(int numSlots)
    {…}

  public synchronized void deposit(Object value)
    {…}

  public synchronized Object fetch ()
    {…}
}
```

Data Dependence

```
public class BoundedBuffer {
  protected int numSlots = 0;
  protected Object[] buffer = null;
  protected int putIn = 0;
  protected int takeOut = 0;
  protected int count = 0;

  public BoundedBuffer(int numSlots) {
    if (numSlots <= 0) {
      throw new IllegalArgumentException();
    }

    this.numSlots = numSlots;
    buffer = new Object[numSlots];
  }

  ....
}
```

Data Dependence

Definition of variable V at statement S_1 reaches a use of V at statement S_2

…these statements depend on the data value assigned to parameter `numSlots`

Data Dependence

```
public class BoundedBuffer {
  protected int numSlots = 0;
  protected Object[] buffer = null;
  protected int putIn = 0;
  protected int takeOut = 0;
  protected int count = 0;

  public BoundedBuffer(int numSlots) {
    if (numSlots <= 0) {
      throw new IllegalArgumentException();
    }

    this.numSlots = numSlots;
    buffer = new Object[numSlots];
  }

  ....
}
```

Data Dependence

Definition of variable V in statement S_1 reaches a use of V at statement S_2

...these statements depend on the data value assigned to parameter `numSlots`

Data Dependence

```
public synchronized void deposit(Object value) {
  while (count == numSlots) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  buffer[putIn] = value;
  putIn = (putIn + 1) % numSlots;
  count++;

  if (count == 1) {
    notify();
  }
}
```

```
public synchronized Object fetch() {
  Object value;

  while (count == 0) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  value = buffer[takeOut];
  takeOut = (takeOut + 1) % numSlots;
  count--;

  if (count == (numSlots - 1)) {
    notify();
  }
  return value;
}
```

Data Dependence w/ Interprocedural Control Flow

Data Dependence

```
public synchronized void deposit(Object value) {
  while (count == numSlots) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  buffer[putIn] = value;
  putIn = (putIn + 1) % numSlots;
  count++;

  if (count == 1) {
    notify();
  }
}
```

```
public synchronized Object fetch() {
  Object value;

  while (count == 0) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  value = buffer[takeOut];
  takeOut = (takeOut + 1) % numSlots;
  count--;

  if (count == (numSlots - 1)) {
    notify();
  }
  return value;
}
```

Data Dependence w/ Interprocedural Control Flow

Data Dependence

```
public synchronized void deposit(Object value) {
  while (count == numSlots) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  buffer[putIn] = value;
  putIn = (putIn + 1) % numSlots;
  count++;

  if (count == 1) {
    notify();
  }
}
```

```
public synchronized Object fetch() {
  Object value;

  while (count == 0) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  value = buffer[takeOut];
  takeOut = (takeOut + 1) % numSlots;
  count--;

  if (count == (numSlots - 1)) {
    notify();
  }
  return value;
}
```

Data Dependence w/ Interprocedural Control Flow

Interference Dependence

```
public synchronized void deposit(Object value) {
  while (count == numSlots) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  buffer[putIn] = value;
  putIn = (putIn + 1) % numSlots;
  count++;

  if (count == 1) {
    notify();
  }
}
```

```
public synchronized Object fetch() {
  Object value;

  while (count == 0) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  value = buffer[takeOut];
  takeOut = (takeOut + 1) % numSlots;
  count--;

  if (count == (numSlots - 1)) {
    notify();
  }
  return value;
}
```

Data Dependence across threads

Control Dependence

```
public class BoundedBuffer {
  protected int numSlots = 0;
  protected Object[] buffer = null;
  protected int putIn = 0;
  protected int takeOut = 0;
  protected int count = 0;

  public BoundedBuffer(int numSlots) {
    if (numSlots <= 0) {
      throw new IllegalArgumentException();
    }

    this.numSlots = numSlots;
    buffer = new Object[numSlots];
  }

  ....
}
```

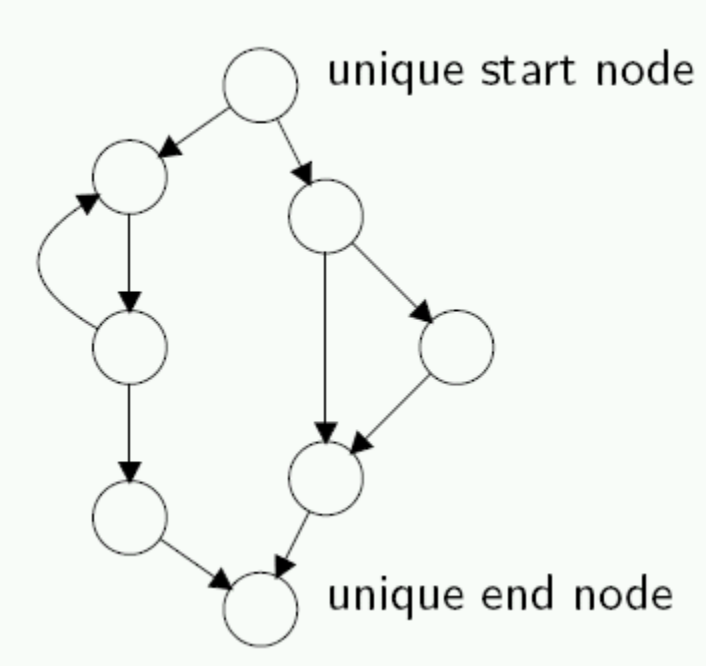
Control Dependence

A conditional statement controls whether or not the current statement is executed

...this statement depends on the conditional which controls whether it is reached or bypassed by throwing an exception

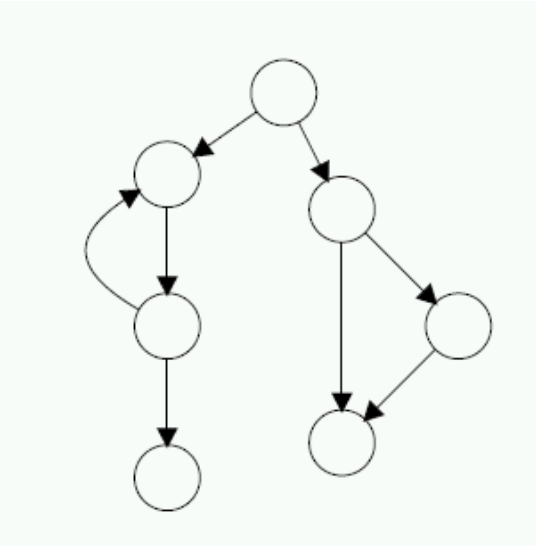
Enhanced Control Dependence

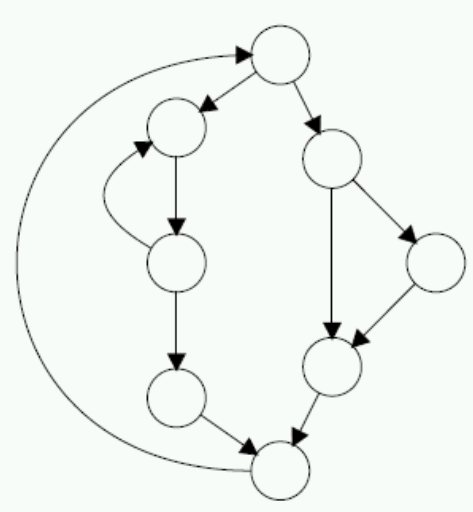
Conventional definitions of control dependence are presented in terms of CFGs with unique start and end nodes



Enhanced Control Dependence

…but CFGs of Java often violate this constraint





Multiple end nodes

- multiple return nodes in a method
- exceptional exits from a method

No end node

- reactive programs (designed to run indefinitely)
- event loops in GUI frameworks, etc.

Enhanced Control Dependence

Indus uses an enhanced notion of control dependence that…

- handles CFGs with multiple ends and no end nodes
- works with both reducible (structured programs) and irreducible (unstructured programs) CFGs
- has formal correctness properties established using a notion of weak bi-simulation for both finite and infinite program traces

Ranganath et. al., ESOP 2005

Divergence Dependence

```
public synchronized void deposit(Object value) {
  while (count == numSlots) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  buffer[putIn] = value;
  putIn = (putIn + 1) % numSlots;
  count++;

  if (count == 1) {
    notify();
  }
}
```

…capturing delaying influences within a single thread

```
public synchronized Object fetch() {
  Object value;

  while (count == 0) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  value = buffer[takeOut];
  takeOut = (takeOut + 1) % numSlots;
  count--;

  if (count == (numSlots - 1)) {
    notify();
  }
  return value;
}
```

Divergence Dependence

Statement S_1 influences S_2 if S_1 may infinitely delay S_2 (e.g., by going into an infinite loop)

Ready Dependence

```
public synchronized void deposit(Object value) {
  while (count == numSlots) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  buffer[putIn] = value;
  putIn = (putIn + 1) % numSlots;
  count++;

  if (count == 1) {
    notify();
  }
}
```

```
public synchronized Object fetch() {
  Object value;

  while (count == 0) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  value = buffer[takeOut];
  takeOut = (takeOut + 1) % numSlots;
  count--;

  if (count == (numSlots - 1)) {
    notify();
  }
  return value;
}
```

...capturing delaying influences across threads

Ready Dependence

The failure of a lock release to complete may cause a lock acquire to be delayed (blocked) indefinitely

Ready Dependence

```
public synchronized void deposit(Object value) {
  while (count == numSlots) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  buffer[putIn] = value;
  putIn = (putIn + 1) % numSlots;
  count++;

  if (count == 1) {
    notify();
  }
}
```

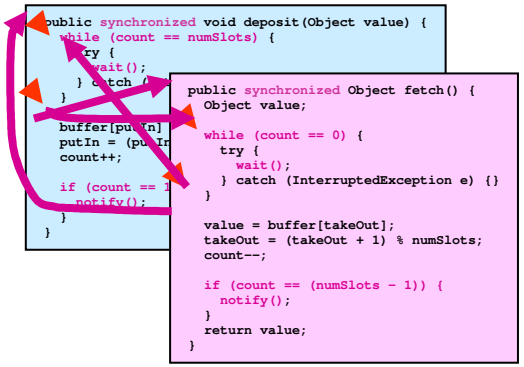
```
public synchronized Object fetch() {
  Object value;

  while (count == 0) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }

  value = buffer[takeOut];
  takeOut = (takeOut + 1) % numSlots;
  count--;

