Search Strategies for Model-Checking Concurrent Programs

Reed Milewicz (rmmilewi@uab.edu)
January 21st, 2016

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

Who am I?

- My advisor: Dr. Peter Pirkelbauer
- Our Lab: iProgress
- What we do…
  - Static Analysis
  - Source Code Transformation Systems
  - Runtime Systems and Continuous Optimizations
  - Software Verification Systems
  - Non-Blocking Software Design

Previously, you've covered…

- Uninformed search strategies (e.g. depth-first search)
- Informed search strategies (e.g. A* search)

Outline

- Concurrent Programming and Concurrency Bugs
- Model-Checking: State-space search to the rescue!
- Coping with State Space Explosion; Limitations of Current Approaches
- My Work

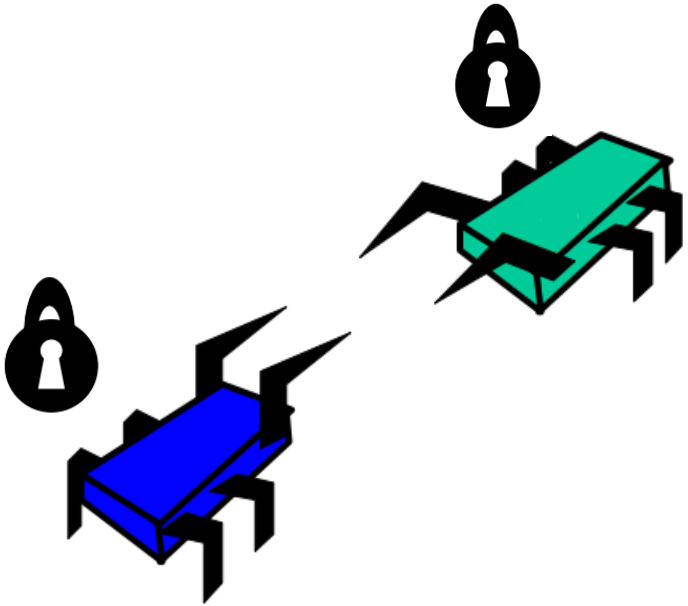
Concurrent Programming: A handful of terms

- A **thread** is a sequence of instruction blocks that are linked to one another by a chain of continuations.
- A multi-threaded program executing on a multi-core architecture consists of the birth, life, and dissolution of arbitrarily many threads operating in parallel across a fixed number of cores.
- These days, this is just called “programming.”

A handful of terms

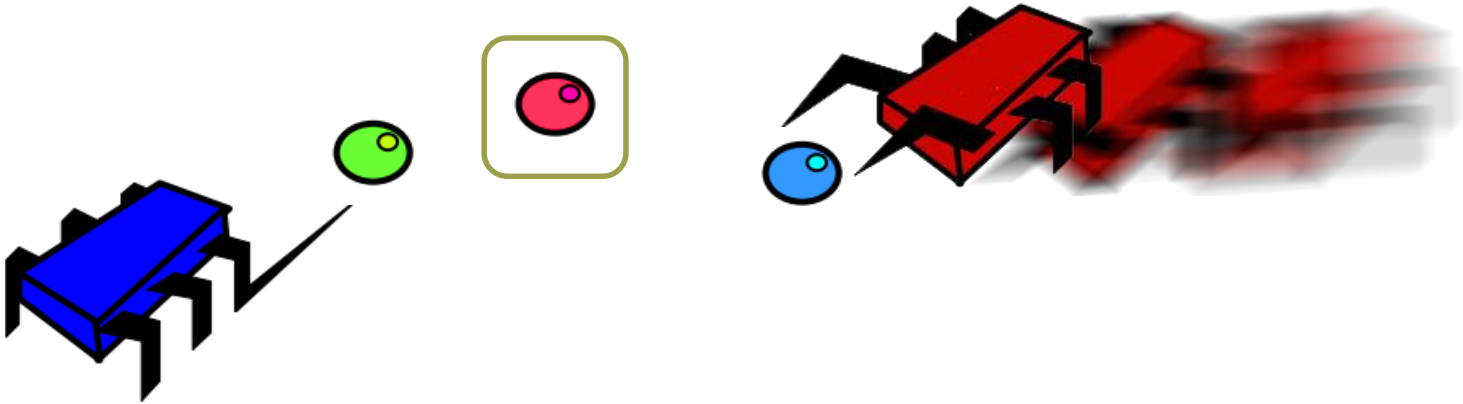
- **“Concurrency bug”** is an umbrella term for classes of bugs which arise as a consequence of improper synchronization between threads over the use of shared resources.
- These include…

