A meta heuristic for effectively detecting concurrency errors

Neha Rungta & Eric G. Mercer

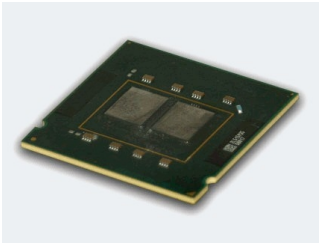
Software Model Checking Lab

Brigham Young University

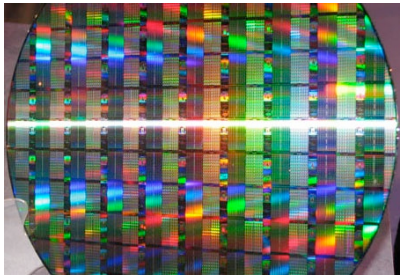
Provo, UT

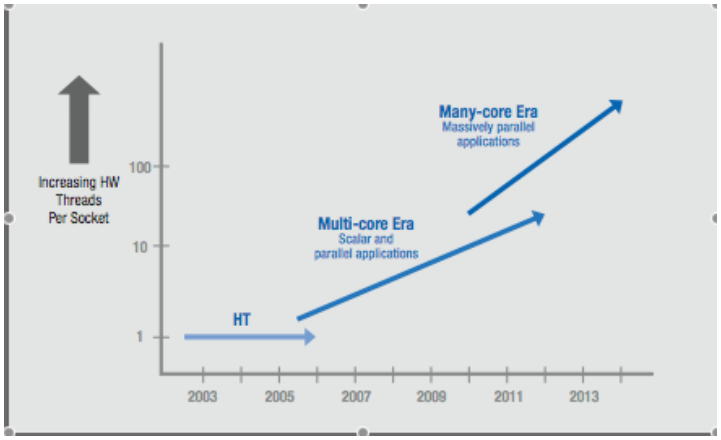
Current Trend

Dual and Quad Core Processors are becoming increasingly common



Intel's 80 core prototype





More processors on a single die *

* Image courtesy Intel white paper

State of the art

Static Analysis

- Reports Possible errors
- Imprecise analyses
- Scalable to large systems
- Jlint, FindBugs
- RacerX, Chord

Dynamic Analysis

- Instrumenting Code
- Runtime Monitor & Test
- May miss certain errors
- ConTest, RaceFinder
- Chess, Eraser

Model Checking

- Precise and Sound
- All possible errors
- Does not scale
- JPF, SLAM, Blast
- SPIN, Bogor

Directed Model Checking

- Error Discovery is the goal
- Uses heuristic function
- Localize the search
- Information about program or the user-defined property

Static Analysis

Analyze the source code of the program

```
 */
public synchronized boolean equals(Object o) {
if (o == this)
    return true;

if (!(o instanceof Map))
    return false;
Map t = (Map) o;
if (t.size() != size())
    return false;

    try {
        Iterator i = entrySet().iterator();
        while (i.hasNext()) {
            Map.Entry e = (Map.Entry) i.next();
            Object key = e.getKey();
            Object value = e.getValue();
            if (value == null) {
                if (!(t.get(key)==null && t.containsKey(key)))
                    return false;
            } else {
                if (!value.equals(t.get(key)))
                    return false;
            }
        }
    } catch(ClassCastException unused)   {
        return false;
    } catch(NullPointerException unused) {
        return false;
    }

return true;
}

/**
 * Returns the hash code value for this Map as per the definition in the
 * Map interface.
-:--  Hashtable.java    72% (723,20)    (Java/l Abbrev)----------
```

```
 */
public synchronized boolean equals(Object o) {
    return super.equals(o);
}

/**
 * Returns the hash code value for this Vector.
 */
public synchronized int hashCode() {
    return super.hashCode();
}

/**
-:--  Vector.java     88% (911,0)     (Java/l Abbrev)-----
```

Static Analysis

Output is a set of ***possible errors*** in the program

```
 */
public synchronized boolean equals(Object o) {
if (o == this)
    return true;

if (!(o instanceof Map))
    return false;
Map t = (Map) o;
if (t.size() != size())
    return false;

    try {
        Iterator i = entrySet().iterator();
        while (i.hasNext()) {
            Map.Entry e = (Map.Entry) i.next();
            Object key = e.getKey();
            Object value = e.getValue();
            if (value == null) {
                if (!(t.get(key)==null && t.containsKey(key)))
                    return false;
            } else {
                if (!value.equals(t.get(key)))
                    return false;
            }
        }
    } catch(ClassCastException unused)   {
        return false;
    } catch(NullPointerException unused) {
        return false;
    }

return true;
}

/**
 * Returns the hash code value for this Map as per the definition in the
 * Map interface.
-:--  Hashtable.java    72% (723,20)    (Java/l Abbrev)----------
```

```
 */
public synchronized boolean equals(Object o) {
    return super.equals(o);
}

/**
 * Returns the hash code value for this Vector.
 */
public synchronized int hashCode() {
    return super.hashCode();
}

/**
-:--  Vector.java     88% (911,0)     (Java/l Abbrev)-----
```

```
/Users/neha/Documents/test/java/bin/java/util/Vector.java:363: Loop 2: invocation of synchronized method java/util/Hashtable.equals(java.lang.Object) can cause deadlock.
/Users/neha/Documents/test/java/bin/java/util/Hashtable.java:333: Loop 2: invocation of synchronized method java/util/Vector.hashCode() can cause deadlock.
/Users/neha/Documents/test/java/bin/java/util/Hashtable.java:336: Loop 3: invocation of synchronized method java/util/Hashtable.equals(java.lang.Object) can cause deadlock.
/Users/neha/Documents/test/java/bin/java/util/Vector.java:363: Loop 4: invocation of synchronized method java/util/Hashtable.equals(java.lang.Object) can cause deadlock.
/Users/neha/Documents/test/java/bin/java/util/Hashtable.java:336: Loop 4: invocation of synchronized method java/util/Vector.equals(java.lang.Object) can cause deadlock.
/Users/neha/Documents/test/java/bin/java/util/Hashtable.java:398: Loop 5: invocation of synchronized method java/util/Hashtable.hashCode() can cause deadlock.
/Users/neha/Documents/test/java/bin/java/util/Vector.java:363: Loop 6: invocation of synchronized method java/util/Hashtable.equals(java.lang.Object) can cause deadlock.
/Users/neha/Documents/test/java/bin/java/util/Hashtable.java:398: Loop 6: invocation of synchronized method java/util/Vector.hashCode() can cause deadlock.
/Users/neha/Documents/test/java/bin/java/util/Hashtable.java:401: Loop 7: invocation of synchronized method java/util/Hashtable.equals(java.lang.Object) can cause deadlock.
/Users/neha/Documents/test/java/bin/java/util/Vector.java:363: Loop 8: invocation of synchronized method java/util/Hashtable.equals(java.lang.Object) can cause deadlock.
/Users/neha/Documents/test/java/bin/java/util/Hashtable.java:401: Loop 8: invocation of synchronized method java/util/Vector.equals(java.lang.Object) can cause deadlock.
/Users/neha/Documents/test/java/bin/java/util/Hashtable.java:311: Loop 9: invocation of synchronized method java/util/Hashtable.hashCode() can cause deadlock.
/Users/neha/Documents/test/java/bin/java/util/Vector.java:363: Loop 10: invocation of synchronized method java/util/Hashtable.equals(java.lang.Object) can cause deadlock.
```

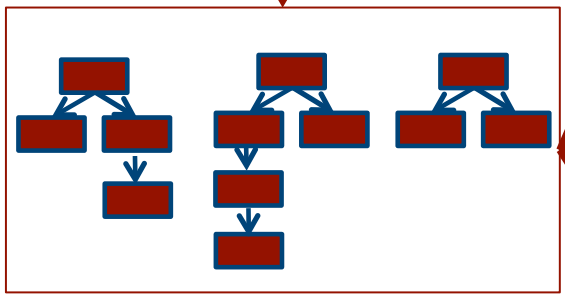
Major Drawback

Static Analysis Warnings

- No
- No
- No
- **Give up!!!**

Overview of our technique

1. Static Analysis Warnings
2. Program
3. Environment



- Yes
  Error Trace
- Yes
  Error Trace
- Yes
  Error Trace
- Don't know
- Don't know
- Don't know
- Don't know

Meta-heuristic

Input Sequence

Greedy depth-first search

Two-tier ranking

Yes, Found Error Counter-example

Don't know

Advantages

- Only reports feasible errors
- Manual effort significantly reduced
- Evaluated on benchmarks and real multi-threaded Java programs

How to generate sequences?