  if (count == (numSlots - 1)) {
    notify();
  }
  return value;
}
```

Ready Dependence

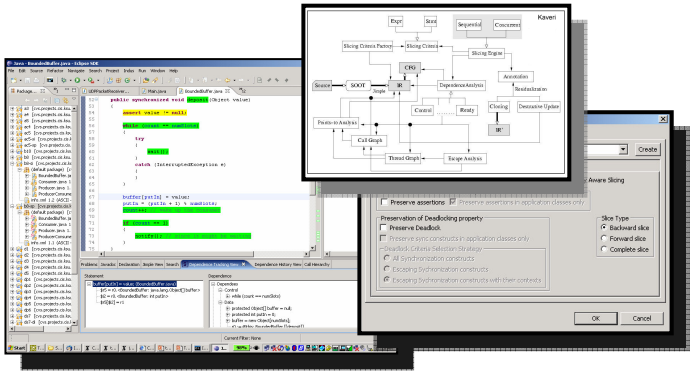
The failure of a `notify` to complete may cause a `wait` to be delayed indefinitely

Outline

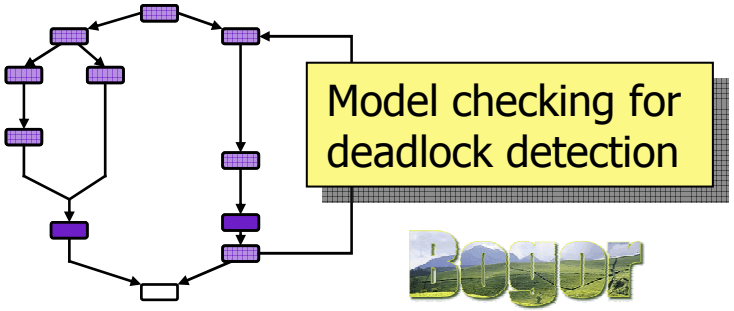
I. Program Dependences for Java



II. Indus & Kaveri

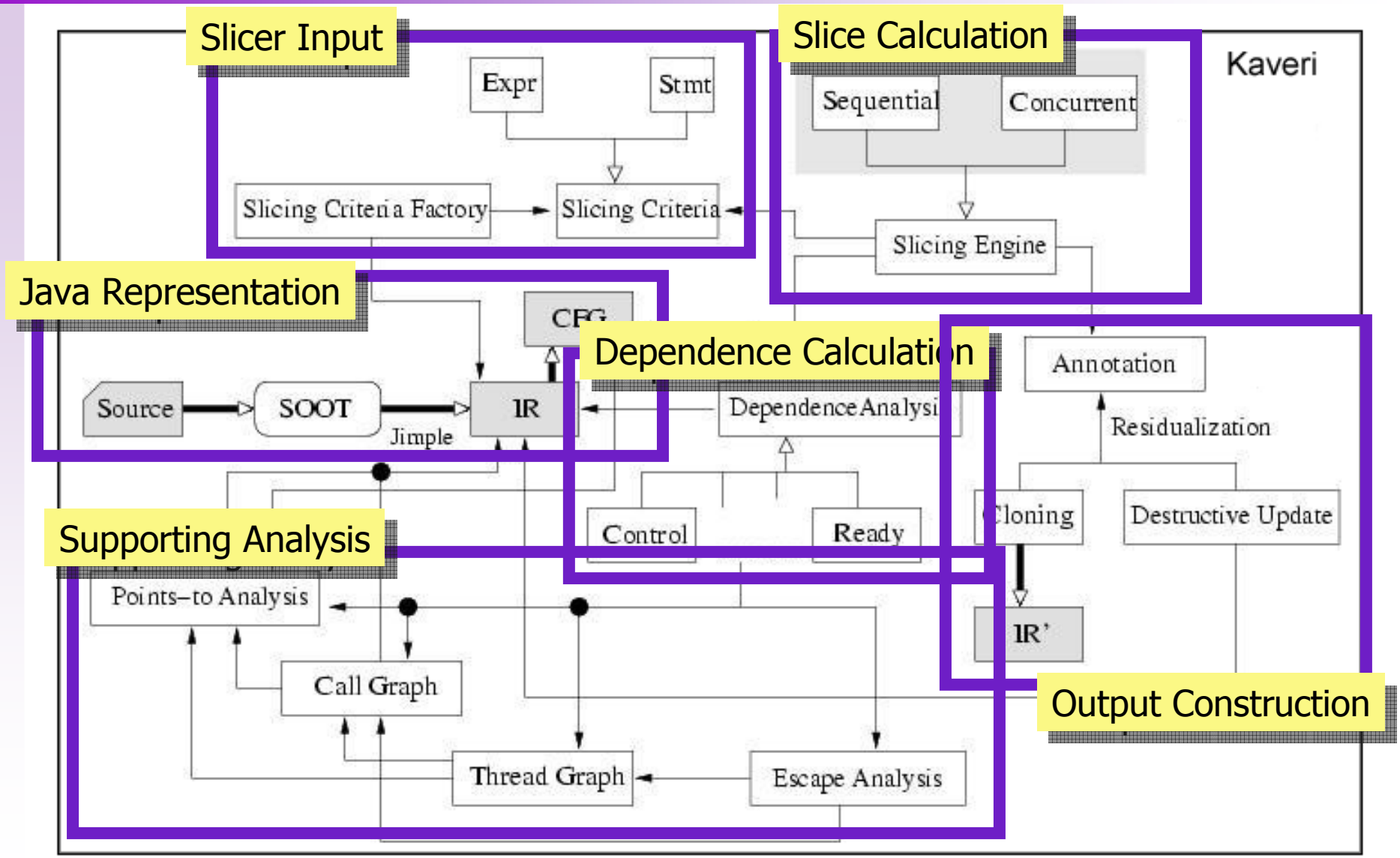


III. Application to Verifying Concurrent Java Programs

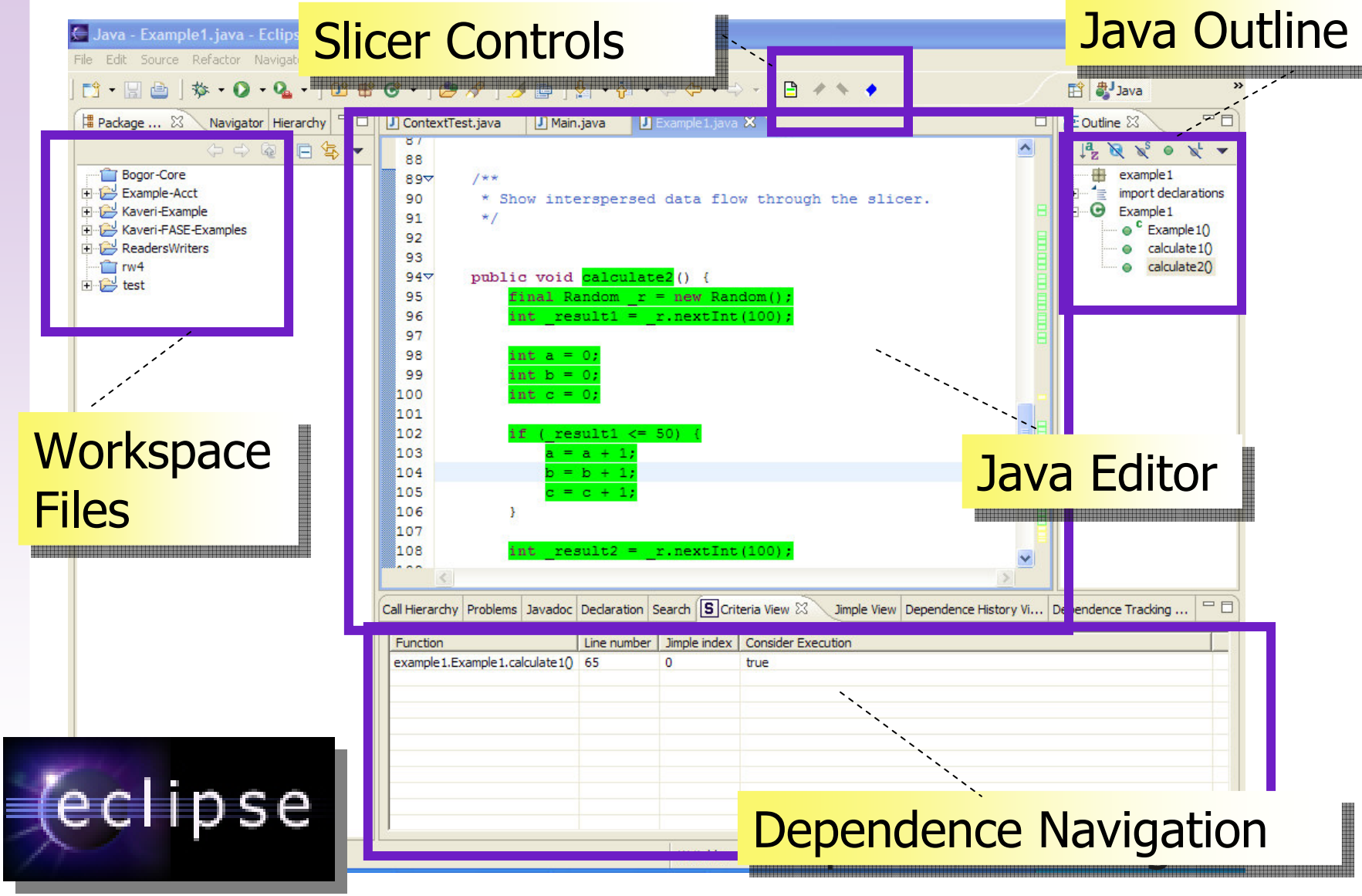


IV. Analysis optimization and mining analysis results

Indus / Kaveri Architecture

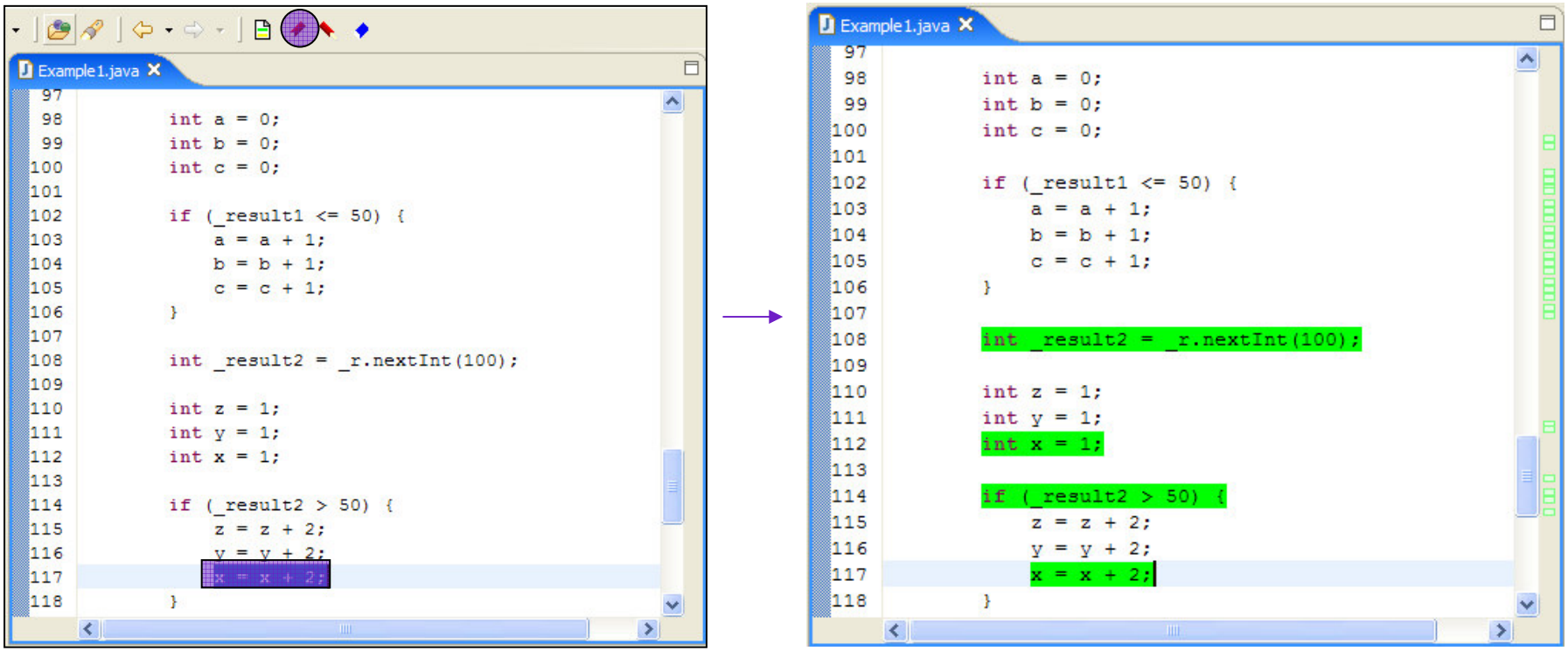


Theme: Java Development Integration



Basic Use Mode

Launch Backward Slice for single statement criteria

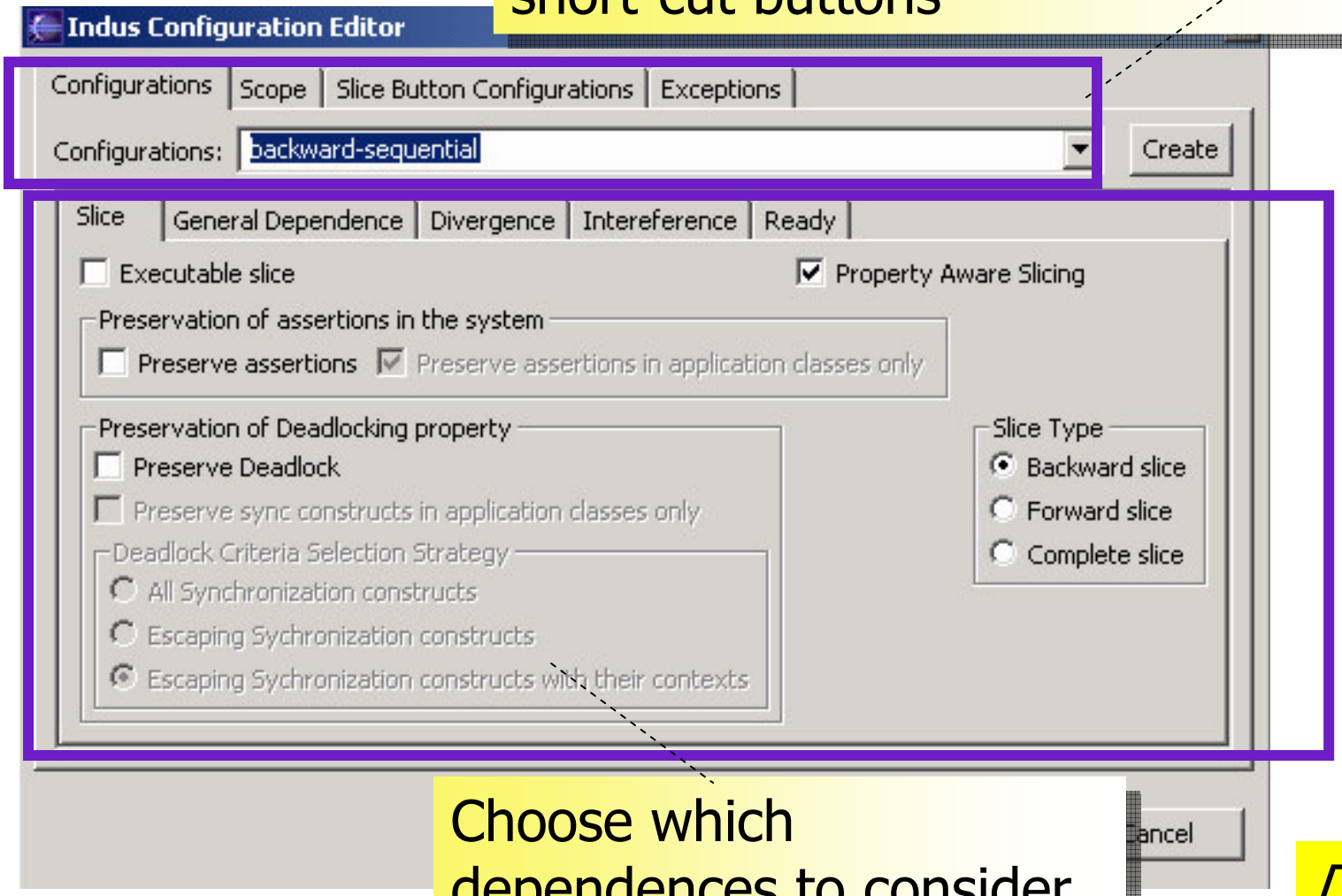


- A single statement is used as the criteria

Demo

Demo: Slicer Configuration

Name/Save configurations and bind to short-cut buttons



Choose which dependences to consider

Demo

Source Code / Byte Code Issues

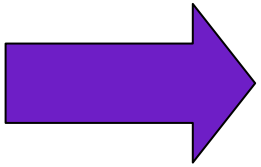
Understanding the corresponding between Java source and byte code representation (Jimple)

- We would like to enable developers to work at the Java source level…
- …yet in many cases the semantics of Java and dependences of Java can only be explained precisely by working at the byte code level

Jimple Representation of Java

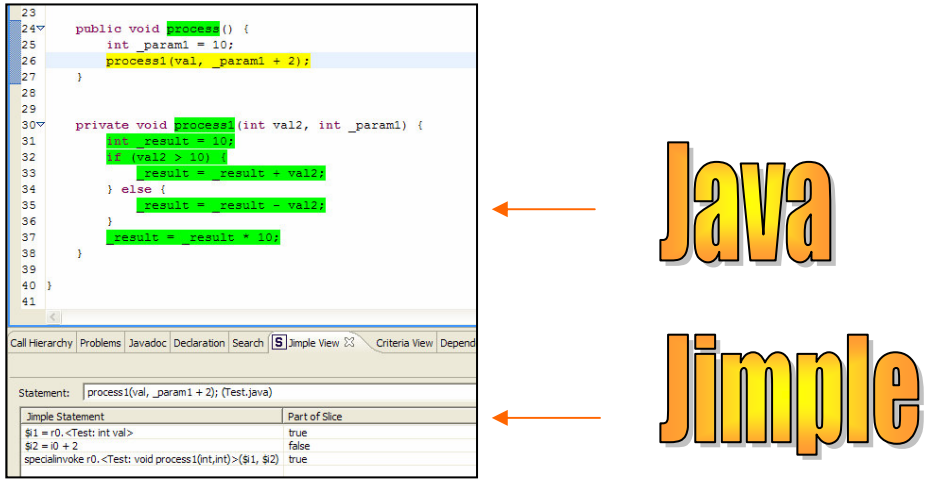
Jimple is an intermediate representation of Java

if (!empty(Clist))



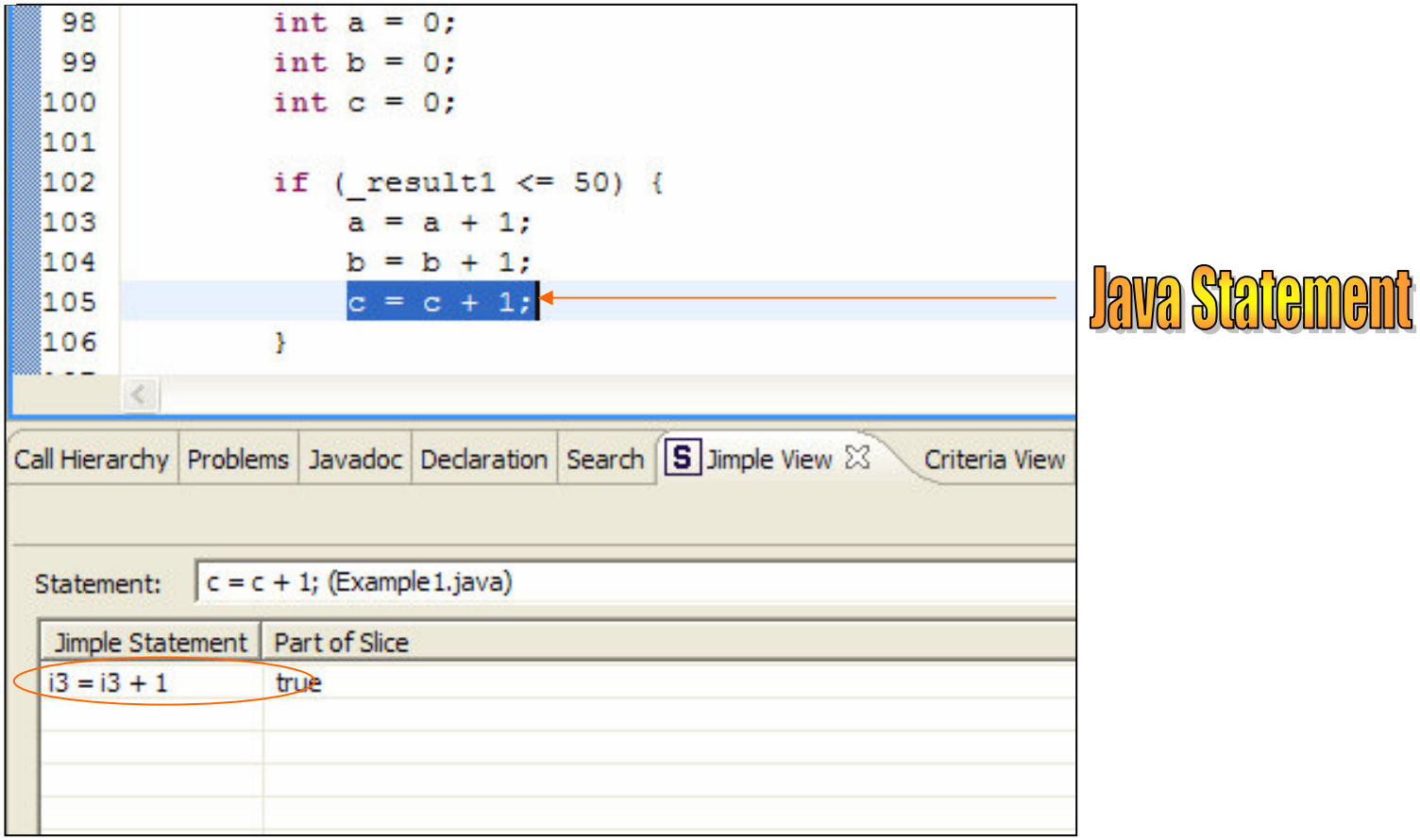
- **$r4 = r0.<temp.DiskScheduler:java.util.LinkedList Clist>**
- **$z0 = virtualinvoke r0.<temp.DiskScheduler: boolean empty(java.util.LinkedList)>($r4)**
- **if $z0 != 0 goto $r7 = r0.<temp.DiskScheduler:java.util.LinkedList NList>**

Kaveri provides a special view to see the Jimple statements that correspond to a particular Java statement



Jimple Representation of Java

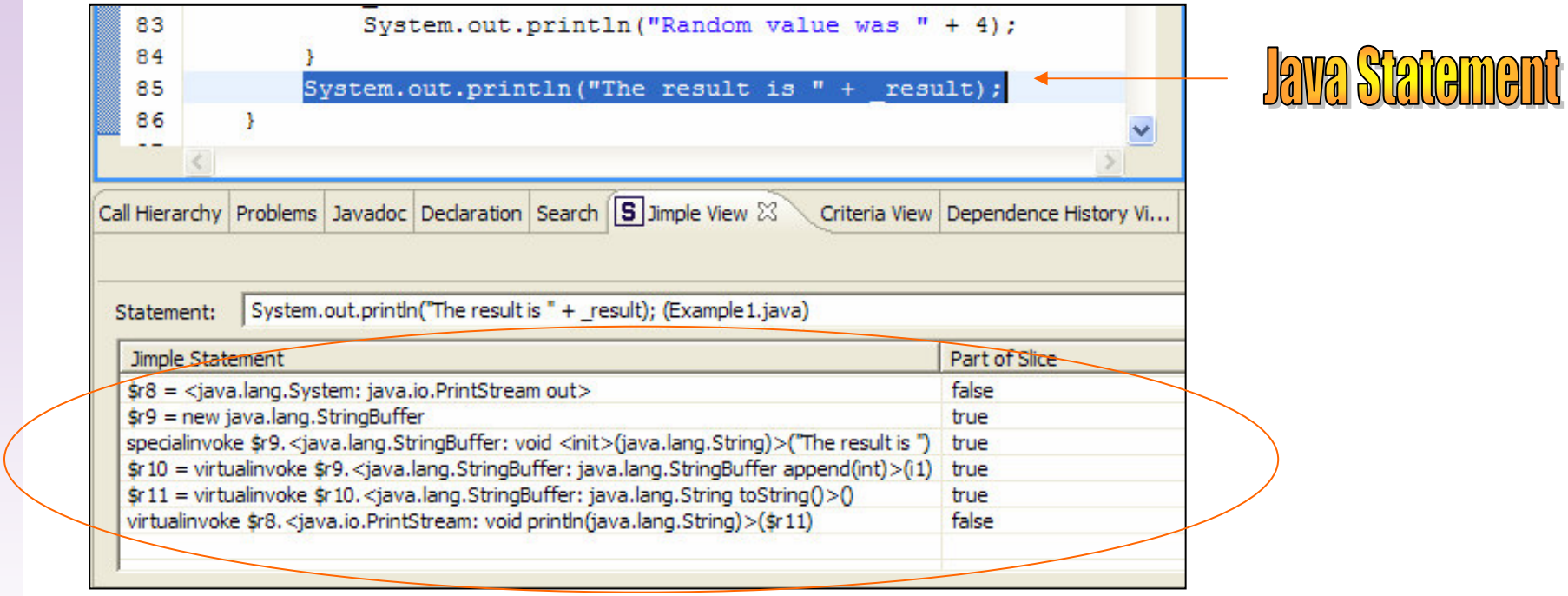
Example



- Some Java statements correspond to a single Jimple statement

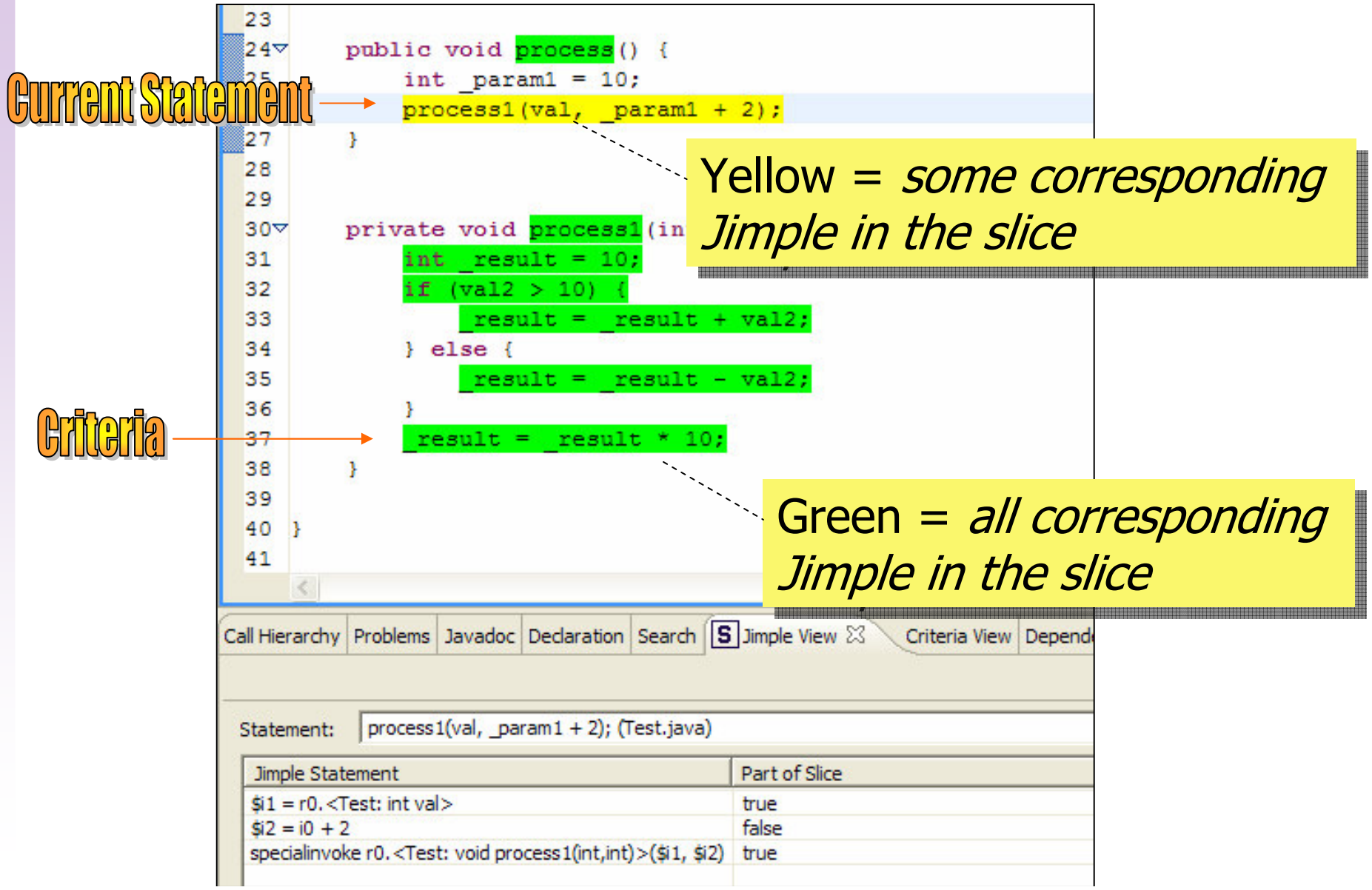