Hazards



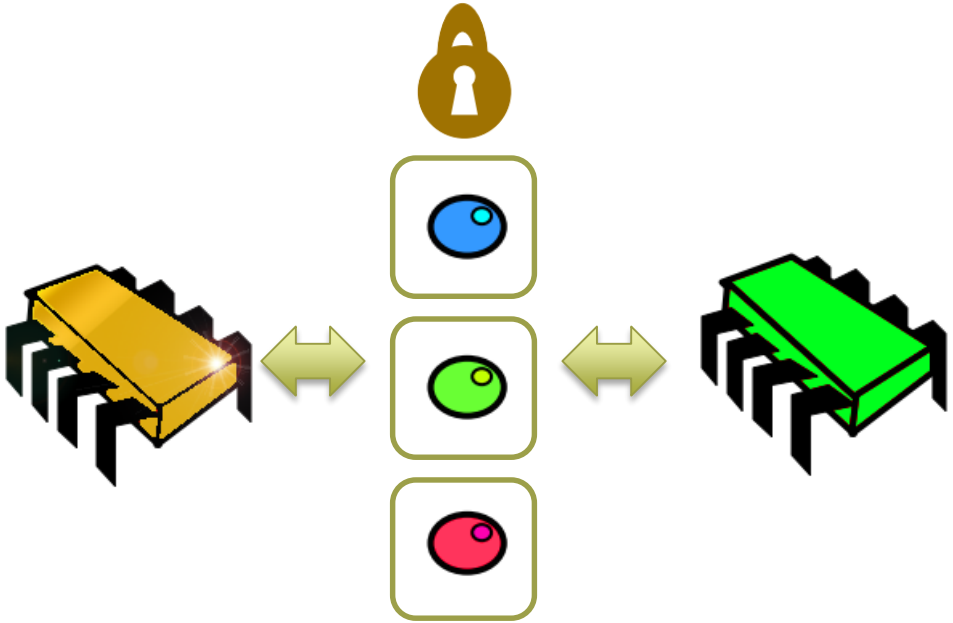
Deadlocks

Hazards



Race Conditions

Hazards



Atomicity Violations

Why are concurrency bugs challenging?

- Concurrency bugs are notoriously resistant to testing. They may only show up under very precise conditions.
- Their effects can be highly latent. Subtle memory corruptions, progressive paralysis of resources, etc.
- There can be many concurrency bugs in a program, some benign, some malignant. It’s difficult to tell them apart.

Why should you care?

- More than ever, the quality of our lives is determined by the quality of our software.
- Advancements in verification technology are vital to **the long-term survival of our species**.

How to deal with concurrency bugs…

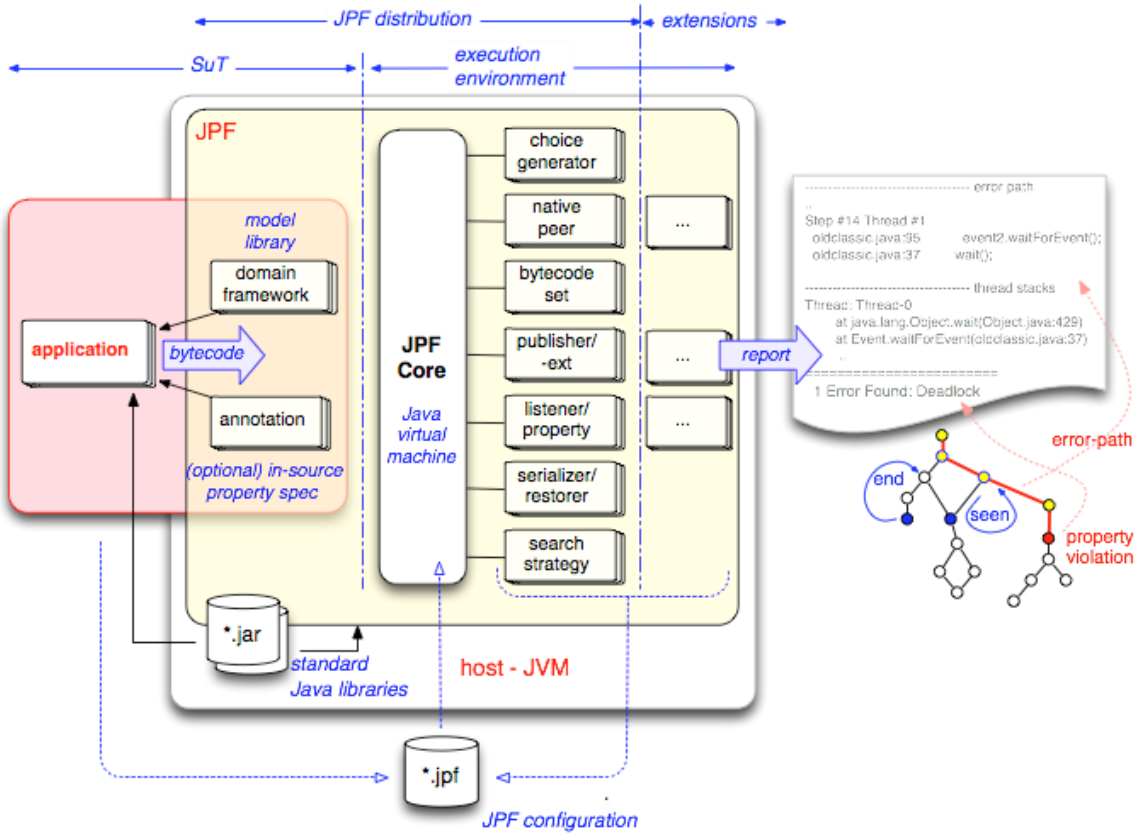
- There are many tools and techniques that we can use to detect and correct concurrency bugs.
- Today we will be focusing on **model-checking**.
- The Model-Checking Problem: Given a state-transition graph of a system and a specification, prove that the system conforms to or “models” that specification.