Motivating Example

```
class testProgram {
    …
    public static void main () {
       AbstractList X := new Vector();
       AbstractList Y := new Vector();
       Athread A:= new Athread(X, Y);
       Bthread B:= new Bthread(X, Y);
       A.start();
       B.start();
         ...
    }
}
```

```
class Athread extends Thread{
    …
    public void run () {
       Y.add(some_object);
          ...
}
```

```
class Bthread extends Thread{
    …
    public void run () {
      bool test = X.equals(Y);
}
```

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
          …
   }
}
```

```
class Vector extends AbstractList{
    …
 public synchronized boolean equals (Object o) {
     super.equals(o);
 }

public synchronized boolean add (Object o) {
     /* Add element to its own instance */
 }

}
```

Motivating Example

```
class testProgram {
    …
    public static void main () {
       AbstractList X := new Vector();
       AbstractList Y := new Vector();
       Athread A:= new Athread(X, Y);
       Bthread B:= new Bthread(X, Y);
       A.start();
       B.start();
        ...
    }
}
```

```
class Athread extends Thread{
    …
    public void run () {
       Y.add(some_object);
          ...
}
```

```
class Bthread extends Thread{
    …
    public void run () {
      bool test = X.equals(Y);
}
```

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
          …
   }
}
```

```
class Vector extends AbstractList{
    …
 public synchronized boolean equals (Object o) {
     super.equals(o);
 }

public synchronized boolean add (Object o) {
     /* Add element to its own instance */
 }

}
```

Motivating Example

```
class testProgram {
    …
    public static void main () {
       AbstractList X := new Vector();
       AbstractList Y := new Vector();
       Athread A:= new Athread(X, Y);
       Bthread B:= new Bthread(X, Y);
       A.start();
       B.start();
        ...
    }
}
```

```
class Athread extends Thread{
    …
    public void run () {
       Y.add(some_object);
          ...
}
```

```
class Bthread extends Thread{
    …
    public void run () {
      bool test = X.equals(Y);
}
```

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
          …
   }
}
```

```
class Vector extends AbstractList{
    …
 public synchronized boolean equals (Object o) {
     super.equals(o);
 }

public synchronized boolean add (Object o) {
     /* Add element to its own instance */
 }

}
```

Motivating Example

```
class testProgram {
    …
    public static void main () {
       AbstractList X := new Vector();
       AbstractList Y := new Vector();
       Athread A:= new Athread(X, Y);
       Bthread B:= new Bthread(X, Y);
       A.start();
       B.start();
         ...
    }
}
```

```
class Athread extends Thread{
    …
    public void run () {
       Y.add(some_object);
          ...
}
```

```
class Bthread extends Thread{
    …
    public void run () {
      bool test = X.equals(Y);
}
```

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
          …
   }
}
```

```
class Vector extends AbstractList{
    …
 public synchronized boolean equals (Object o) {
     super.equals(o);
 }

public synchronized boolean add (Object o) {
     /* Add element to its own instance */
 }

}
```

Motivating Example

```
class testProgram {
    …
    public static void main () {
       AbstractList X := new Vector();
       AbstractList Y := new Vector();
       Athread A:= new Athread(X, Y);
       Bthread B:= new Bthread(X, Y);
       A.start();
       B.start();
        ...
    }
}
```

```
class Athread extends Thread{
    …
    public void run () {
       Y.add(some_object);
          ...
}
```

```
class Bthread extends Thread{
    …
    public void run () {
      bool test = X.equals(Y);
}
```

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
          …
   }
}
```

```
class Vector extends AbstractList{
    …
 public synchronized boolean equals (Object o) {
     super.equals(o);
 }

public synchronized boolean add (Object o) {
     /* Add element to its own instance */
 }

}
```

Motivating Example

```
class testProgram {
    …
    public static void main () {
       AbstractList X := new Vector();
       AbstractList Y := new Vector();
       Athread A:= new Athread(X, Y);
       Bthread B:= new Bthread(X, Y);
       A.start();
       B.start();
         ...
    }
}
```

```
class Athread extends Thread{
    …
    public void run () {
       Y.add(some_object);
          ...
}
```

```
class Bthread extends Thread{
    …
    public void run () {
      bool test = X.equals(Y);
}
```

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
          …
   }
}
```

```
class Vector extends AbstractList{
    …
 public synchronized boolean equals (Object o) {
     super.equals(o);
 }

public synchronized boolean add (Object o) {
     /* Add element to its own instance */
 }

}
```

Motivating Example

```
class testProgram {
    …
    public static void main () {
       AbstractList X := new Vector();
       AbstractList Y := new Vector();
       Athread A:= new Athread(X, Y);
       Bthread B:= new Bthread(X, Y);
       A.start();
       B.start();
        ...
    }
}
```

```
class Athread extends Thread{
    …
    public void run () {
       Y.add(some_object);
          ...
}
```

```
class Bthread extends Thread{
    …
    public void run () {
      bool test = X.equals(Y);
}
```

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
           …
   }
}
```

```
class Vector extends AbstractList{
    …
 public synchronized boolean equals (Object o) {
     super.equals(o);
 }

public synchronized boolean add (Object o) {
     /* Add element to its own instance */
 }

}
```

Jlint Warnings

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
           …
   }
}
```

```
AbstractList.java:627: Method java/util/Iterator.hasNext() can be called from different threads and is not synchronized.
AbstractList.java:628: Method java/util/Iterator.next() can be called from different threads and is not synchronized.
AbstractList.java:612: Method java/util/ListIterator.hasNext() can be called from different threads and is not synchronized.
AbstractList.java:612: Method java/util/ListIterator.hasNext() can be called from different threads and is not synchronized.
AbstractList.java:606: Method java/util/ListIterator.hasNext() can be called from different threads and is not synchronized.
AbstractList.java:606: Method java/util/ListIterator.hasNext() can be called from different threads and is not synchronized.
AbstractList.java:608: Method java/util/ListIterator.next() can be called from different threads and is not synchronized.
AbstractList.java:607: Method java/util/ListIterator.next() can be called from different threads and is not synchronized.
AbstractList.java:605: Method java/util/List.listIterator() can be called from different threads and is not synchronized.
```

Warning: List operations are not synchronized

Generating Input Sequence

```
class Athread extends Thread{
    …
    public void run () {
       Y.add(some_object);
          ...
}
```

```
class Bthread extends Thread{
    …
    public void run () {
      bool test = X.equals(Y);
}
```

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
           …
   }
}
```

- Element of interest is Vector Y
- Athread adds an element to Y
- Bthread checks equality of X and Y

Generating Input Sequence

```
class Athread extends Thread{
    …
    public void run () {
       Y.add(some_object);
          ...
}
```

```
class Bthread extends Thread{
    …
    public void run () {
      bool test = X.equals(Y);
}
```

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
          …
   }
}
```

```
1  ListIterator y:= (List o).listIterator();
2  y.hasNext()
3  Y.add(some_object);
4  y.next();
```