Jimple Representation of Java

Example

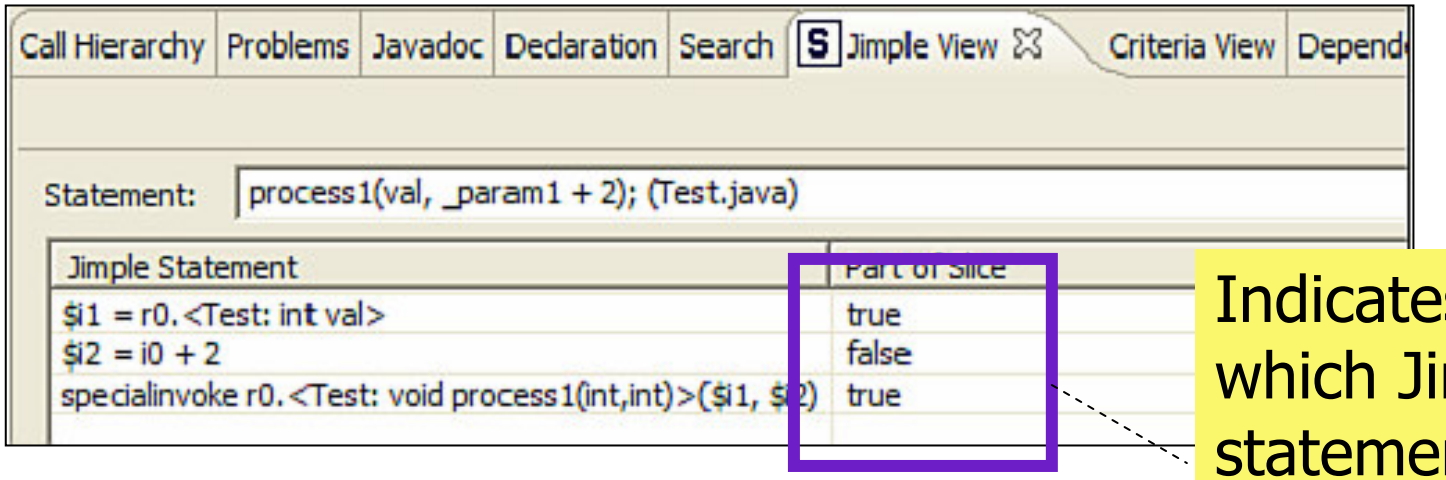


- Some Java statements correspond to a several Java statements
- The example shows the underlying use of *StringBuffer* to perform String concatenation

Jimple View


Current Statement
Criteria
23
24 public void process() {
25 int _param1 = 10;
process1(val, _param1 + 2);
27 }
28
29
30 private void process1(in
31 int _result = 10;
32 if (val2 > 10) {
33 _result = _result + val2;
34 } else {
35 _result = _result - val2;
36 }
37 _result = _result * 10;
38 }
39
40 }
41
Yellow = some corresponding Jimple in the slice
Green = all corresponding Jimple in the slice
Call Hierarchy
Problems
Javadoc
Declaration
Search
Jimple View
Criteria View
Statement:
process1(val, _param1 + 2); (Test.java)
Jimple Statement
Part of Slice
$i1 = r0.<Test: int val>
true
$i2 = i0 + 2
false
specialinvoke r0.<Test: void process1(int,int)>($i1, $i2)
true

Jimple View



Indicates which Jimple statements are in the slice

- The call to method *process1* has two parameters: **val**, **_param1 + 2**
- Only **val** is used in the method *process1*
- Hence, only the parameter **val** has the slice tag

Demo

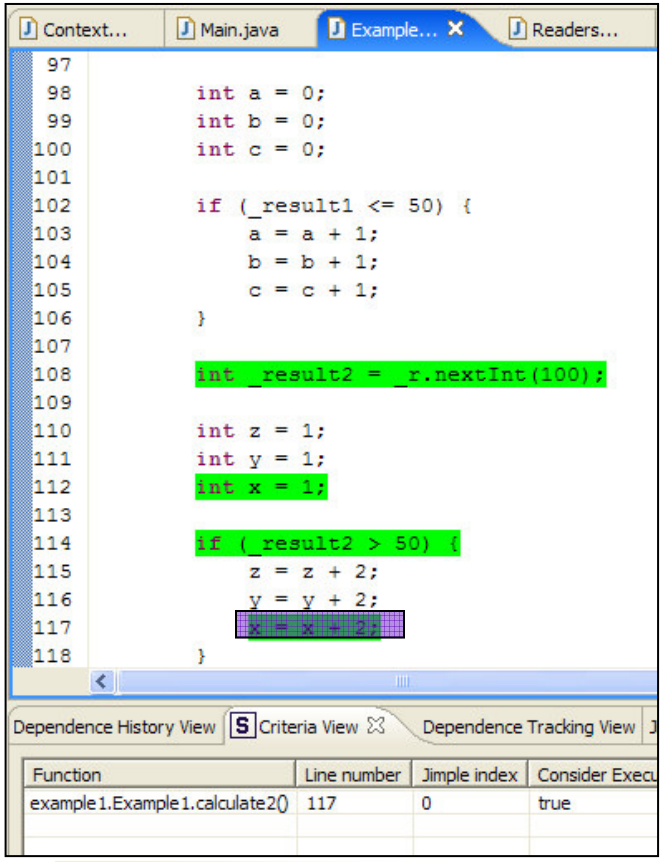
Now Let's Slice

Setting the Slice Criterion

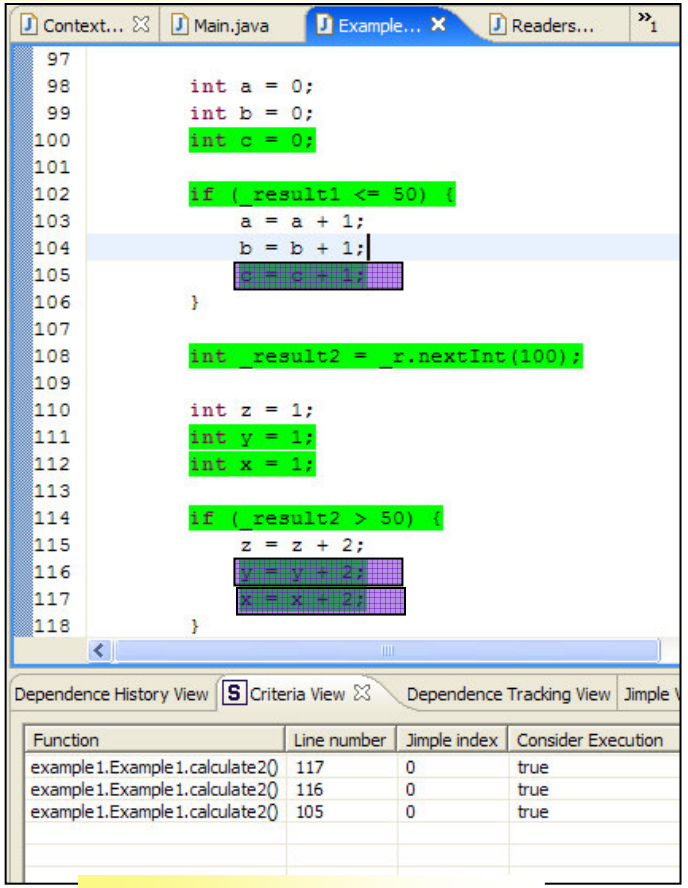
- A *Slicing Criterion* indicates a point of interest for slicing
  - Represented by a Jimple statement in Kaveri
  - The slicing criteria for a slice can be:
    - …one Jimple statement
    - …multiple Jimple statements

Criteria List

Slicing Criteria can be either a *single* Jimple statement or *multiple* Jimple statements



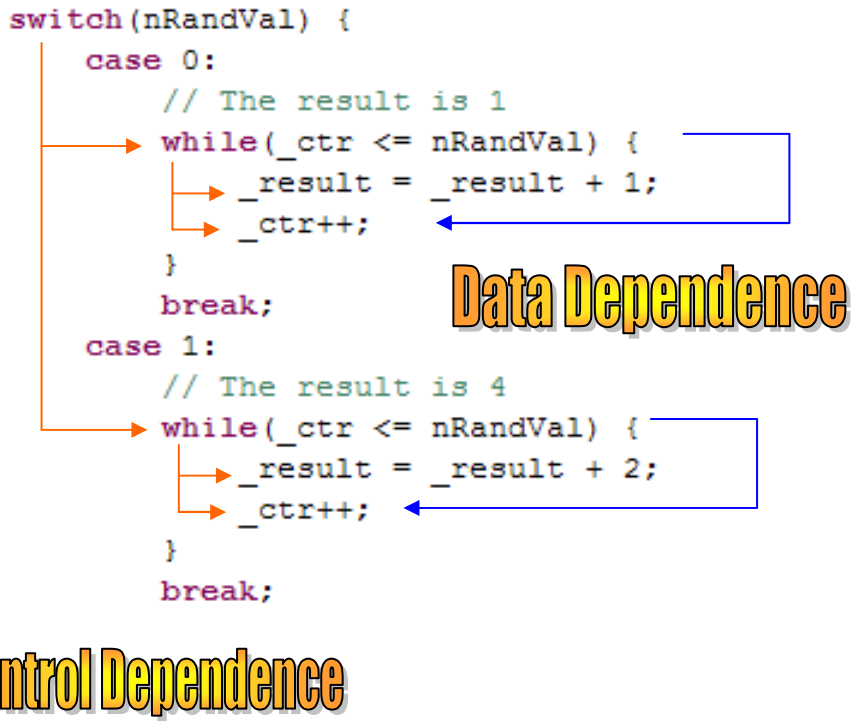
Single statement



Multiple statements

Kaveri Dependence Navigation

We have a bunch of "green stuff" – how did it get there?



Kaveri provides support for filtering and navigating dependence graphs

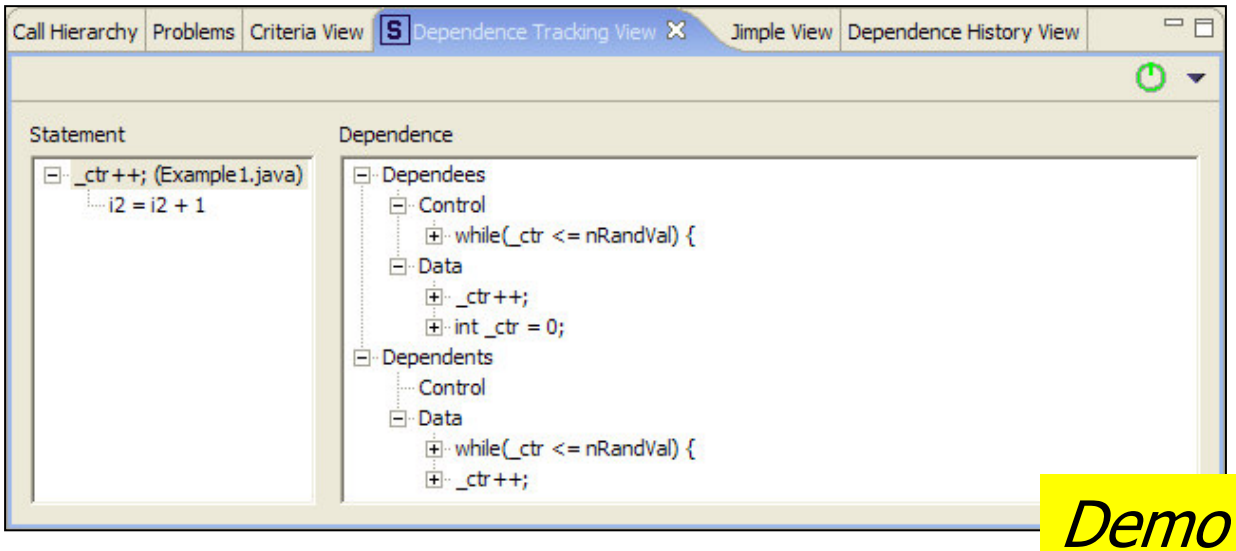
Even simple code can have a lot of dependencies

Dependence Tracking View

```
switch(nRandVal) {
    case 0:
        // The result is 1
        while(_ctr <= nRandVal) {
            _result = _result + 1;
            _
        }
        break;
    case 1:
        // The result is 4
        while(_ctr <= nRandVal) {
            _result = _result + 2;
            _ctr++;
        }
        break;
```

Data Dependence

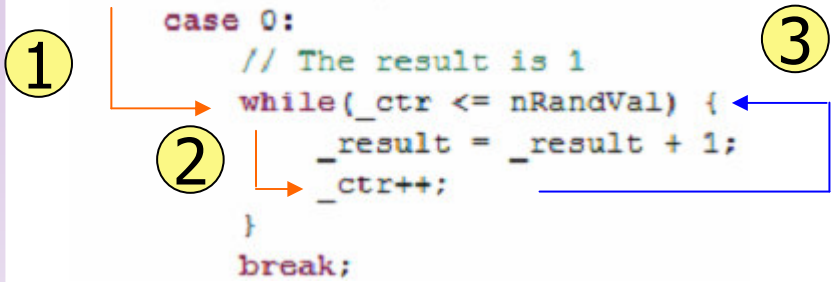
Control Dependence

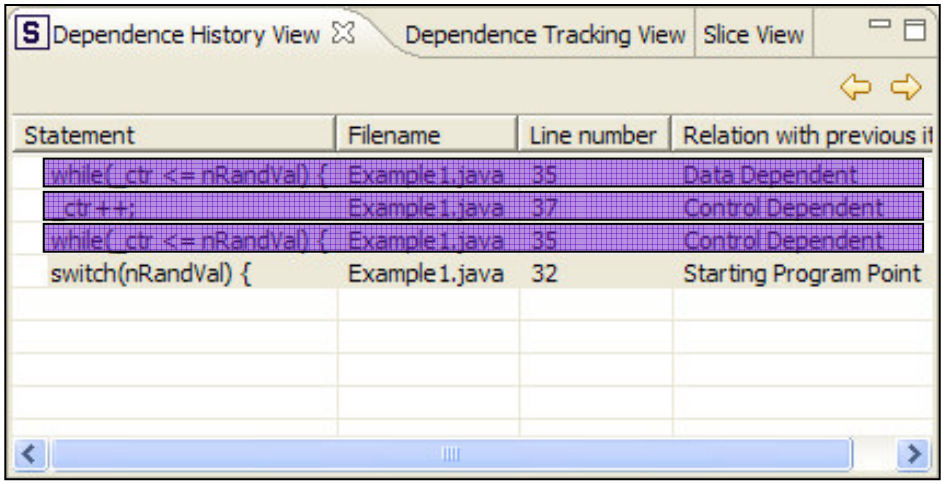


Demo

Dependence History View