Generating Input Sequence

```
class Athread extends Thread{
    …
    public void run () {
       Y.add(some_object);
          ...
}
```

```
class Bthread extends Thread{
    …
    public void run () {
      bool test = X.equals(Y);
}
```

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
          …
   }
}
```

```
1  ListIterator y:= (List o).listIterator();

2  y.hasNext()

3  Y.add(some_object);

4  y.next();
```

Generating Input Sequence

```
class Athread extends Thread{
    …
    public void run () {
       Y.add(some_object);
          ...
}
```

```
class Bthread extends Thread{
    …
    public void run () {
      bool test = X.equals(Y);
}
```

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
          …
   }
}
```

```
1  ListIterator y:= (List o).listIterator();

2  y.hasNext()

3  Y.add(some_object);

4  y.next();
```

Generating Input Sequence

```
class Athread extends Thread{
    …
    public void run () {
       Y.add(some_object);
          ...
}
```

```
class Bthread extends Thread{
    …
    public void run () {
      bool test = X.equals(Y);
}
```

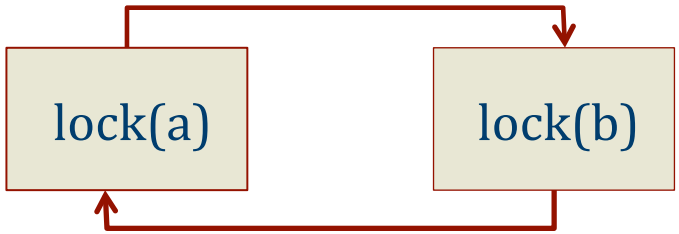
```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
          …
   }
}
```

```
1  ListIterator y:= (List o).listIterator();

2  y.hasNext()

3  Y.add(some_object);

4  y.next();
```

Input Sequence – Other Approaches

Cyclic Lock Dependencies

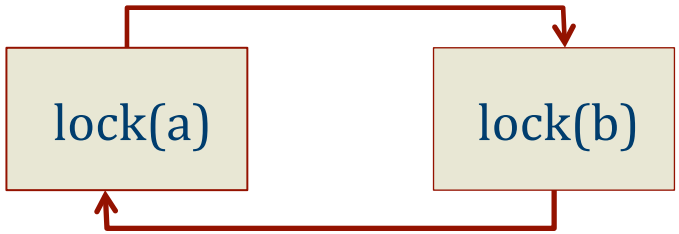
```
l0: lock(a) {
l1:  lock(b) {
     }
   }

l2: lock(b) {
l3:  lock(a) {
     }
   }
```



Input Sequence – Other Approaches

Cyclic Lock Dependencies

```
l0: lock(a) {
l1:  lock(b) {
     }
   }

l2: lock(b) {
l3:  lock(a) {
     }
   }
```



Input Sequence

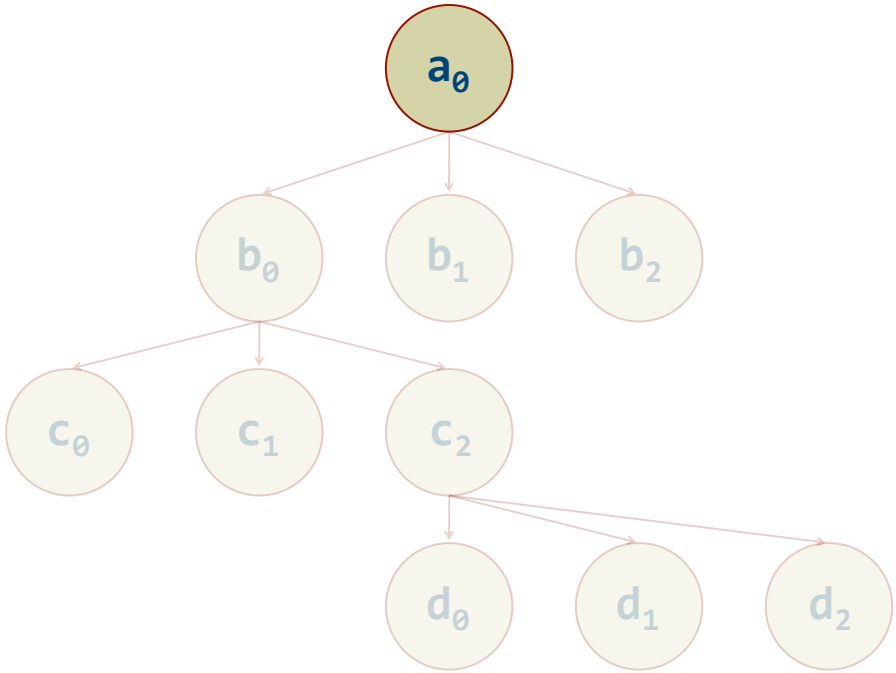
```
l0: lock(a)
l2: lock(b)
```

Input Sequence – Other Approaches

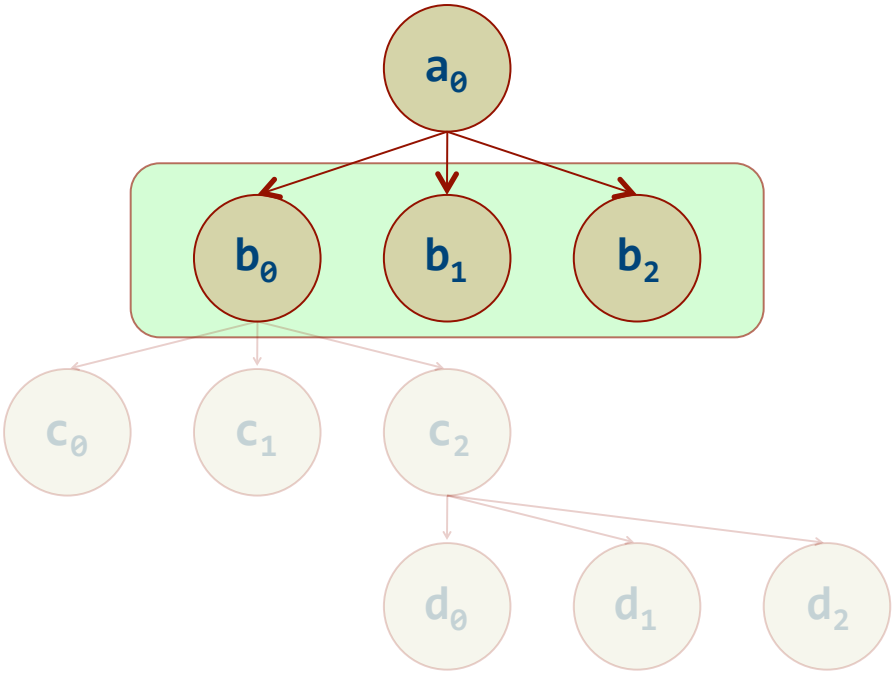
- Reachability properties
- Null-pointer exceptions
- Assert Statements
- Pattern-based Concurrency Errors (*Farchi et. al*)