```
switch(nRandVal) {
    case 0:
        // The result is 1
        while(_ctr <= nRandVal) {
            _result = _result + 1;
            _ctr++;
        }
        break;
```



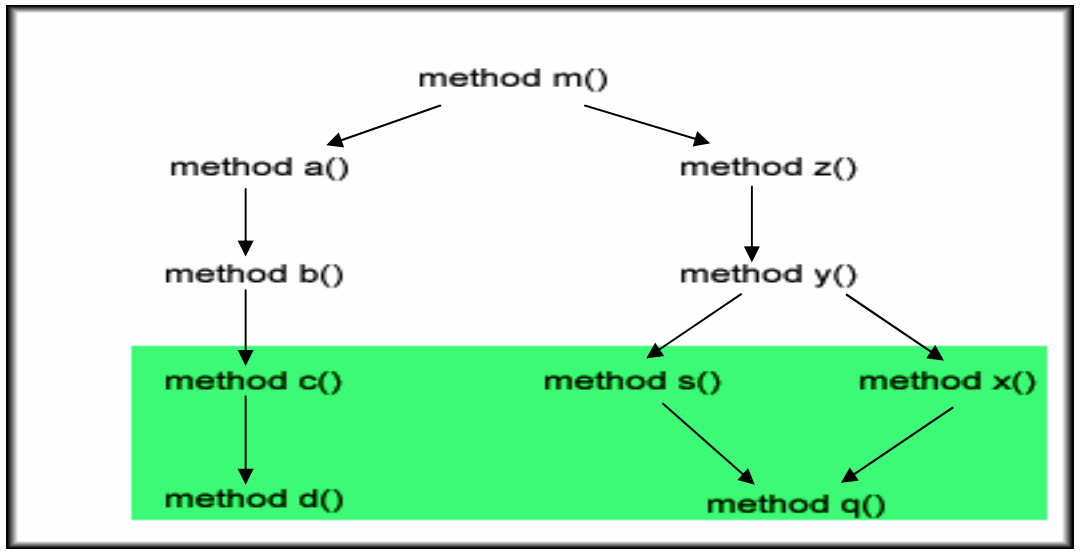


- Maintains a history of program points visited through dependence tracking
- Navigate to earlier points in the path

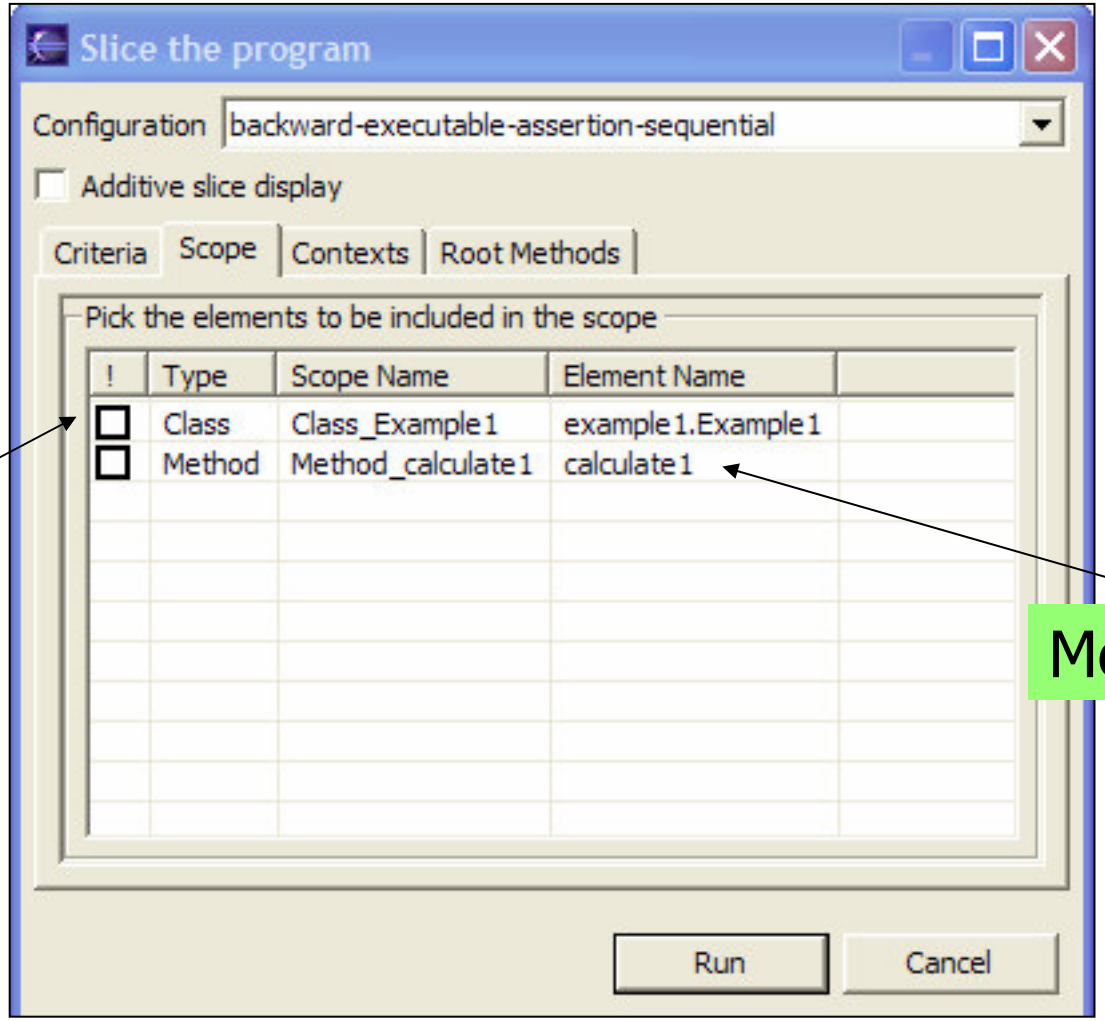
Demo

Scoped Slicing

- Scoped slicing allows the user to restrict entities that are considered for the slice
- Entities that can be added to the scope include Classes and Methods



Scoped Slicing



Class Scope

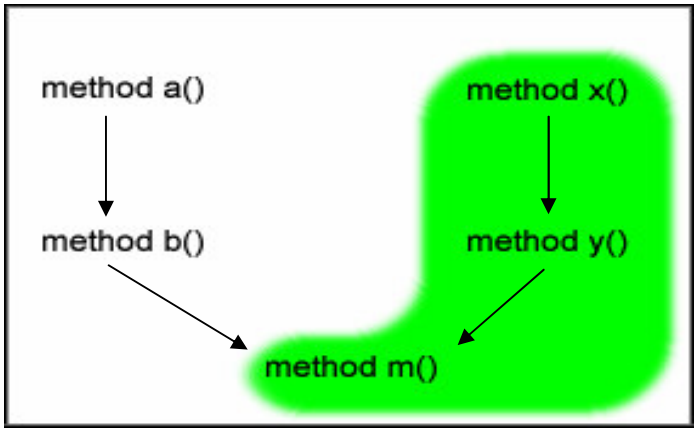
Method Scope

Demo

Call graph context-bounded slice

Imagine that your program has just crashed and you have a stack dump show the call chain that lead to the error

- You could slice at the error point to help you identify the coding flaw
  - …but that will show dependence throughout the program (including other call paths that arrived at the error point).
- Indus/Kaveri provides context-bounded slicing

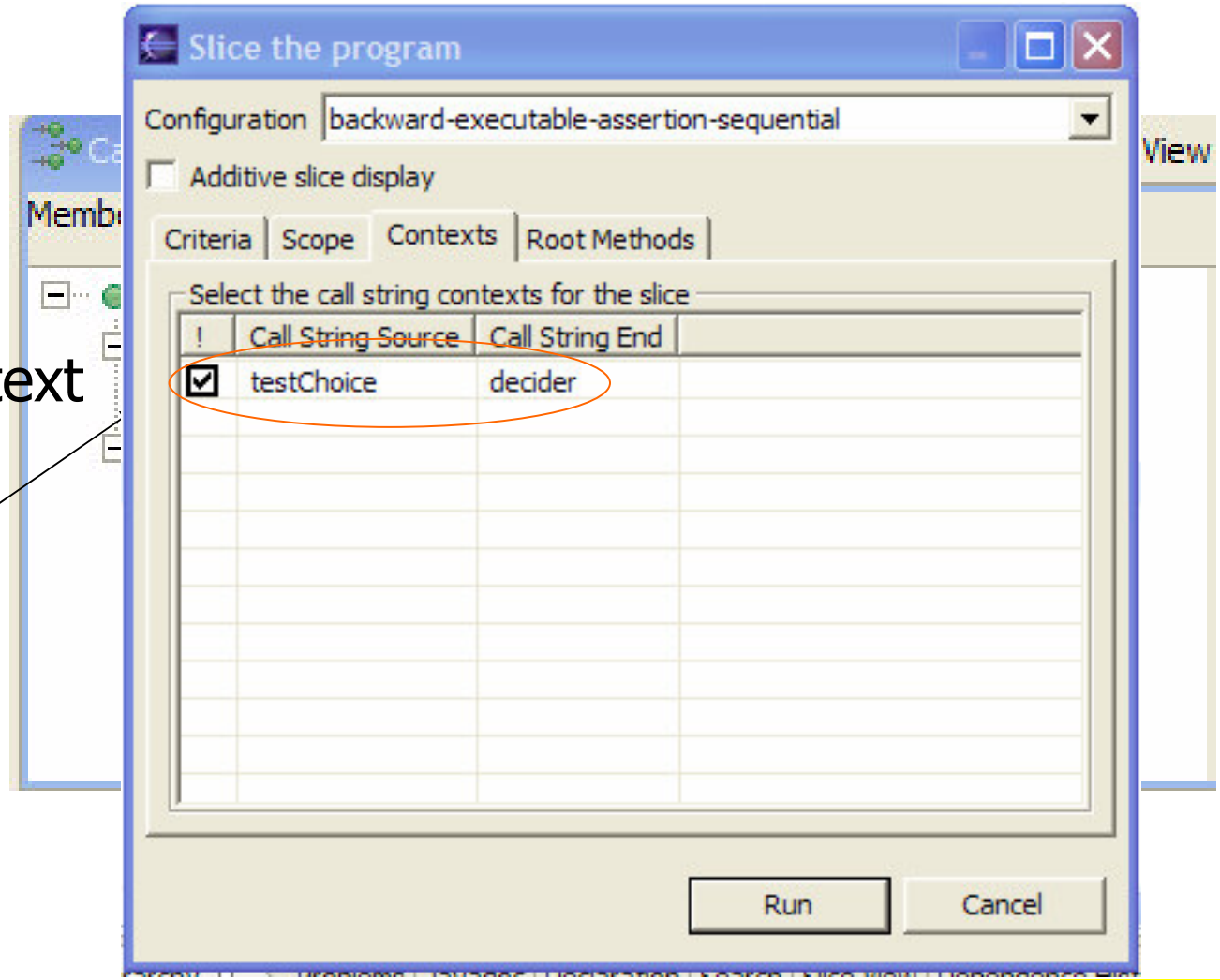


Context-bounded slice

Call leaf

Call String context

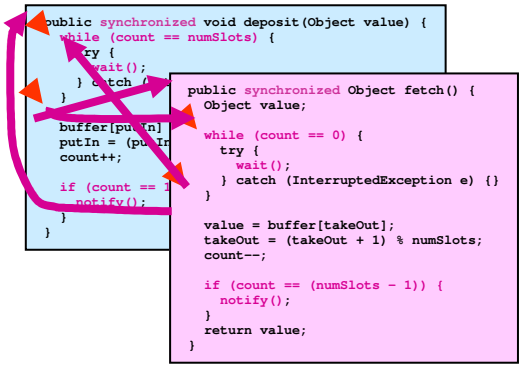
Call root



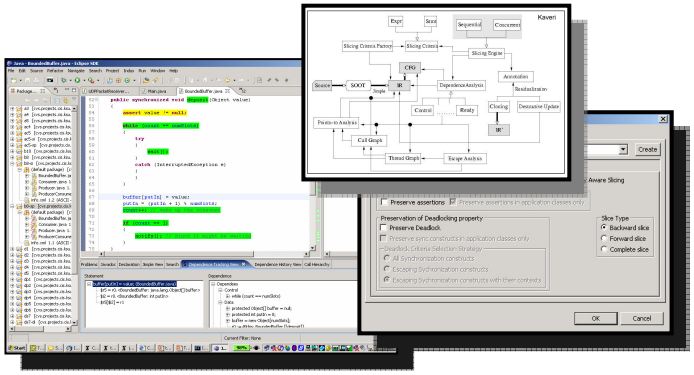
Demo

Outline

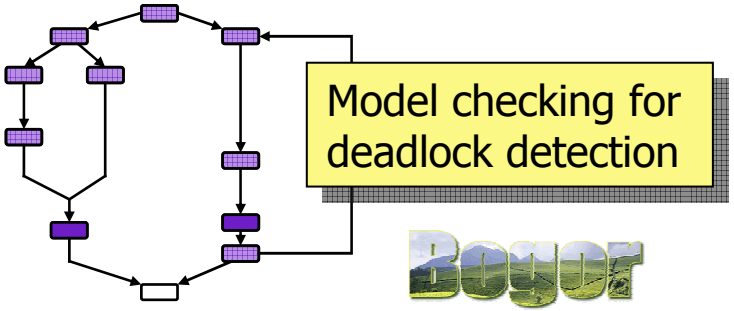
I. Program Dependences for Java



II. Indus & Kaveri

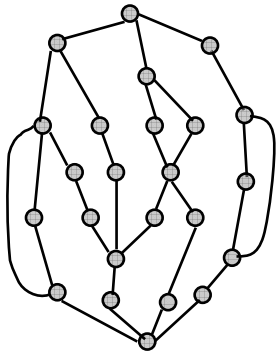


III. Application to Verifying Concurrent Java Programs



IV. Analysis optimization and mining analysis results

Model Checking



Finite-state model

$\Box(\Phi \rightarrow \Diamond\, \Omega)$

Specification

Model Checker

OK

or

Error trace

```
Line 5: …
Line 12: …
Line 15:…
Line 21:…
Line 25:…
Line 27:…
   …
Line 41:…
Line 47:…
```

Why Try to Use Model Checking for Software?

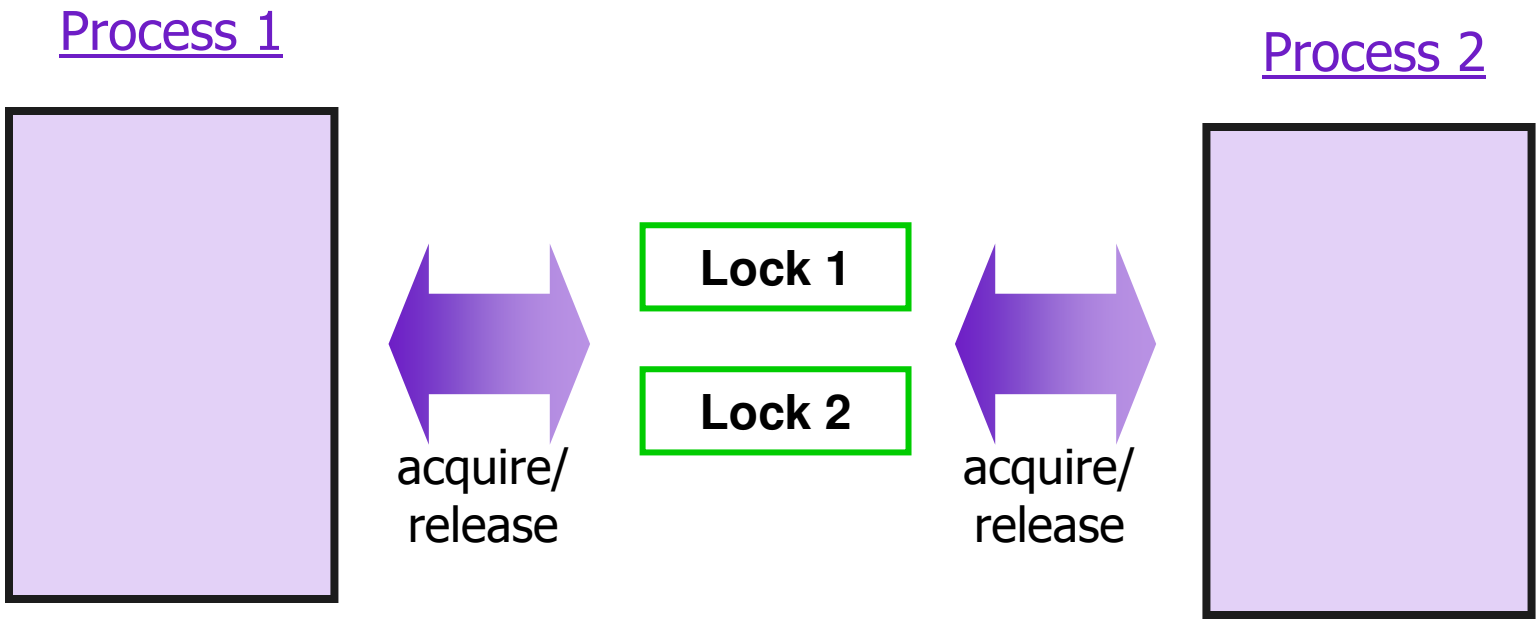
- Automatically check, e.g.,
  - invariants, simple safety & liveness properties
  - absence of dead-lock and live-lock,
  - complex event sequencing properties,

> "Between the window open and the window close, button X can be pushed at most twice."

- In contrast to testing, gives complete coverage – even in presence of concurrency -- by exhaustively exploring all paths in system,
- It's been used for years with good success in hardware and protocol design

This suggests that model-checking can complement existing software quality assurance techniques.

Simple Deadlock Example



Simple Deadlock Example

```
public class Deadlock {
  static Lock lock1;
  static Lock lock2;
  static int  state;

public static
   void main(String[] args) {
     lock1 = new Lock();
     lock2 = new Lock();
     Process1 p1
         = new Process1();
     Process2 p2
         = new Process2();
     p1.start();
     p2.start();
}
```

```
class Lock {}
```

```
class Process1 extends Thread {
  public void run() {
    Deadlock.state++;
    synchronized (Deadlock.lock1) {
      synchronized (Deadlock.lock2) {
        Deadlock.state++;
      }}}}
```

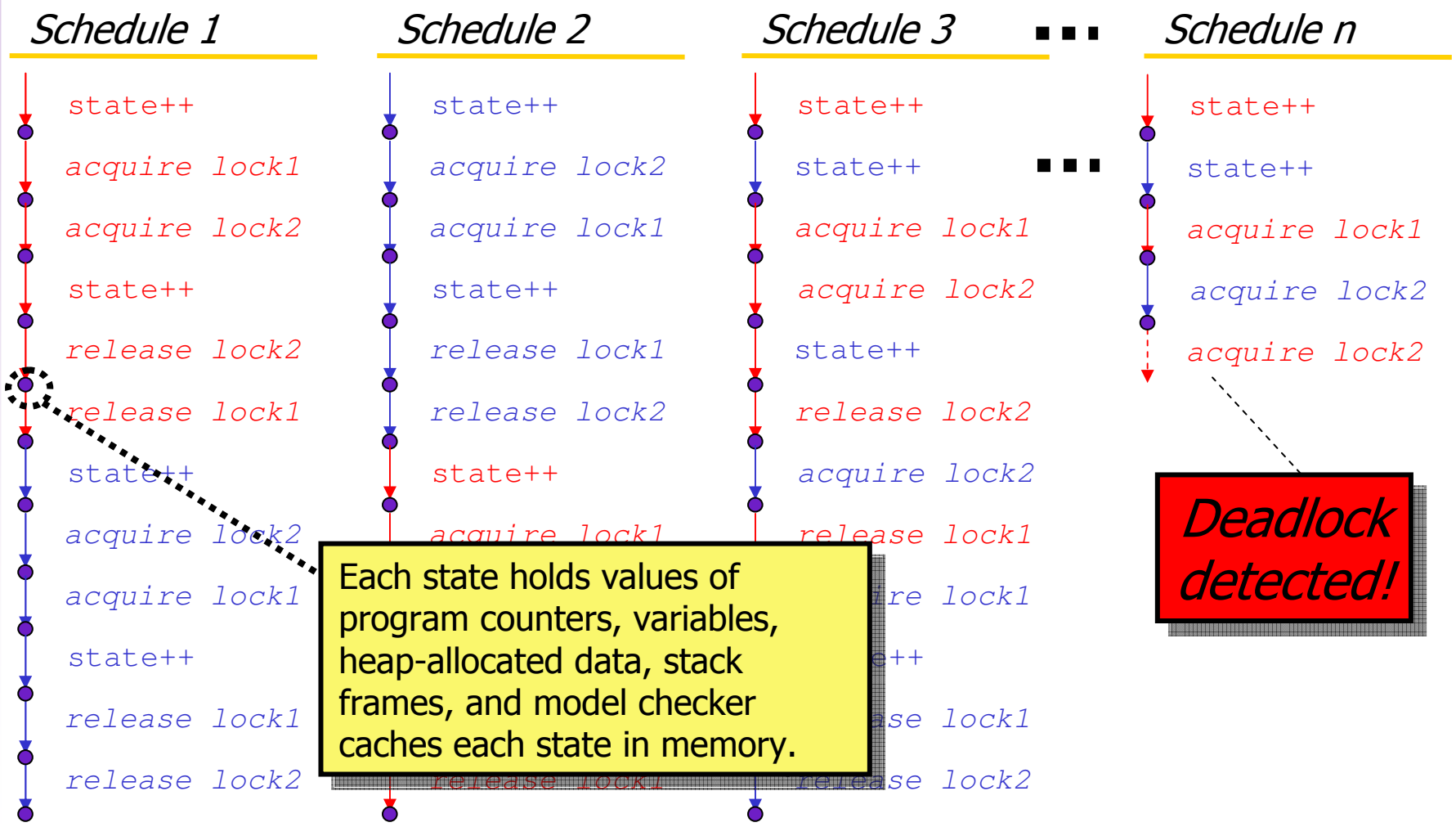
```
class Process2 extends Thread {
  public void run() {
    Deadlock.state++;
    synchronized (Deadlock.lock2) {
      synchronized (Deadlock.lock1) {
        Deadlock.state++;
      }}}}
```

Simple Deadlock Example

Process 1

Process 2

Model checking explores all schedules/interleavings



Model Checking Realistic Systems

Goal

model check implementations of realistic Java systems against a particular property or collection of properties

Complexities leading to state explosion

- concurrency, heap intensive data, extensive use of layering, design patterns, etc.
- *…there's just a lot of code!*
  - libraries, infrastructure code, etc.

Slicing for Model Reduction

Folklore: program slicing is useful for model-reduction

Basic Thesis:

"…often, significant amounts of the program are irrelevant wrt (independent of) the property being checked…"

Observation:

Program slicing is a static analysis technique for…

- reasoning about program (in)dependences
- eliminating program statements that are irrelevant to computations at particular program points.

Idea:

Use program slicing to prune away parts of the system that are irrelevant wrt property being checked.

Properties and Program Features

Property:

- absence of deadlock

Program features to drive slicing (slicing criterion):

- synchronized statements (lock acquires/releases)

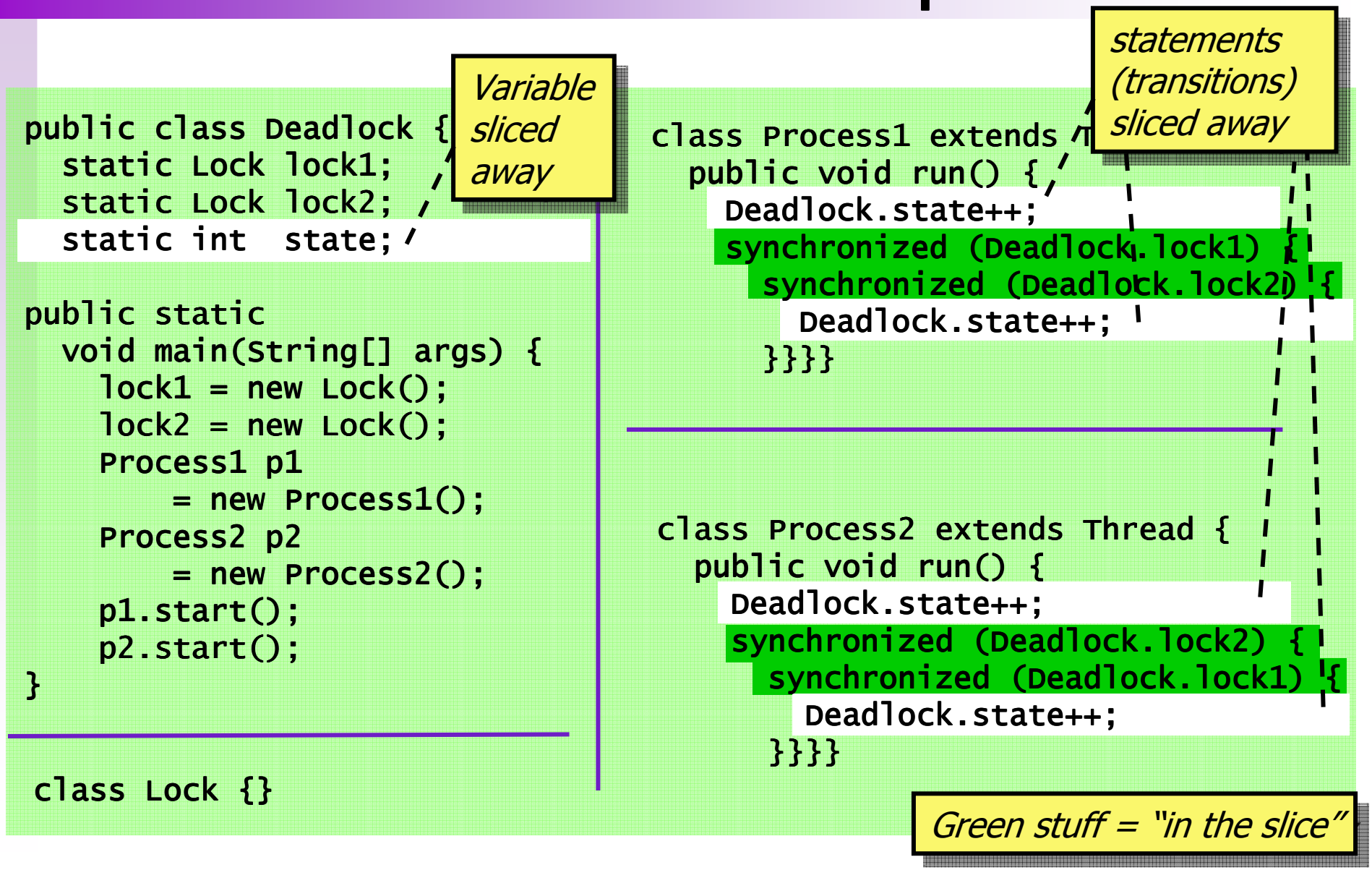
```
class Process1 extends Thread {
  public void run() {
    Deadlock.state++;
    synchronized (Deadlock.lock1) {
      synchronized (Deadlock.lock2) {
        Deadlock.state++;
      }}}}
class Process2 extends Thread {
  public void run() {
    Deadlock.state++;
    synchronized (Deadlock.lock2) {
      synchronized (Deadlock.lock1) {
        Deadlock.state++;
      }}}}
```

Slicing criteria

Slicing action:

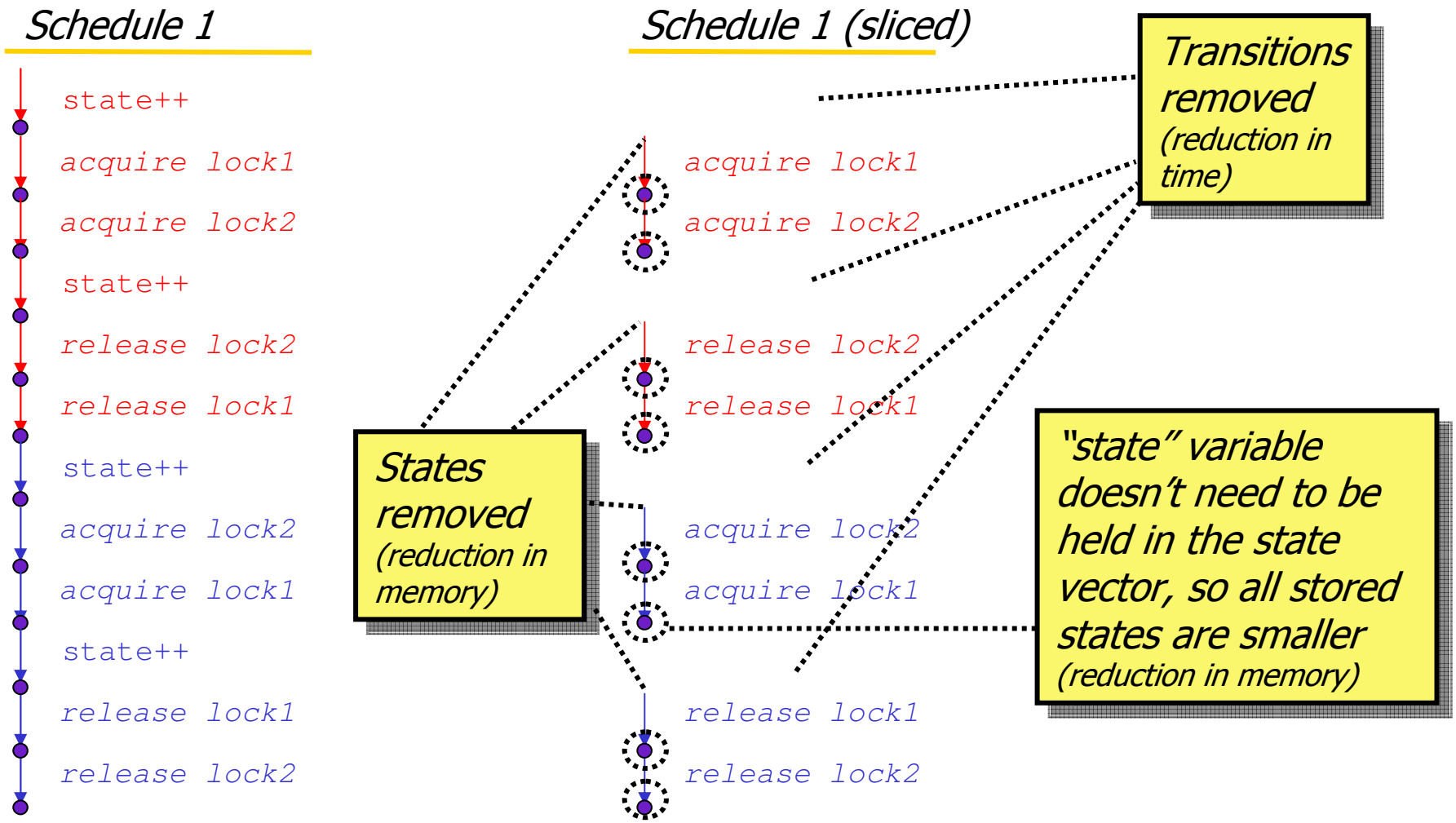
- finds all program statements that influence the computation of the synchronized statements (removes the rest)

Sliced Deadlock Example



Simple Deadlock Example

Effects of slicing on model checking costs



Expected Benefits of Slicing

Statements Removed

- fewer transitions
- fewer interleavings
- fewer states

Variables Removed

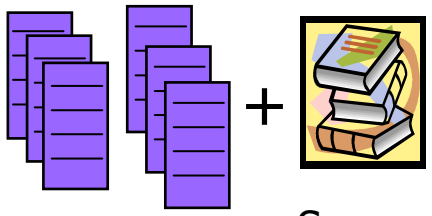
- smaller state vector

Unreachable Methods/Classes Removed

- static initializers removed
- reduces preprocessing / translation overhead

Experiments

I. Code Bases



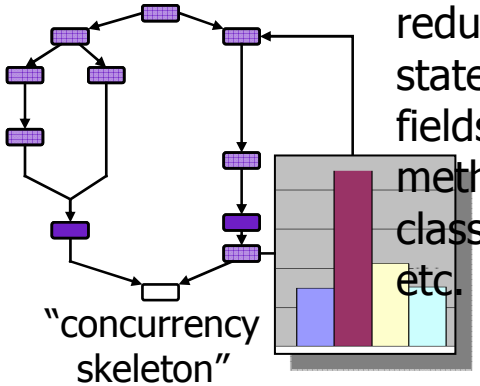
Java application code

Source version of Java libraries

slice on synchronization statements for deadlock checking

Indus

II. Slicing – Static Metrics



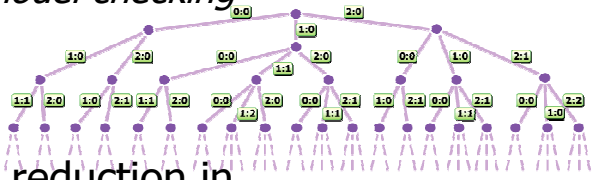
"concurrency skeleton"

reduction in statements, fields, methods, classes, etc.

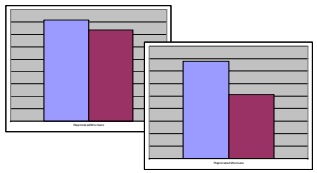
Bogor

III. Slicing – Dynamic Metrics

model checking



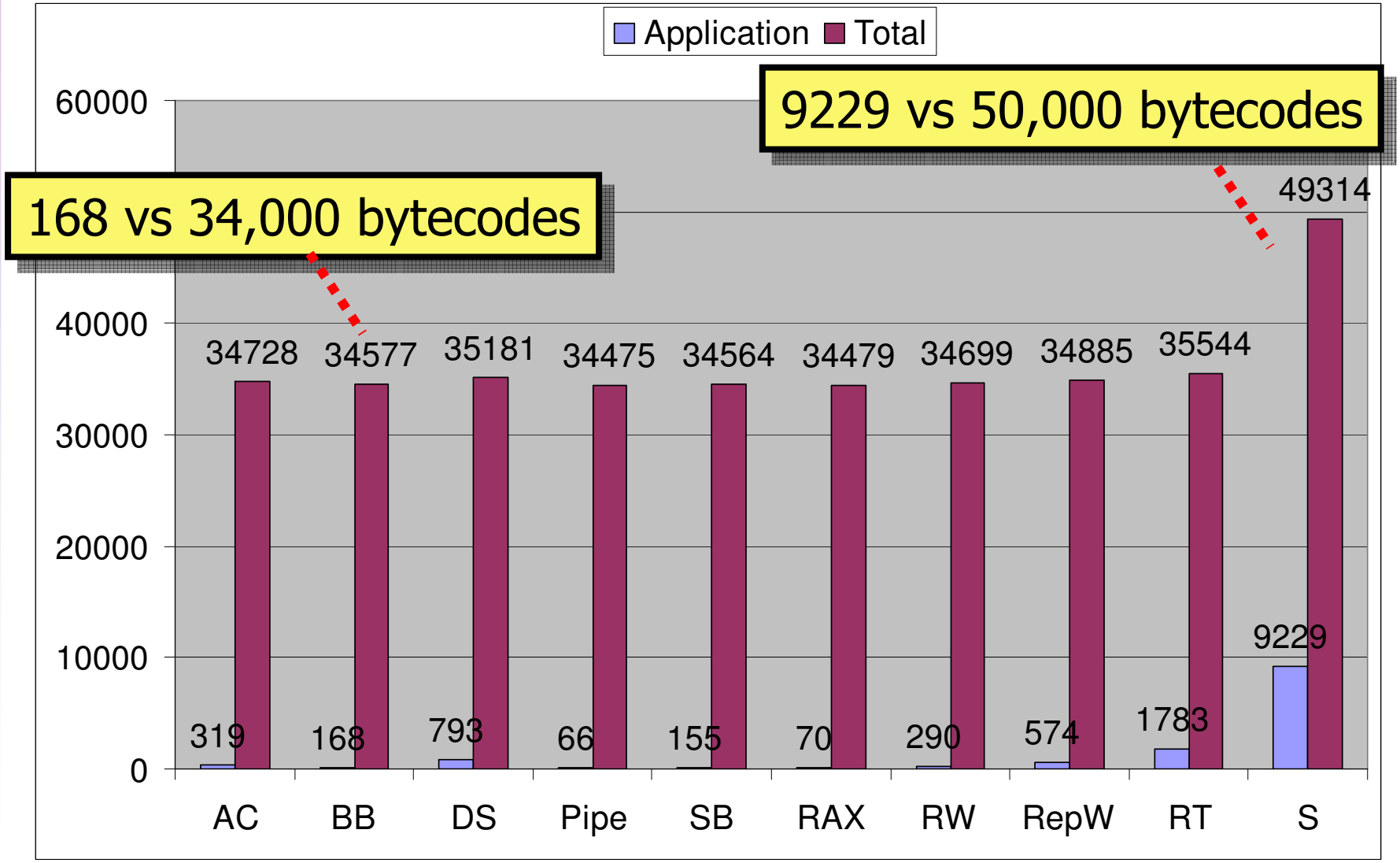
reduction in states, transitions, memory, time, etc.



Classes of Examples

- Algorithm Sketches (used elsewhere in the literature)
  - Bounded Buffer
  - Pipeline
  - Sleeping Barbers
  - Readers/Writers
  - RAX (distillation of NASA Remote Agent Bug)
- Small Applications (significantly more heap intensive)
  - Disk Scheduler
  - Alarm Clock
- Larger Applications
  - Replicated Workers (client/server data distribution framework)
  - Ray Tracer (Java Grande)
  - Siena (internet-scale publish/subscribe middleware)

Size in Bytecodes



Libraries -- Assessment

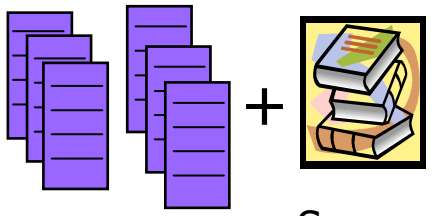
- Significant portion of program bytecodes come from libraries
- Some library code may be *unreachable during execution* but still contribute to state vector
- Some library code may be reachable but *irrelevant to property*

- Much unreachable code can be pruned away using a precise call-graph analysis (based on precise "points-to" analysis)
  - To avoid "straw man" arguments, we will take CallGraph-reachability-pruned code-bases as the baseline of slicing/model-checking experiments

Comparing slicing to an already "smart" framework!

Experiments

I. Code Bases



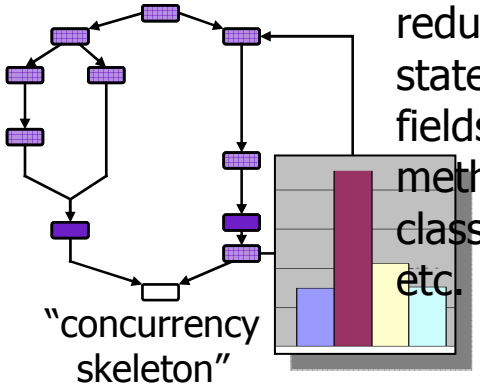
Java application code

Source version of Java libraries

slice on synchronization statements for deadlock checking

Indus

II. Slicing – Static Metrics



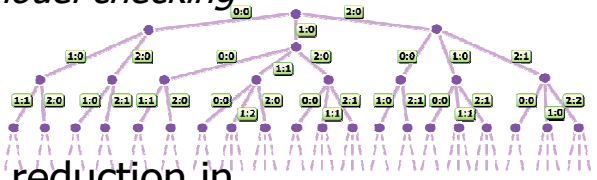
"concurrency skeleton"

reduction in statements, fields, methods, classes, etc.

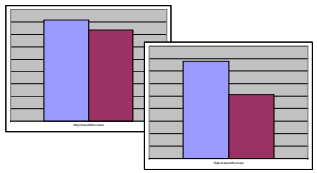
Bogor

III. Slicing – Dynamic Metrics