- Guide concrete execution along the sequence of locations to test feasibility

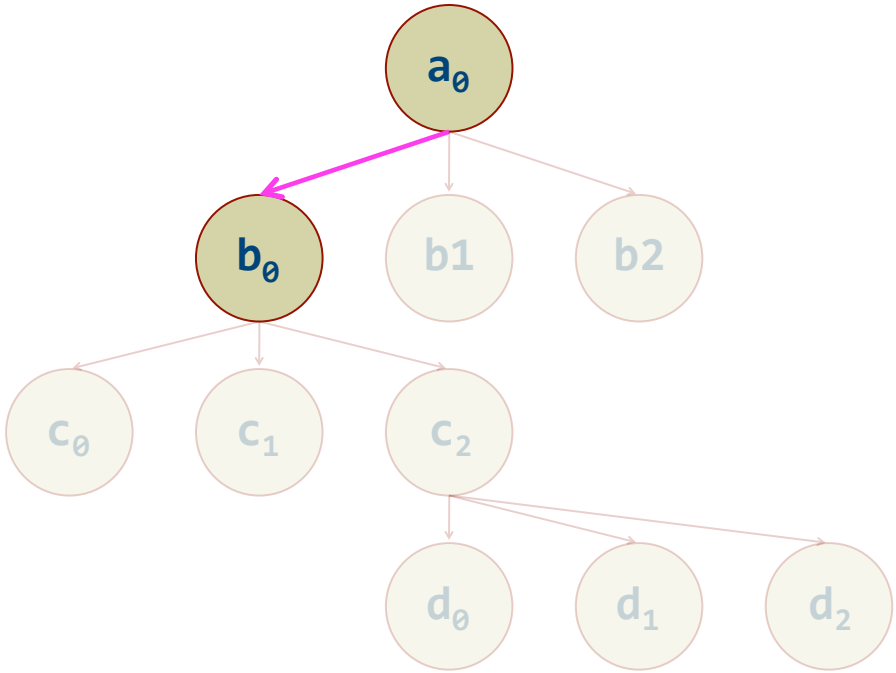
Greedy depth-first search



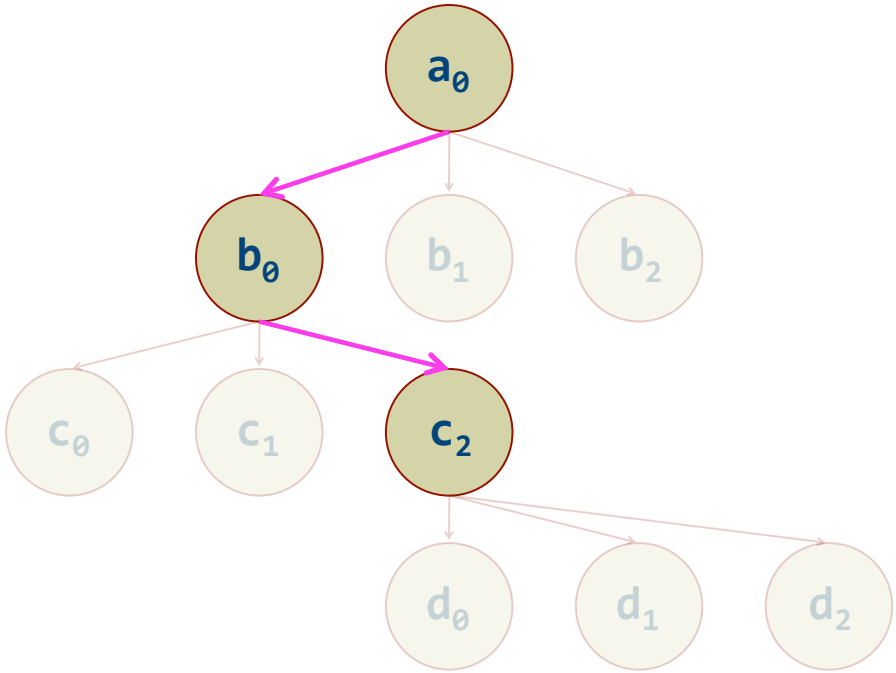
Greedy depth-first search



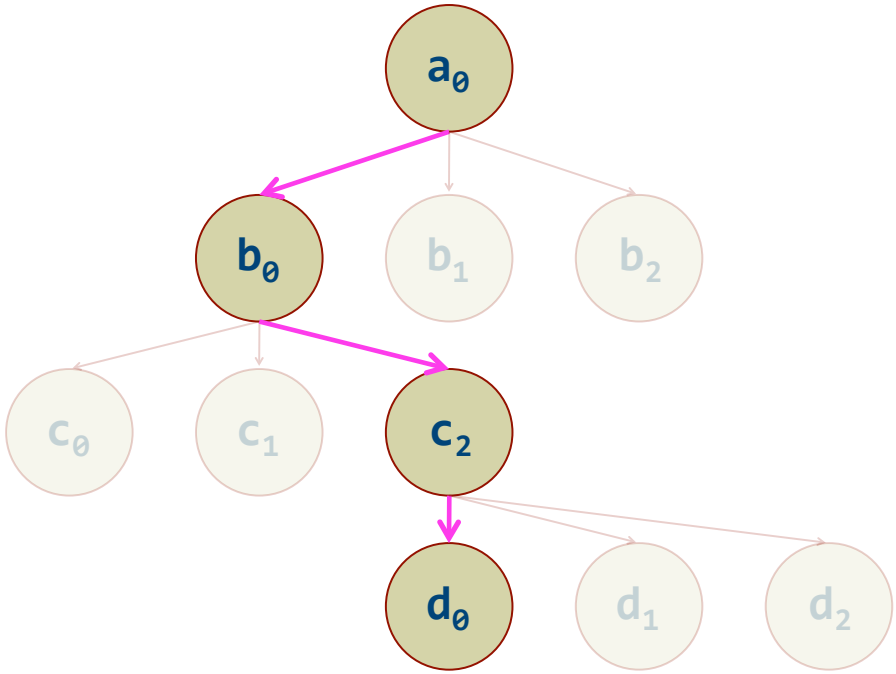
Greedy depth-first search



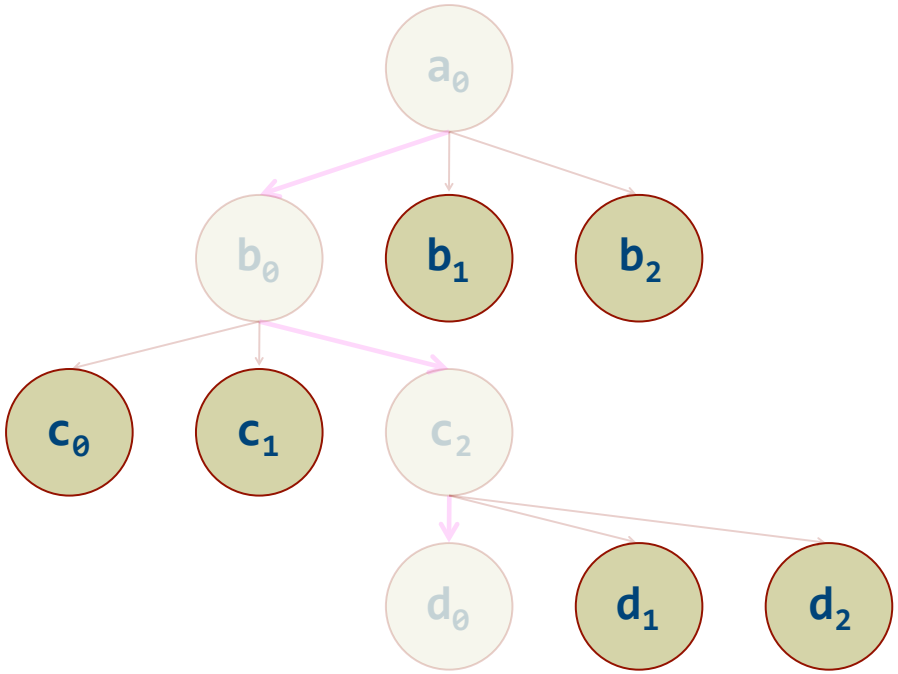
Greedy depth-first search



Greedy depth-first search