model checking

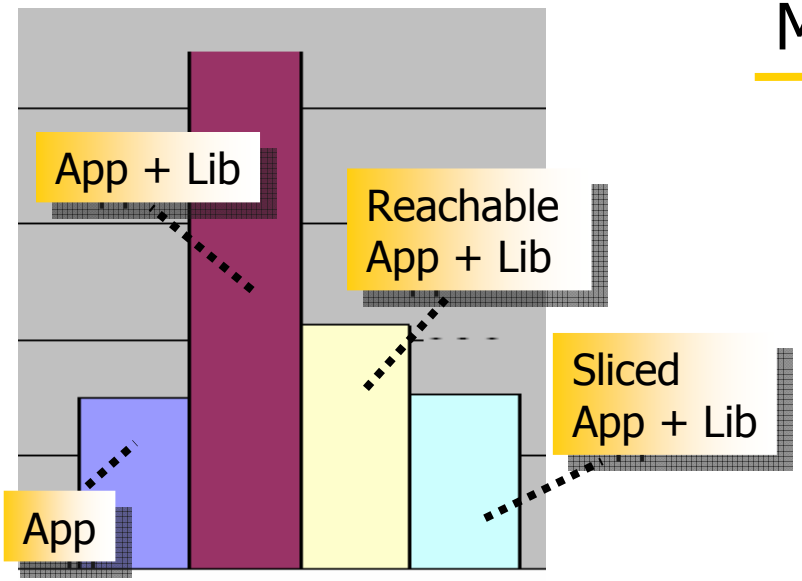


reduction in states, transitions, memory, time, etc.



Static Metrics for Slicing

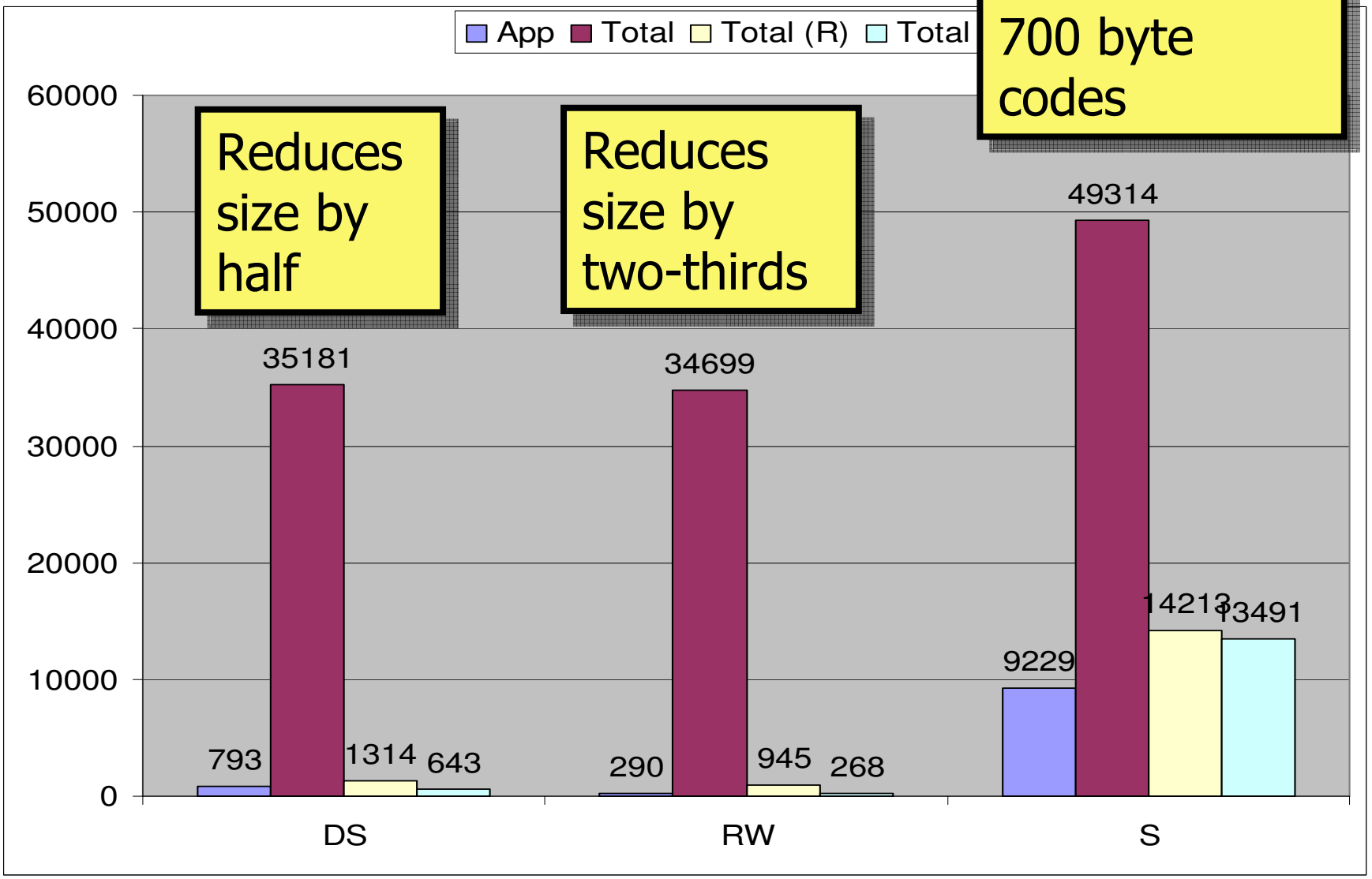
What is the effect of Call-Graph-reachability and slicing on the static program model structure?



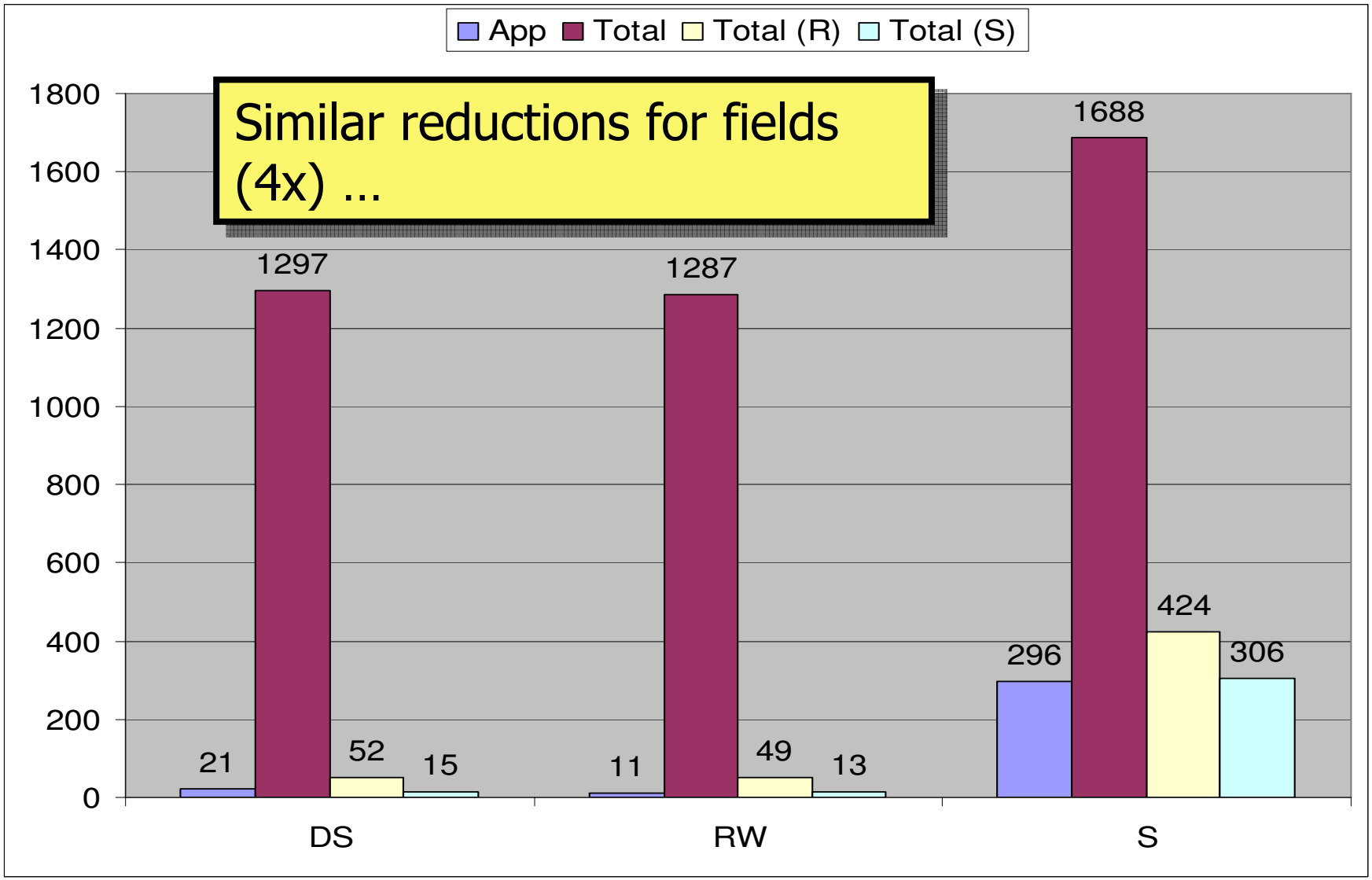
Measurements (see paper)

- bytecodes
- classes
- methods
- fields
- new (allocations)
- threads
- exceptions
- synchronization statements
- wait
- notify

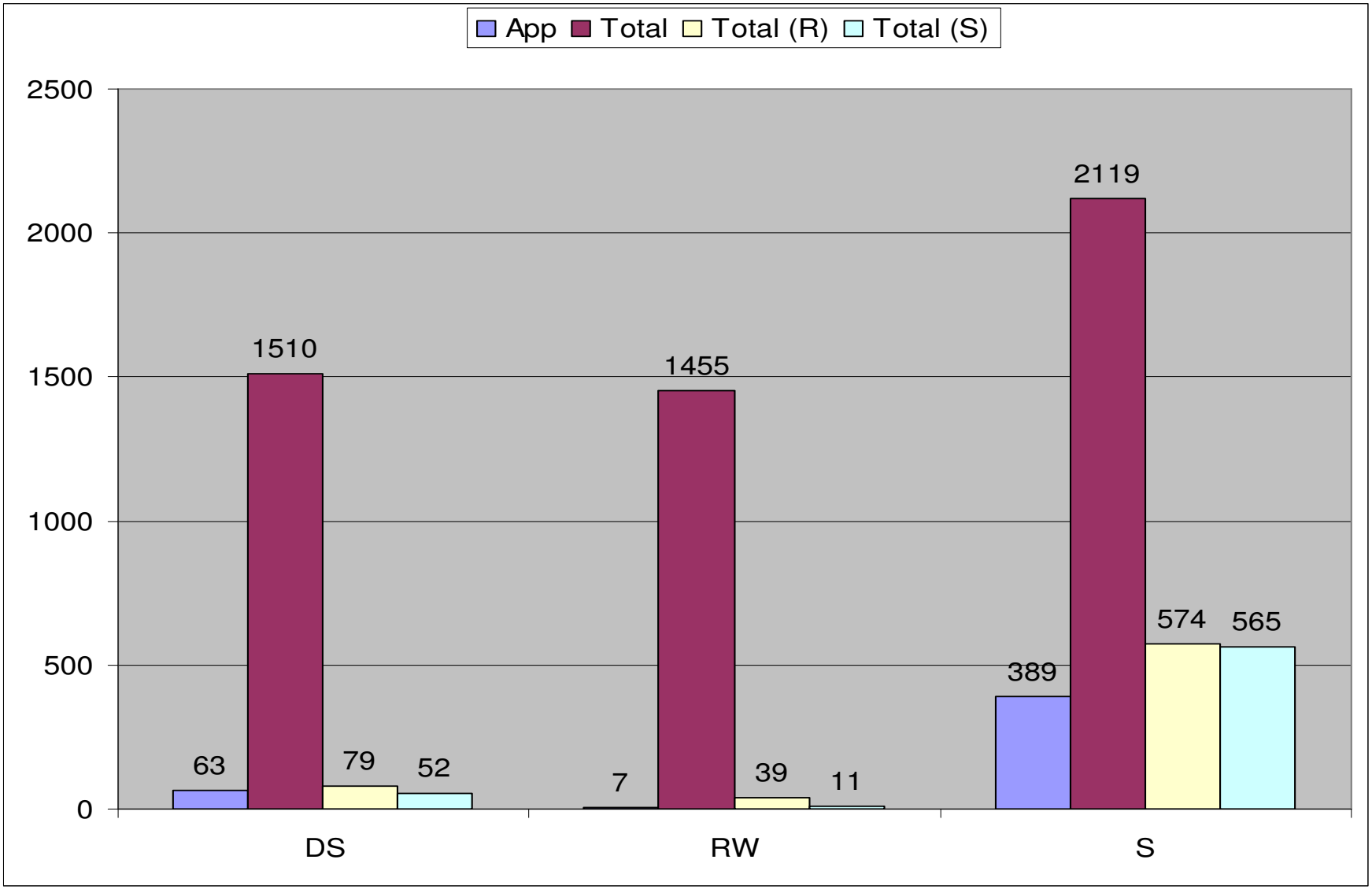
Static Metrics – Byte-codes



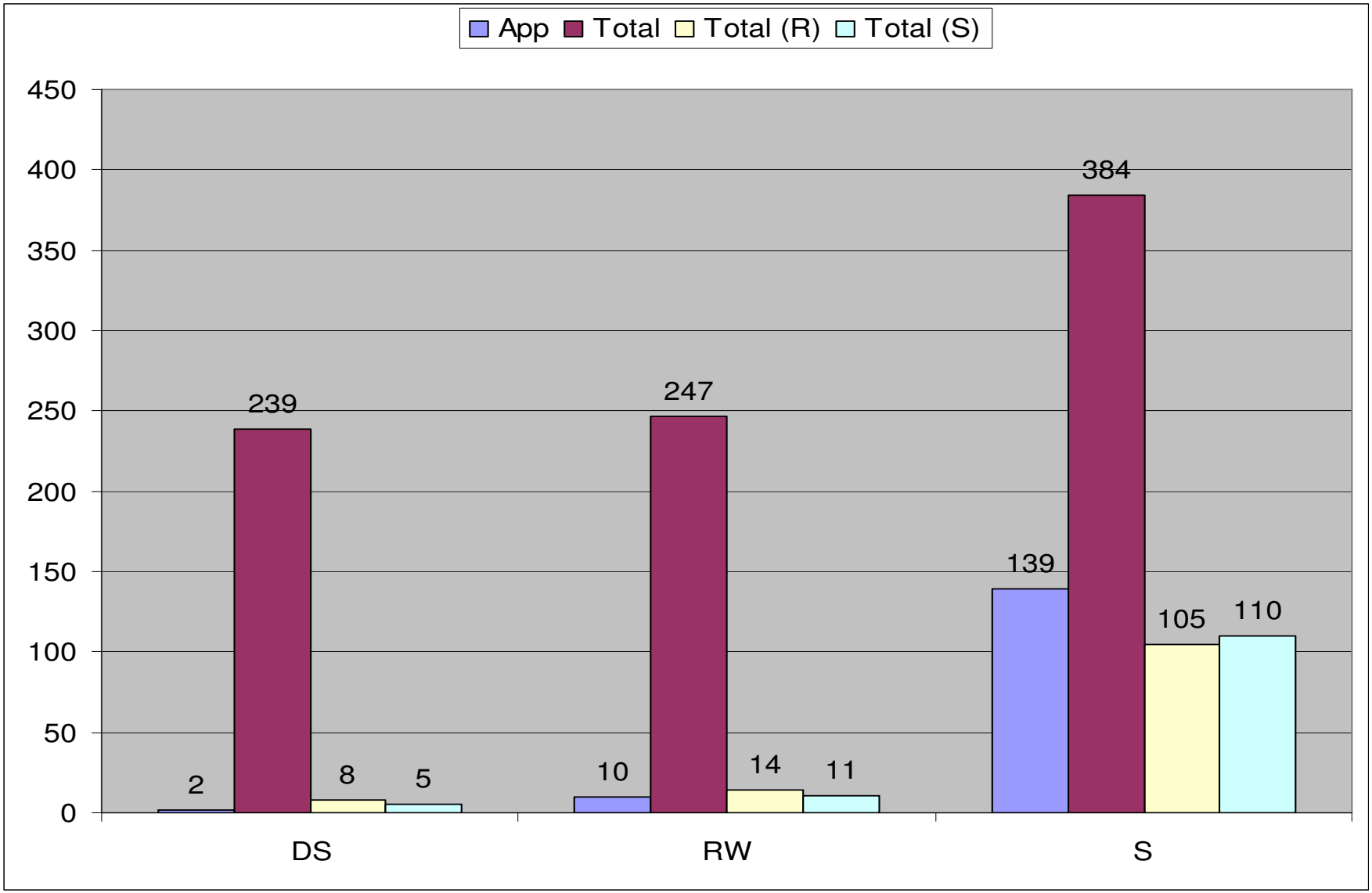
Static Metrics – Fields



Static Metrics - New

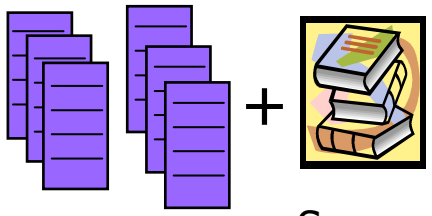


Static Metrics - Sync



Experiments

I. Code Bases



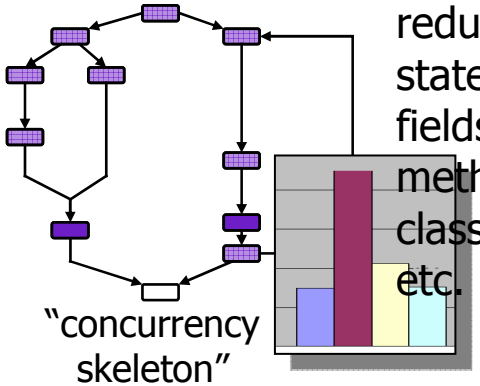
Java application code

Source version of Java libraries

slice on synchronization statements for deadlock checking

Indus

II. Slicing – Static Metrics



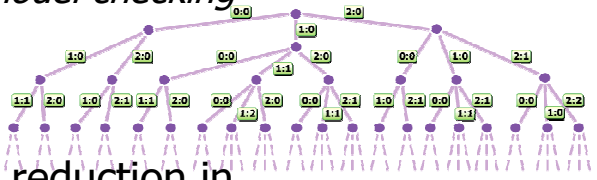
"concurrency skeleton"

reduction in statements, fields, methods, classes, etc.

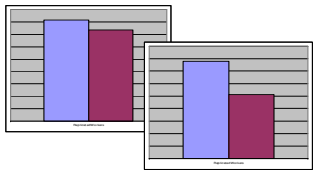
Bogor

III. Slicing – Dynamic Metrics

model checking



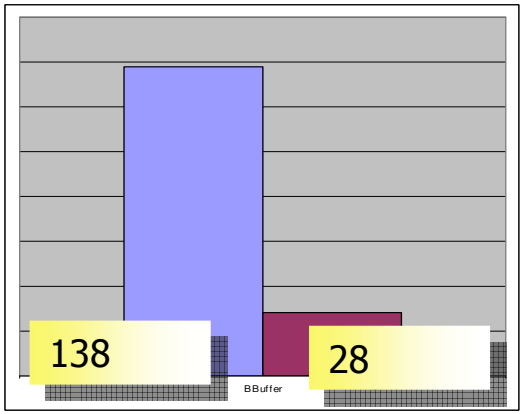
reduction in states, transitions, memory, time, etc.



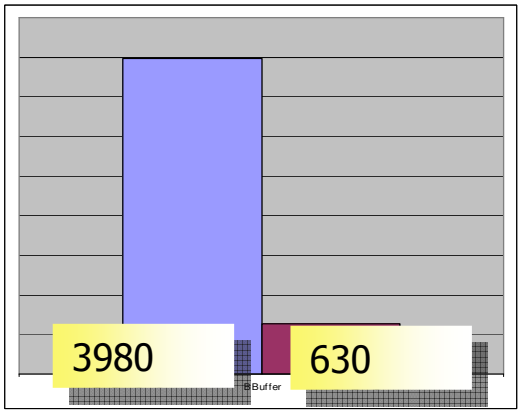
Dynamic Metrics - BBuffer w/POR

Between factor of 4-5 reduction

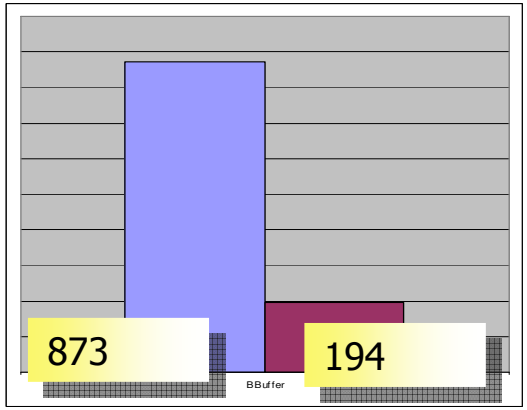
States



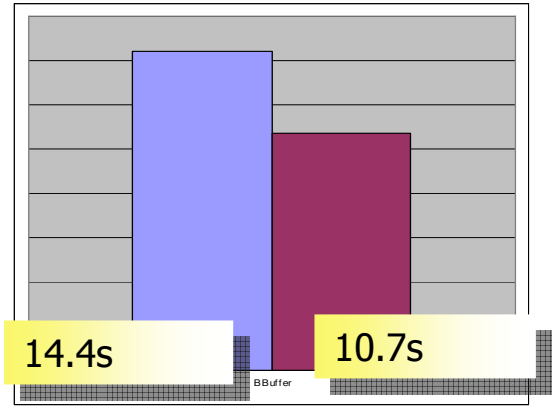
Transitions



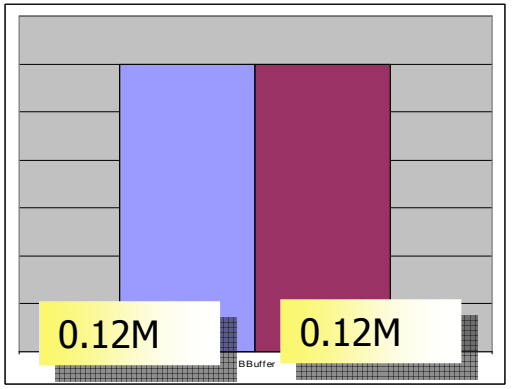
Byte-codes

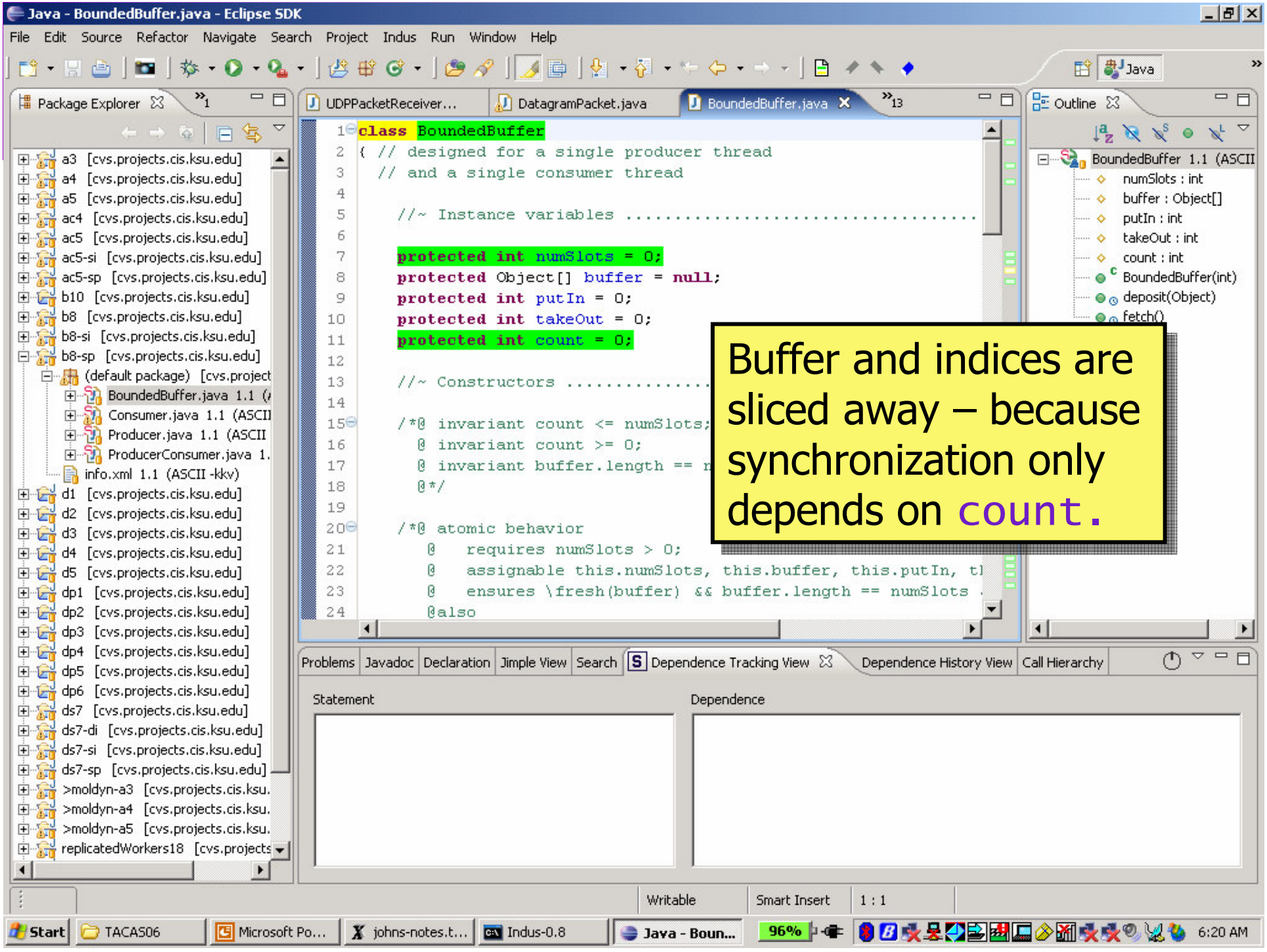


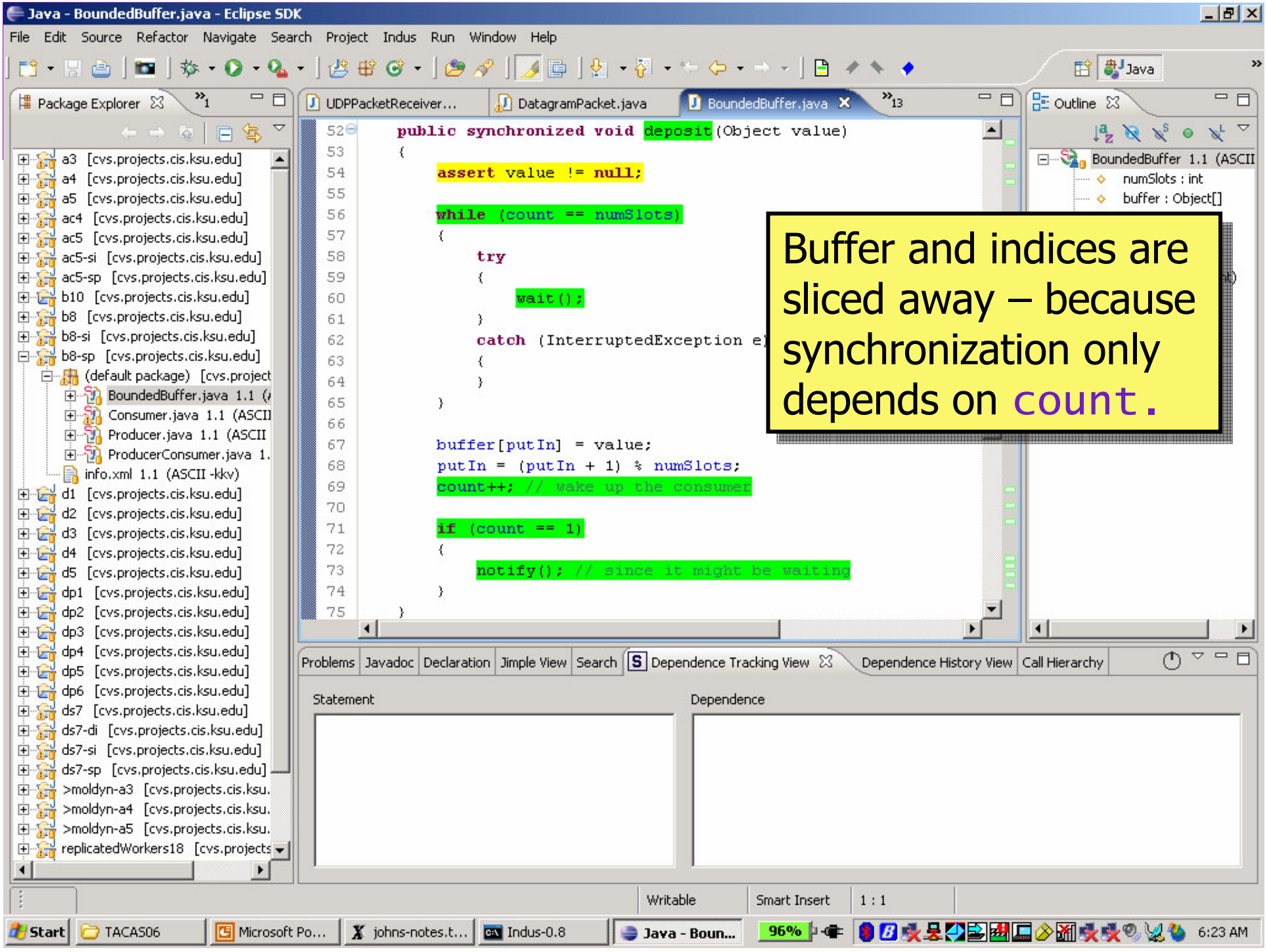
Time



Memory



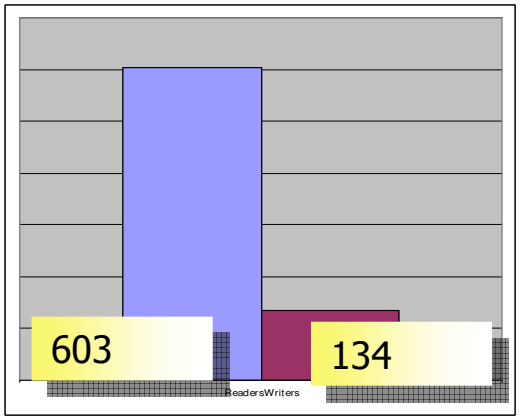

Java - BoundedBuffer.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Indus Run Window Help
Package Explorer
UDPPacketReceiver...
DatagramPacket.java
BoundedBuffer.java
Outline
1 class BoundedBuffer
2 { // designed for a single producer thread
3 // and a single consumer thread
5 //~ Instance variables .....
7 protected int numSlots = 0;
8 protected Object[] buffer = null;
9 protected int putIn = 0;
10 protected int takeOut = 0;
11 protected int count = 0;
13 //~ Constructors .....
15 /*@ invariant count <= numSlots;
16 @ invariant count >= 0;
17 @ invariant buffer.length ==
18 @*/
20 /*@ atomic behavior
21 @ requires numSlots > 0;
22 @ assignable this.numSlots, this.buffer, this.putIn, t
23 @ ensures \fresh(buffer) && buffer.length == numSlots
24 @also
BoundedBuffer 1.1 (ASCII
numSlots : int
buffer : Object[]
putIn : int
takeOut : int
count : int
BoundedBuffer(int)
deposit(Object)
fetch()
Buffer and indices are sliced away – because synchronization only depends on count.
Problems Javadoc Declaration Jimple View Search Dependence Tracking View Dependence History View Call Hierarchy
Statement
Dependence
Writable
Smart Insert
1 : 1
Start
TACAS06
Microsoft Po...
johns-notes.t...
Indus-0.8
Java - Boun...
96%
6:20 AM


Java - BoundedBuffer.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Indus Run Window Help
Package Explorer
UDPPacketReceiver...
DatagramPacket.java
BoundedBuffer.java
Outline
BoundedBuffer 1.1 (ASCII
numSlots : int
buffer : Object[]
52 public synchronized void deposit(Object value)
53 {
54 assert value != null;
55
56 while (count == numSlots)
57 {
58 try
59 {
60 wait();
61 }
62 catch (InterruptedException e)
63 {
64 }
65 }
66
67 buffer[putIn] = value;
68 putIn = (putIn + 1) % numSlots;
69 count++; // wake up the consumer
70
71 if (count == 1)
72 {
73 notify(); // since it might be waiting
74 }
75 }
Buffer and indices are sliced away – because synchronization only depends on count.
Problems Javadoc Declaration Jimple View Search Dependence Tracking View Dependence History View Call Hierarchy
Statement
Dependence
Writable
Smart Insert
1 : 1
Start
TACAS06
Microsoft Po...
johns-notes.t...
Indus-0.8
Java - Boun...
96%
6:23 AM

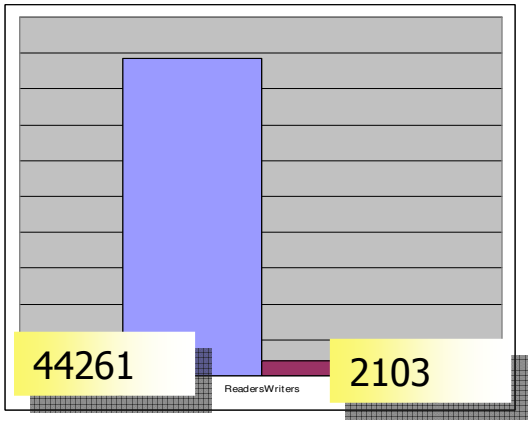
Dynamic Metrics - ReadersWriters w/POR

Significant reductions…

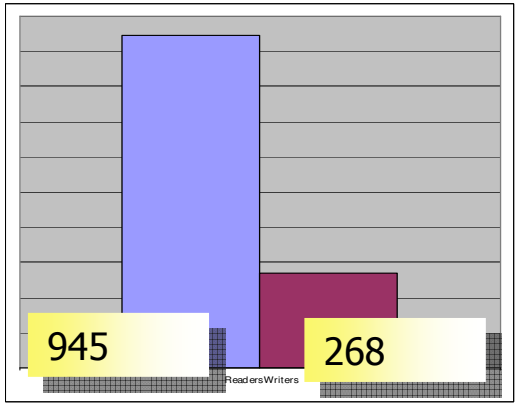
States



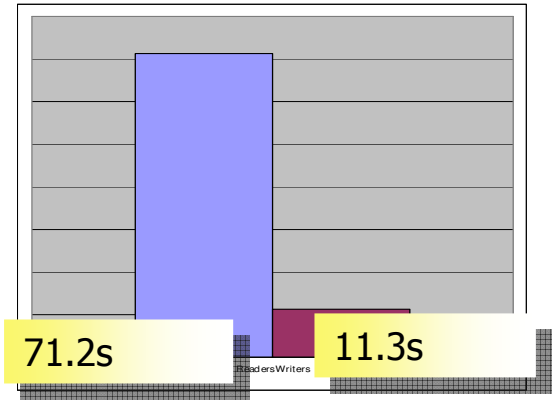
Transitions



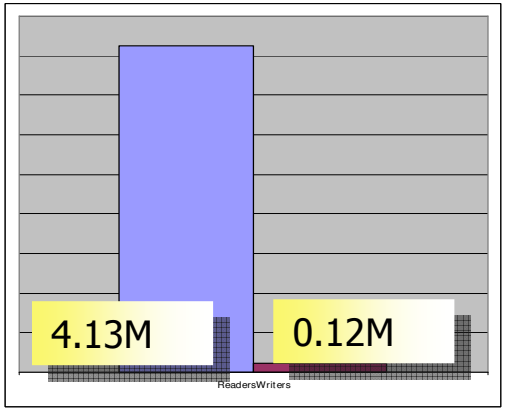
Byte-codes



Time



Memory



Assessment

- Very little application code sliced away (a counter, and system.out.println's)
- Dramatic amount of library code is sliced away
  - 25 classes, 53 methods, 40 fields, and 683 statements
  - java.lang.System gets completed gutted

java.util.Properties
java.io.PrintStream
java.io.InputStream
java.io.OutputStream

java.util.Hashtable
java.io.BufferedReader
java.io.InputStreamReader,
java.io.Reader
java.lang.StringBuffer
java.io.PrintWriter
java.util.Iterator
...and much more

Slicing Siena Middleware

UDPPacketReceiver.java

```
…
packet = new DatagramPacket(new byte[1], 1);
…
```

Not in the slice. Why?

DatagramPacket.java

```
…
public DatagramPacket(final byte[] b,
                      final int length) {
        setData(b, 0, length);
    }
…
```

Passed to constructor, and then to setData.

```
public void setData(final byte[] data,
                    final int offset,
                    final int lengt
        buffer = data;
        setLength(length);
    }
```

Passed to setData, and then assigned to buffer which is not read by any statement in the slice!

Slicing Siena Middleware

UDPPacketReceiver.java

```
…
packet = new DatagramPacket(new byte[1], 1);
…
```

Not in the slice. Why?

DatagramPacket.java

```
…
public DatagramPacket(final byte[
                      final int l
        setData(b, 0, length);
    }
…
```

```
public void setData(final byte[]
                    final int off
                    final int len
        buffer = data;
        setLength(length);
    }
```

Summary:

Via several intermediate steps, `new byte[1]` is flowing across multiple methods and classes only into a variable that is never used in the slice.

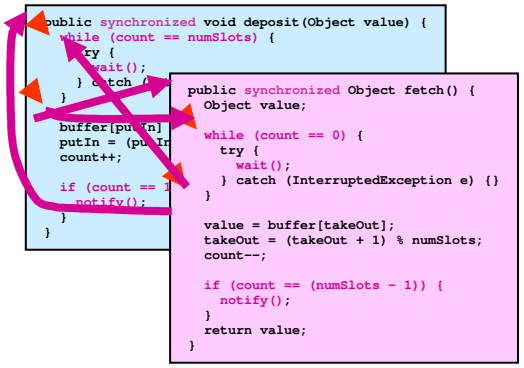
Very difficult to detect manually – especially on large code bases!

Assessment

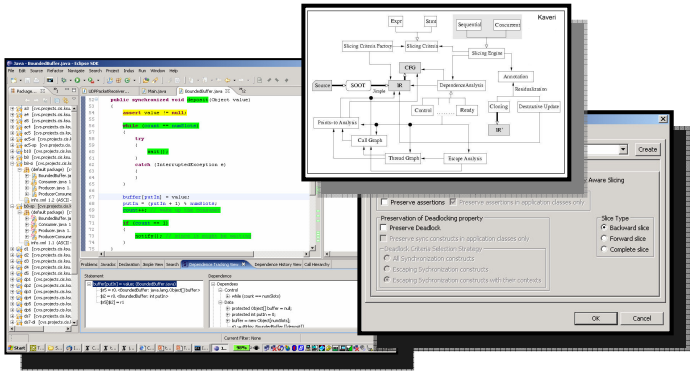
- Applying Indus slicing for Java model checking
  - is easy (completely automatic – no user intervention)
  - is inexpensive (compared to model checking costs)
  - is effective and orthogonal to other reduction strategies (typical reduction factors range from 2x – 5x)
- Key enabling technology: additional notions of dependence related to preserving blocking/deadlocking behavior
- Slicing techniques like Indus seem particularly relevant for real systems that make significant use of libraries

Outline

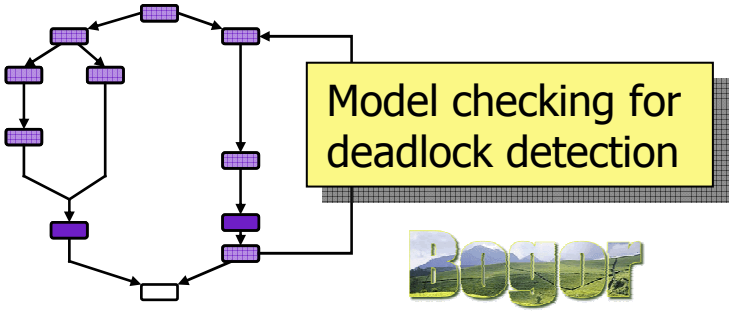
I. Program Dependences for Java



II. Indus & Kaveri



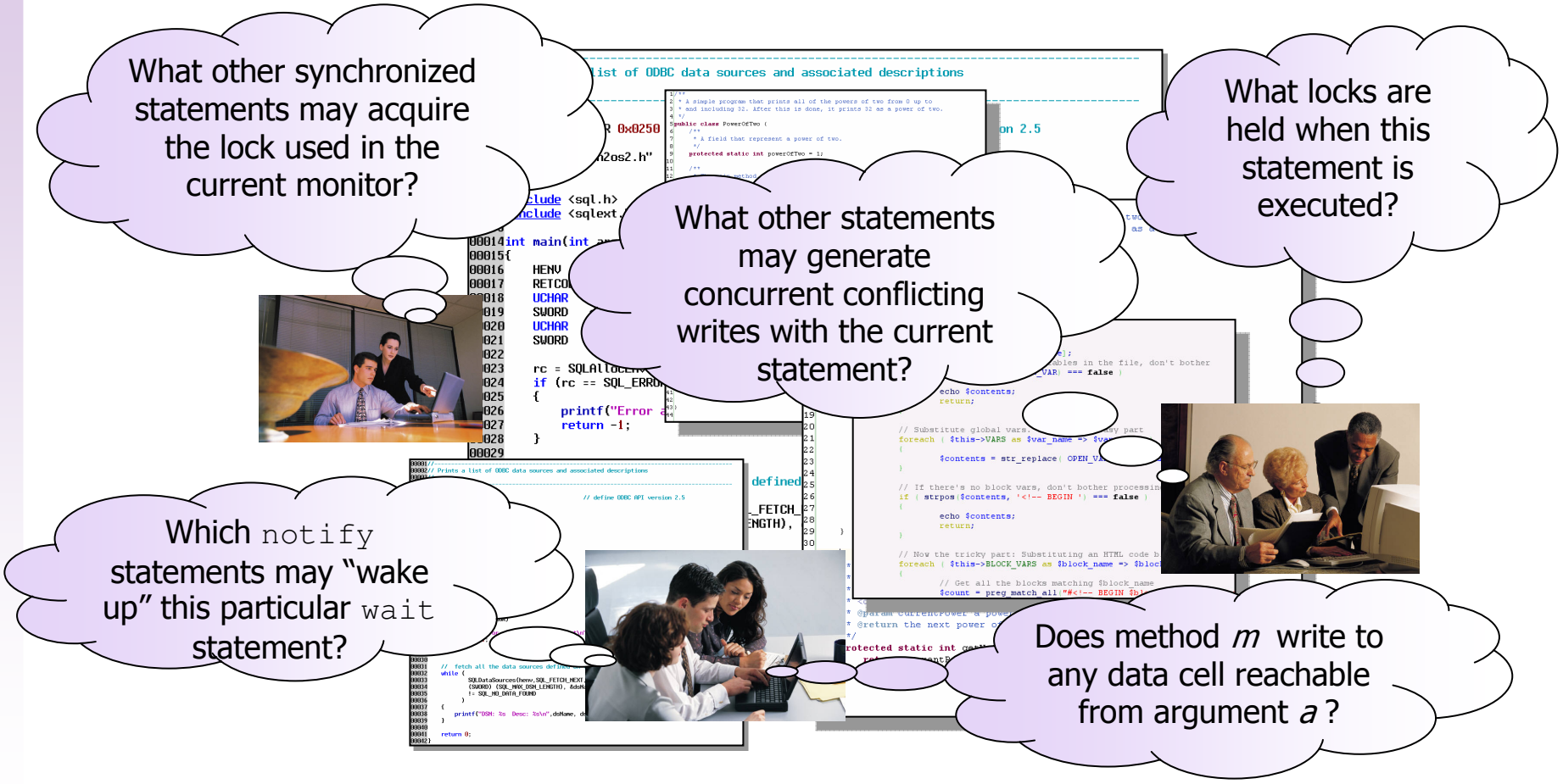
III. Application to Verifying Concurrent Java Programs



IV. Analysis optimization and mining analysis results

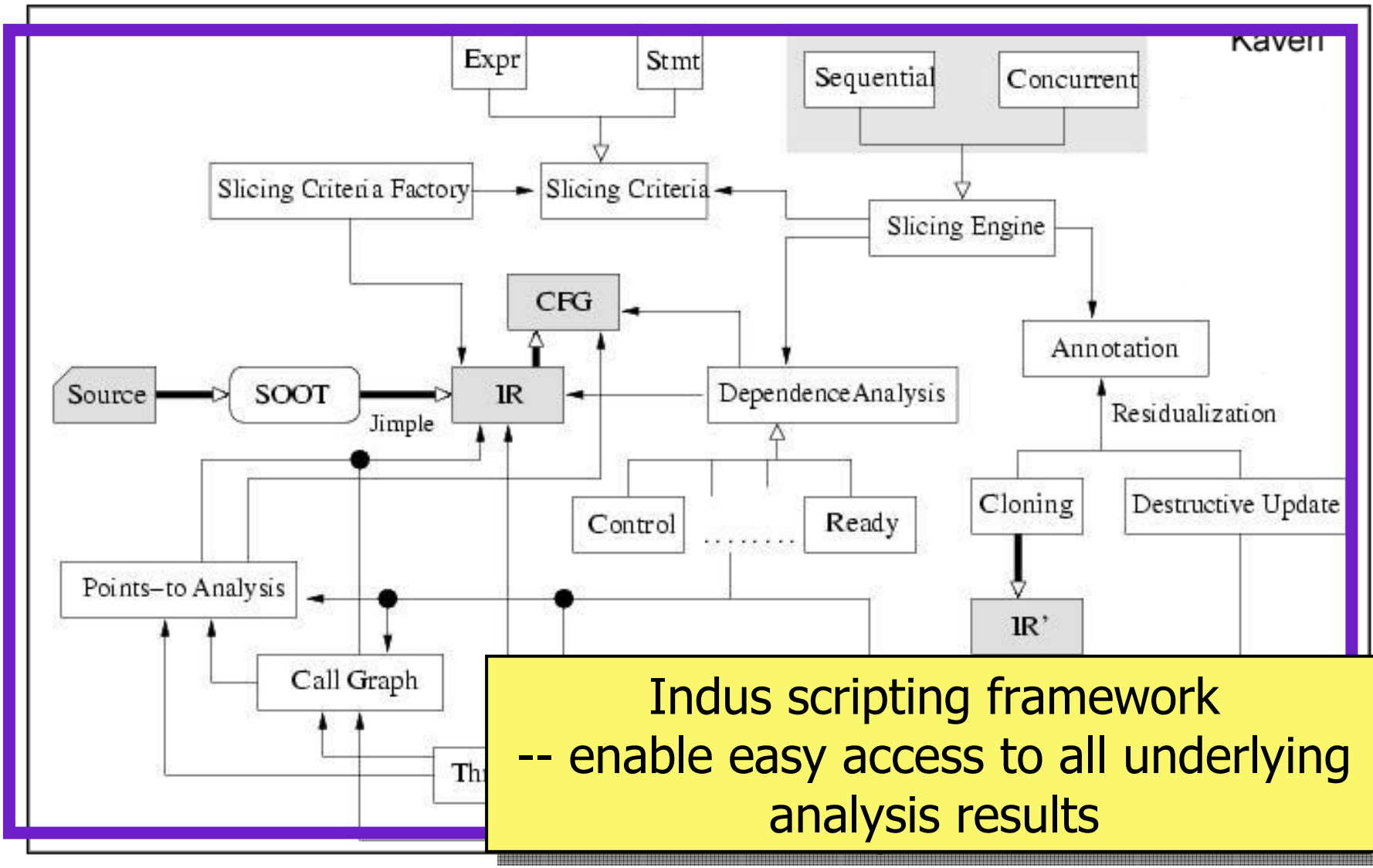
Beyond Traditional Slicing Questions

Mining the results of wide-ranging static analysis for concurrency



…we are interested in going well beyond than the usual slicing questions

Indus / Kaveri Architecture



Indus scripting framework
-- enable easy access to all underlying analysis results

An Example

```
class Acct {
    protected Integer balance;
    Acct() {
        balance = new Integer(100);
    }
}
```

```
public class Home {
    public static void main(String[] s) {
        Acct savings = new Acct();
        Thread wife = null;
        Thread  man = null;
        wife = new Wife(savings);
        husband = new Man(savings);
        wife.start();
        husband.start();
    }
}
```

```
class Wife extends Thread {
    protected Acct savings;
    protected Acct checking;
    Wife(Acct act) {
        this.savings = act;
        checking = new Acct();
    }
    public void run() {
        try {
            synchronized(savings) {
                savings.wait();
            }
            Acct newAcct = new Acct();
            savings.balance = new Integer(savings.balance.intValue() - 20);
            checking.balance = new Integer(checking.balance.intValue() - 10);
            newAcct.balance = new Integer(newAcct.balance.intValue() + 10);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
class Husband  extends Thread {
    protected Acct savings;
    protected Acct checking;
    Husband (Acct act) {
        this.savings = act;
        checking = new Acct();
    }
    public void run() {
        savings.balance = new Integer(savings.balance.intValue() + 20);
        synchronized(savings) {
            savings.notify();
        }

        checking.balance = new Integer(checking.balance.intValue() + 1(
        synchronized(checking) {
            checking.notify();
        }
    }
}
```

An Example

```
class Acct {
    protected Integer balance;
    Acct() {
        balance = new Integer(100);
    }
}
```

Objects of account type are shared between threads.

Access to `balance` may induce inter-thread data interferences.

An Example

`savings` account is *shared* between a `man` and `wife`.

```
public class Home {
    public static void main(String[] s) {
        Acct savings = new Acct();
        Thread wife = null;
        Thread  husband = null;
        wife = new Wife(savings);
        man = new Husband(savings);
        wife.start();
        husband.start();
    }
}
```

An Example

```
class Wife extends Thread {
    protected Acct savings;
    protected Acct checking;
    Wife(Acct act) {
        this.savings = act;
        checking = new Acct();
    }
    public void run() {
        try {
            synchronized(savings) {
                savings.wait();
            }
            Acct newAcct = new Acct();
            savings.balance = new Integer(savings.balance.intValue() - 20);
            checking.balance = new Integer(checking.balance.intValue() - 10);
            newAcct.balance = new Integer(newAcct.balance.intValue() + 10);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Assign the shared savings account in an instance field

Wife has a personal `checking` account.

Wife waits for a transaction on the `savings` account

Wife saves some money in a private local account, `newAcct`.

An Example

```
class Man extends Thread {
    protected Acct savings;
    protected Acct checking;
    Man(Acct act) {
        this.savings = act;
        checking = new Acct();
    }
    public void run() {
        savings.balance = new Integer(savings.balance.intValue() + 20);
        synchronized(savings) {
            savings.notify();
        }

        checking.balance = new Integer
        synchronized(checking) {
            checking.notify();
        }
    }
}
```

Mas has a personal `checking` account.

Man deposits money into the `savings` account and *notifies* others about it.

Man deposits money into the `checking` account and *notifies* others about it. **But it is a personal account, so the effort is wasted!!**

Sociology Issues

Husband using bank account to bet on football

For the good of the family, wife is day-trading on the side and collecting profits in private account

Example Queries

Using escape analysis

- Show all concurrent writes that conflict
- Show other synchronized statements that may be contending for the same lock
- Show all synchronized states encountered along call paths to this line
- At wait, show influencing notifies
- At notify, show dependent waits
- etc.

Example Queries

Using side-effects analysis

- Does method *m* write to any field reachable from argument *a?*
- Does method *m* read any fields reachable from argument *a?*
- Does method *m* depend on the any static fields?

Conclusions

- Indus provides a wide-ranging collection of capabilities for reasoning about dependences in concurrent Java programs
- Current trends suggest many interesting opportunities for slicing in a concurrent context
- Indus/Kaveri is available for download
  - over 3000 downloads within the past two years
  - working with KSU to release under open source license by end of year

Themes for the Future

- synergistic combination with other techniques
  - dynamic analysis for mining dependences
  - symbolic execution (conditional slicing) to calculate path constraints
  - using dependence information to guide heuristics in model checking
- domain-specific layers on top of Indus that mine dependence information according to specific developer needs
- additional optimizations to aid scalability and precision

Related Technical Papers

- Slicing for model checking reductions (TACAS 06)
- Foundations of Control Dependence (ESOP ’05)
- Pruning Interference Dependence using Escape Analysis (CC ’04)
- Preserving LTL with Slicing (HOSC ’01)
- Notions of Dependence for Java (SAS ’99)

For More Information…

SAnToS Laboratory,
Kansas State University
http://www.cis.ksu.edu/santos

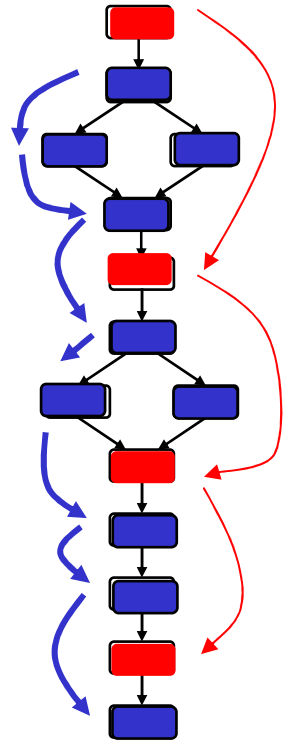
Indus Project
http://indus.projects.cis.ksu.edu

Bogor Project
http://bogor.projects.cis.ksu.edu

Applications

Automatic Parallelization

Program CFG

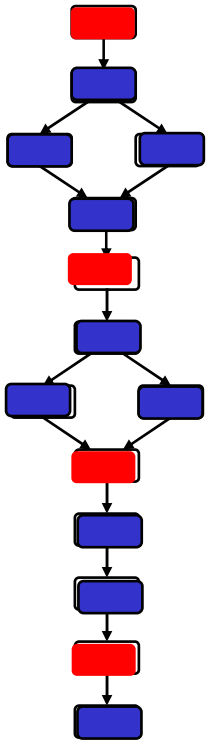


Use slicing to find independent sequences of computation

Applications

Automatic Parallelization

Thread 1

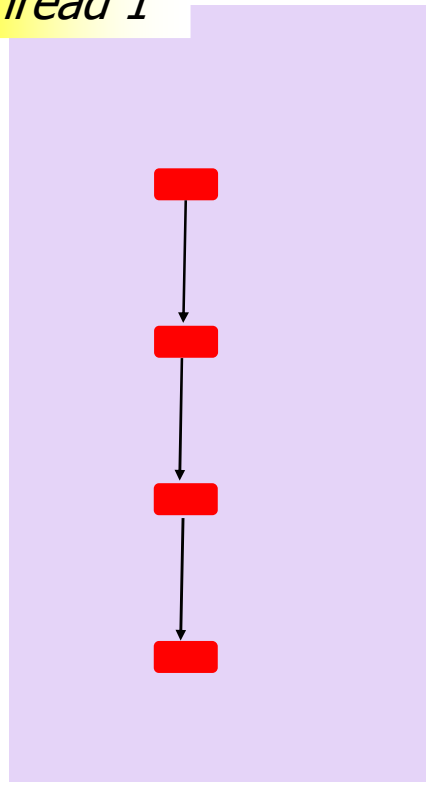


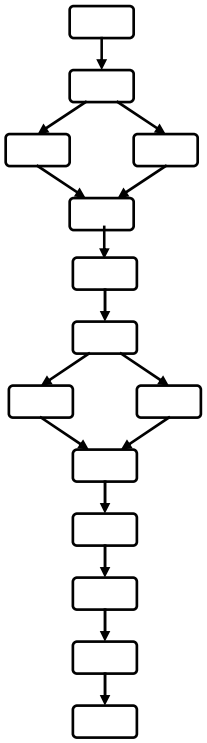
Thread 2

Applications

Automatic Parallelization

Thread 1





Thread 2