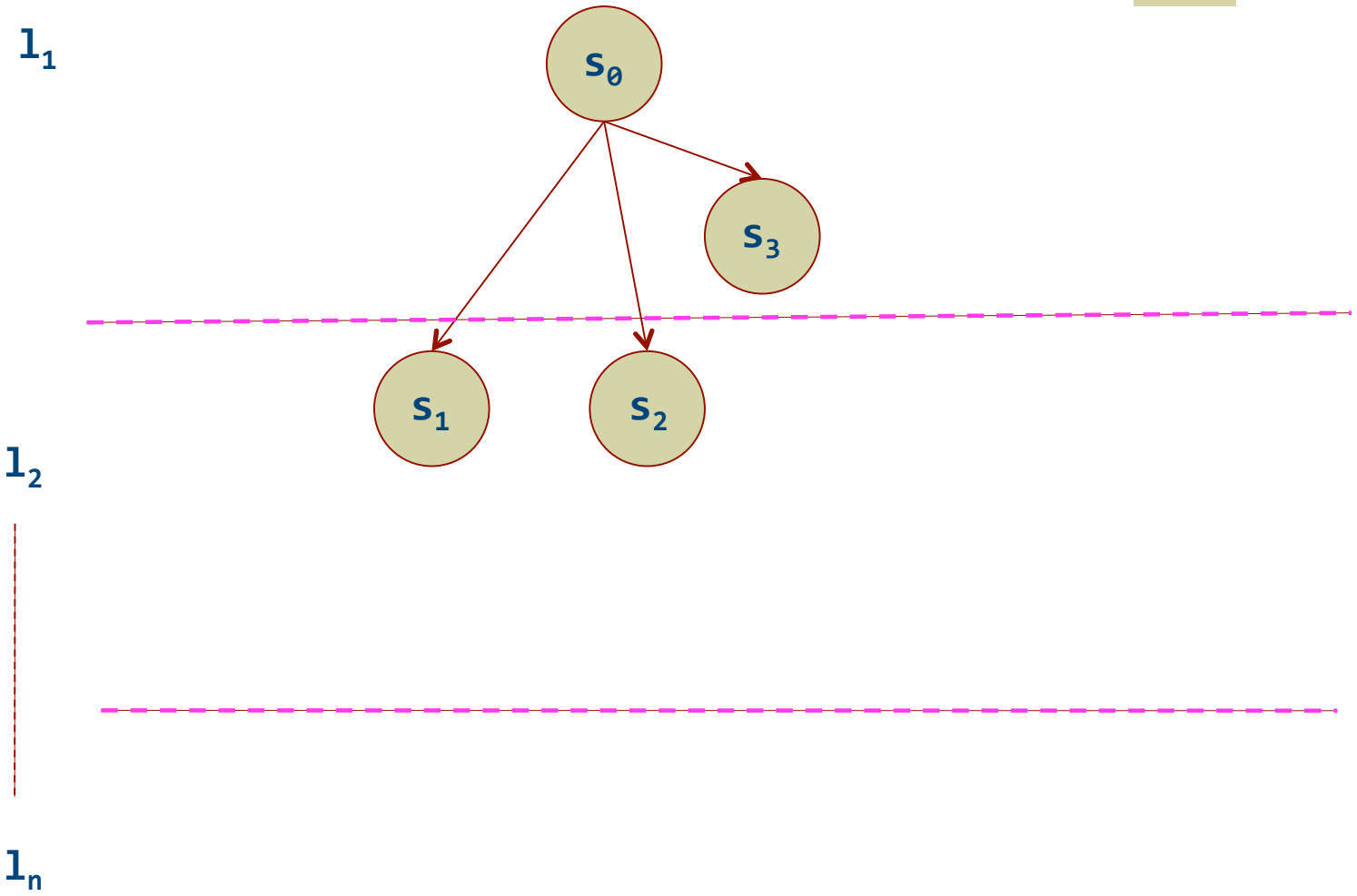
Greedy depth-first search



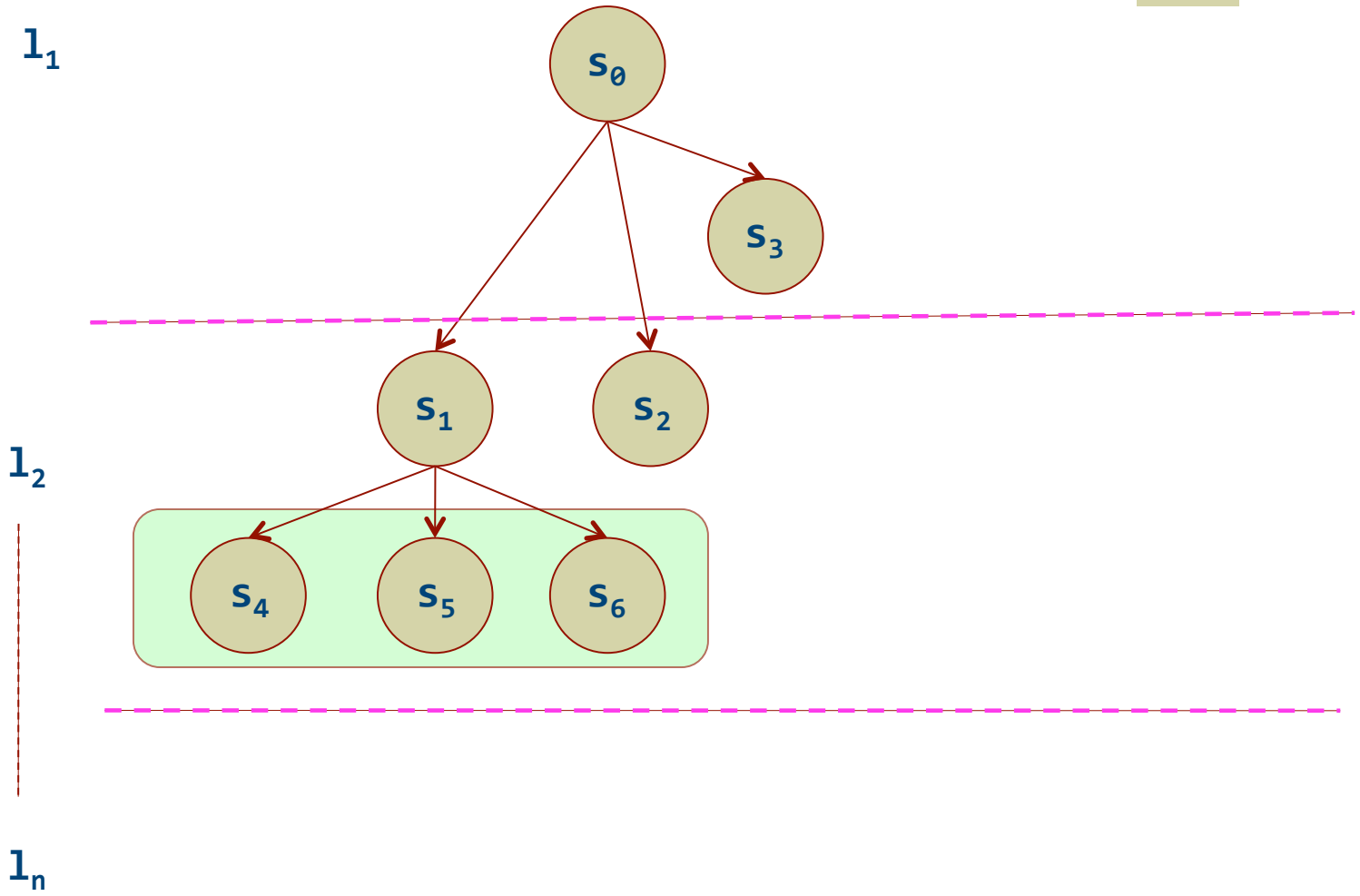
Guidance Strategy

- Two tier ranking scheme
- Rank states based on number of locations observed from input sequence
- Estimate the distance toward next location
- Various secondary heuristics
  - PFSM Heuristic (Rungta & Mercer)
  - Prefer-Thread (Visser & Groce)
  - Random Heuristic

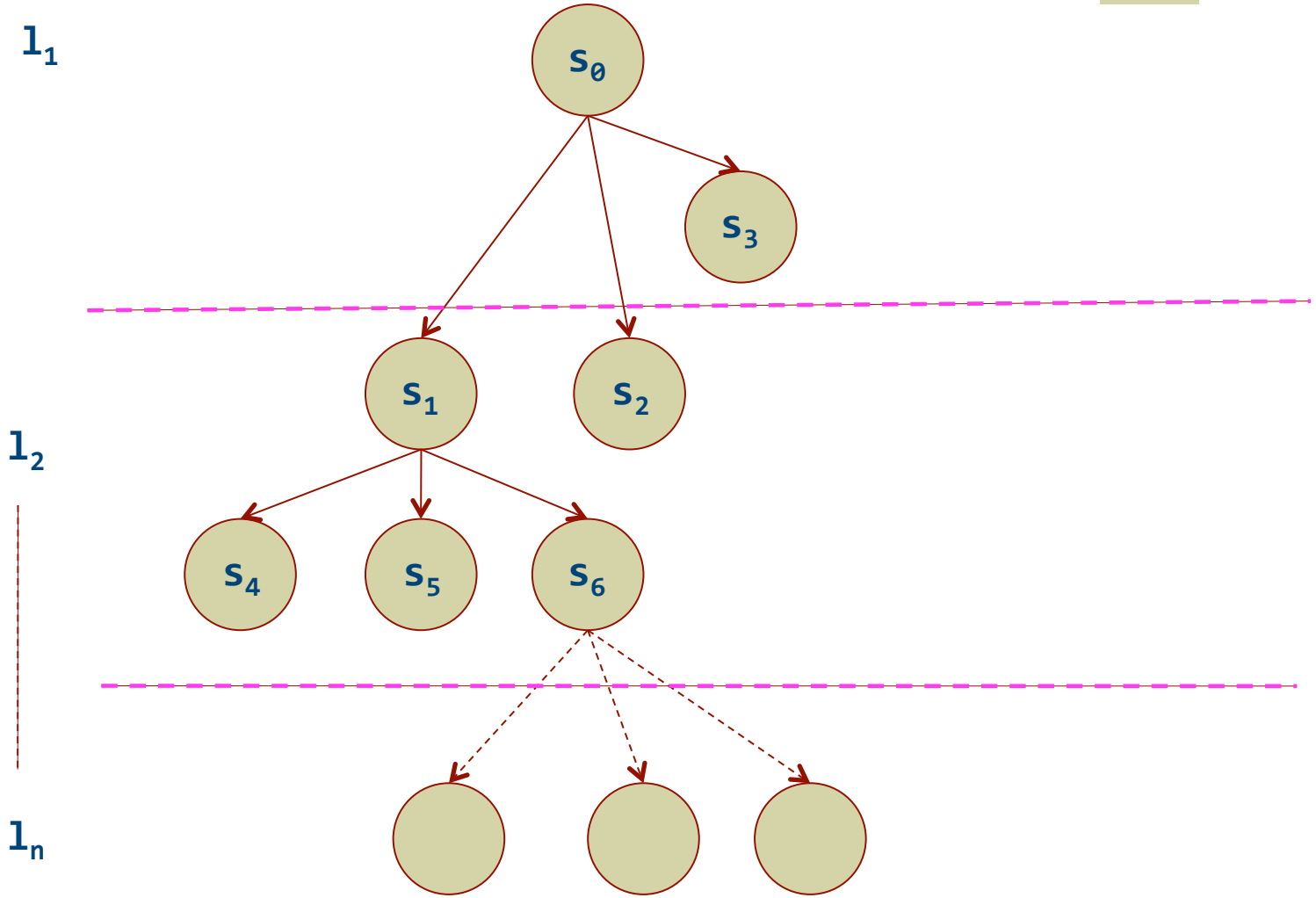
Guidance Strategy



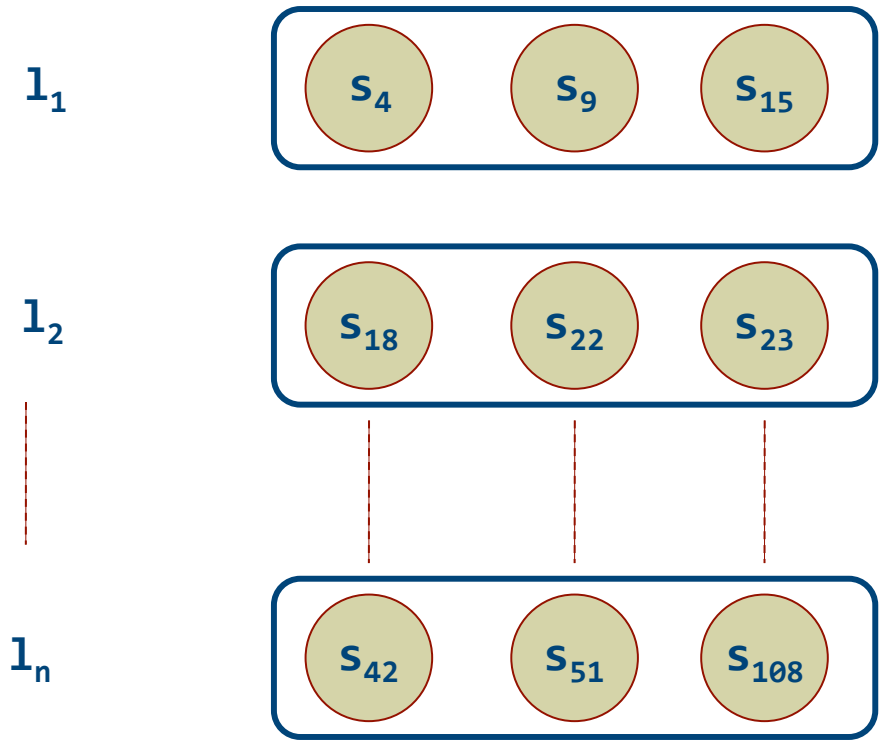
Guidance Strategy



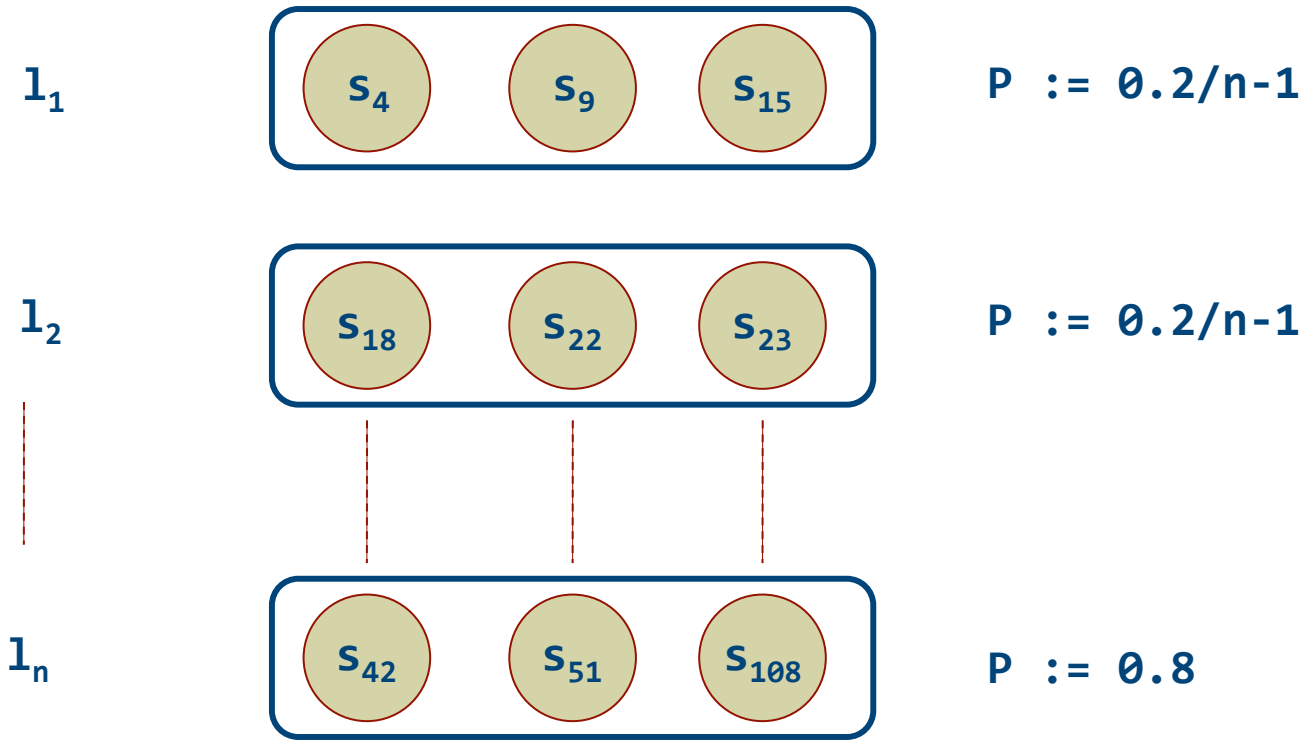
Guidance Strategy



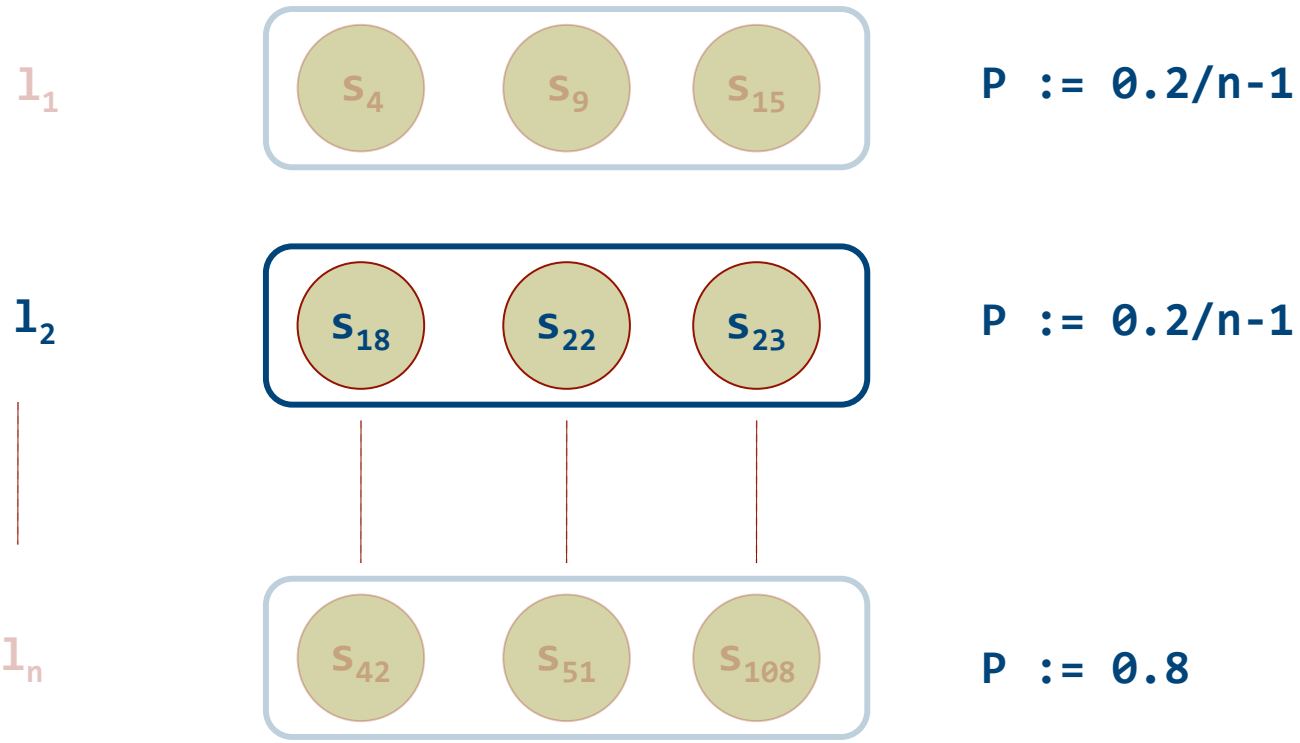
Backtracking



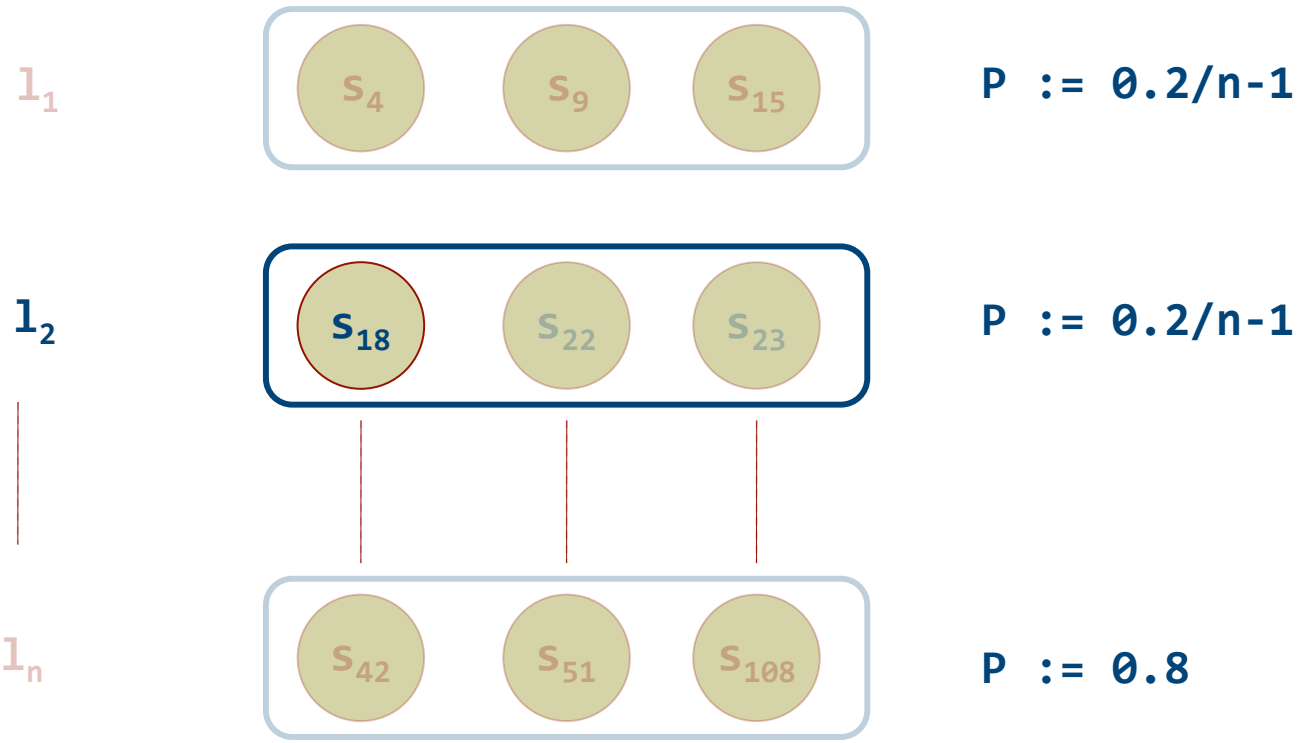
Backtracking



Backtracking



Backtracking



Empirical Evaluation

- 100 Greedy depth-first search and randomized depth search
- Heuristic ties are randomly broken
- Time bound of 1 hour
- Consistent with other recent studies
- Vary the underlying secondary heuristic
- Use JPF model checker as the test execution engine to run the experiments
- No data non-determinism

Examples

- TwoStage (SLOC: 52) – 2 types of threads
  - Two-stage access bug pattern
- AbsLst (SLOC: 7267) – 2 types of threads
  - Race Condition in the AbstractList Class
  - Synchronized Vector sub-class
- AryLst (SLOC: 7169) – 2 types of threads
  - Race Condition in the ArrayList Class
  - Synchronized List implementation

Defect Detection Rate

Subject	Random	Meta Heursitic		
	DFS	PFSM	Random	PreferThread
TwoStage(7,1)	0.41	1.00	1.00	1.00
TwoStage(8,1)	0.04	1.00	1.00	1.00
TwoStage(10,1)	0.00	1.00	1.00	1.00
AbsList(1,7)	0.01	1.00	0.37	0.00
AbsList(1,8)	0.00	1.00	0.08	0.00

Error Density = Error discovering trials ÷ Total Trials

Defect Detection Rate

Subject	Random	Meta Heuristic		
	DFS	PFSM	Random	PreferThread
TwoStage(7,1)	0.41	1.00	1.00	1.00
TwoStage(8,1)	0.04	1.00	1.00	1.00
TwoStage(10,1)	0.00	1.00	1.00	1.00
AbsList(1,7)	0.01	1.00	0.37	0.00
AbsList(1,8)	0.00	1.00	0.08	0.00

Defect Detection Rate

Subject	Random	Meta heuristic		
	DFS	PFSM	Random	PreferThread
TwoStage(7,1)	0.41	1.00	1.00	1.00
TwoStage(8,1)	0.04	1.00	1.00	1.00
TwoStage(10,1)	0.00	1.00	1.00	1.00
AbsList(1,7)	0.01	1.00	0.37	0.00
AbsList(1,8)	0.00	1.00	0.08	0.00

Race-condition in the AbstractList class

Counter-example

```
class AbstractList implements List{
    …
 public boolean equals (Object o) {
    ListIterator x:= ListIterator();
    ListIterator y:= (List o).listIterator();
    while x.hasNext() and y.hasNext()
       Object o1 := x.next();
       Object o2 := y.next();
           …
   }
}
```

```
class Vector extends AbstractList{
    …
 public synchronized boolean equals (Object o) {
     super.equals(o);
 }

public synchronized boolean add (Object o) {
     /* Add element to its own instance */
 }

}
```

X.equals(Y)

lock (X)

`ListIterator y:= (List o). listIterator();`

y.hasNext()

y.add(o)

lock(Y)

unlock(Y)

y.next()

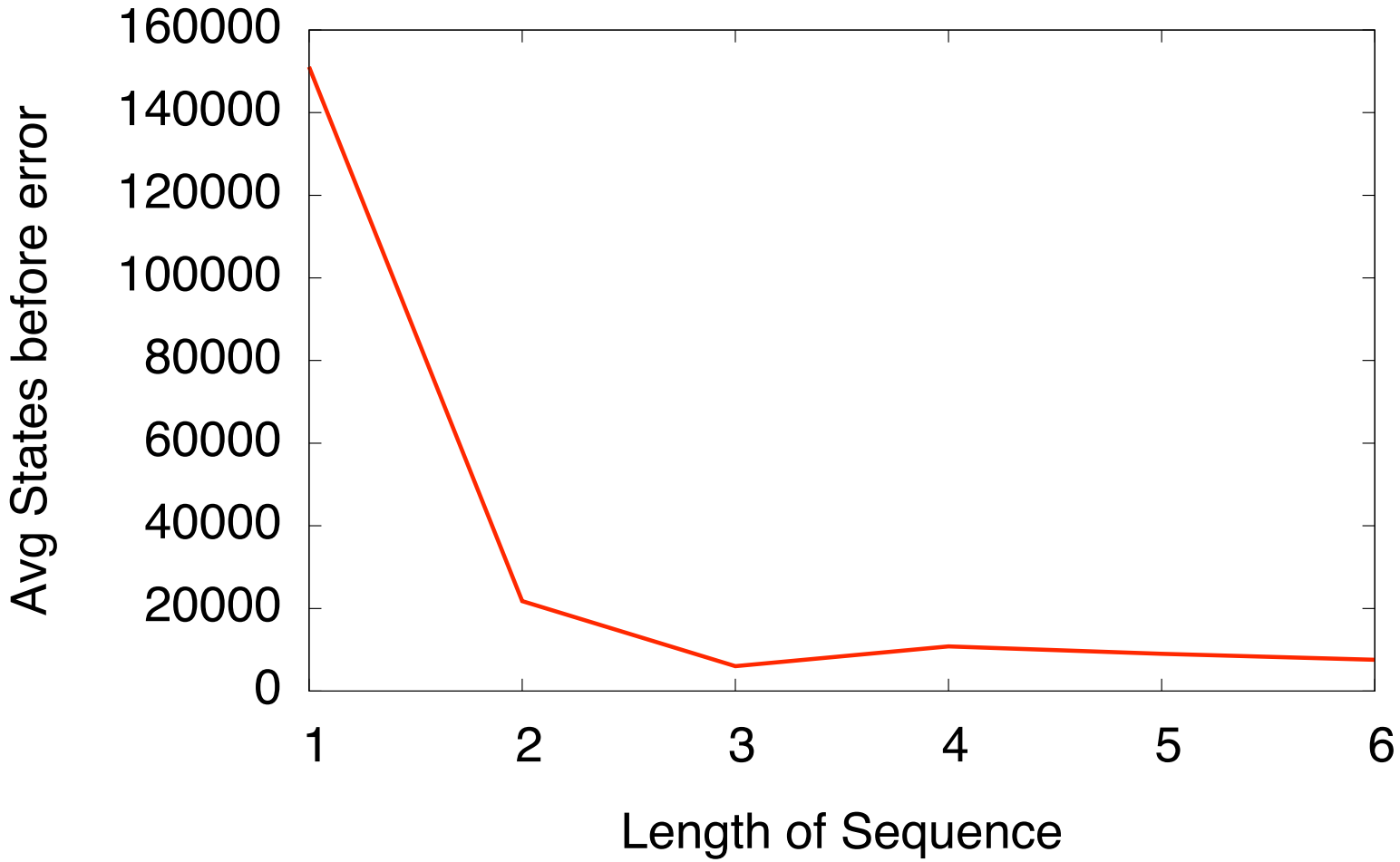
`ConcurrentModification Exception`

Avg. States Explored

Subject	Meta Heursitic		
	PFSM	Random	PreferThread
TwoStage(7,1)	213	130,839	2,206,109
TwoStage(8,1)	250	502,762	4,436,444
TwoStage(10,1)	333	827,830	6,690,008
AryList(1,10)	14,044	15,972	-
AbsList(1,10)	1,382	10,497,302	-

Average states generated in error discovering trials

Effect of varying the sequence length



Conclusion

- Meta heuristic reduces manual effort
- Leverages information of static analysis
- Generates a sequence of locations
- Guides concrete execution along sequence
- Effective for error discovery
- Discovered real errors in JDK 1.4 library

Work in Progress

- Automatic generation of traces
- Additional static analyses
  - Control and Dependence
  - Call Sequences
- Rank Data non-determinism
- Guided hybrid symbolic execution
- Refine traces during program execution
- Initial results are encouraging

Questions

Verification and Validation Lab
Computer Science Department
Brigham Young University
Provo, Utah

Neha Rungta: neha@cs.byu.edu
Eric G. Mercer: eric.mercer@byu.edu

http://vv.cs.byu.edu
http://vv.cs.byu.edu/~neha