Model-checking

A model-checking engine…

- Automatically and exhaustively explores the state space of the system.
- Possible outcomes:
  - It verifies that all states satisfy the constraints of the specification.
  - It finds counterexample, a path from a start state to a state that violates a constraint.

Example: Java Pathfinder



THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

Model-checking: Kripke Structures

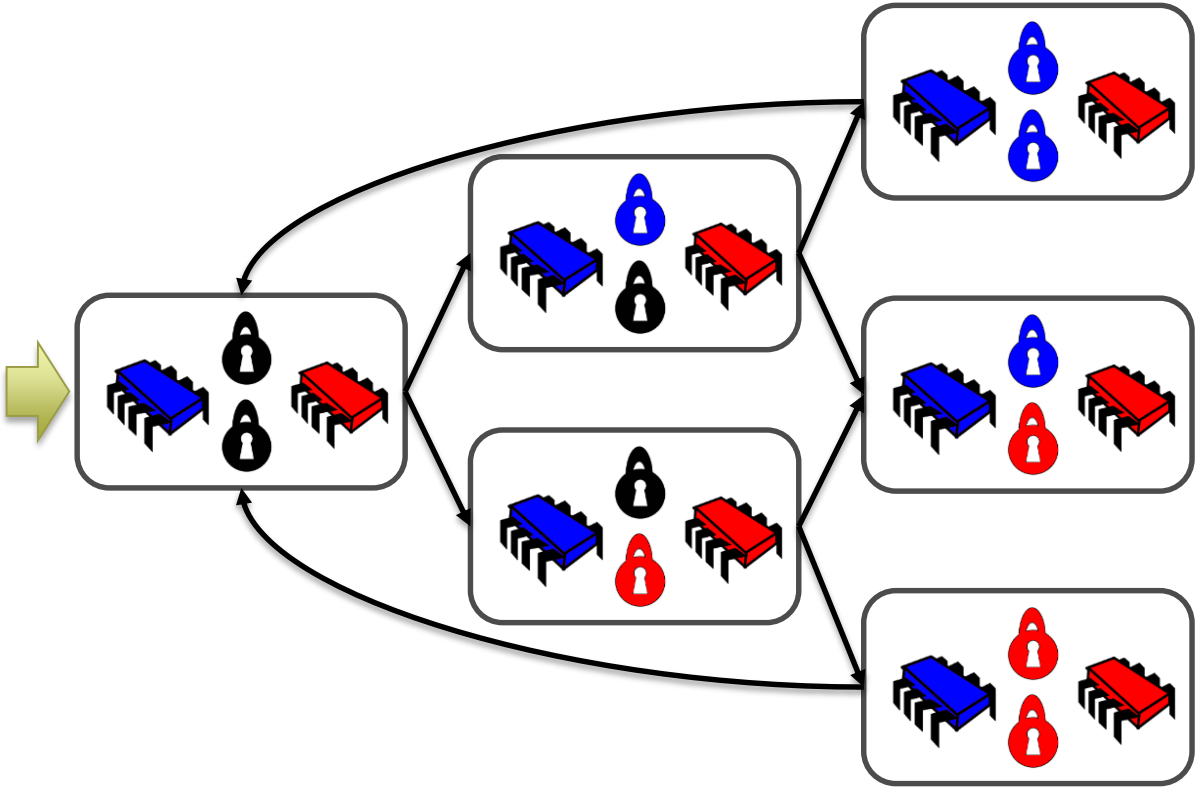
- A Kripke structure is a 5-tuple $\mathrm{M} = (S, I, R, P, L)$ where…
  - S is a finite set of states.
  - I is the set of initial states.
  - $R \subseteq S \times S$ is a transition function.
  - P is a set of atomic, Boolean propositions about the system that is modeled by M.
  - $L \subseteq S \times P$ is a labeling function that maps states in S to true/false values for propositions in P.

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

Example: Dining Philosophers

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

Dining Philosophers Program Modeled by Kripke Structure

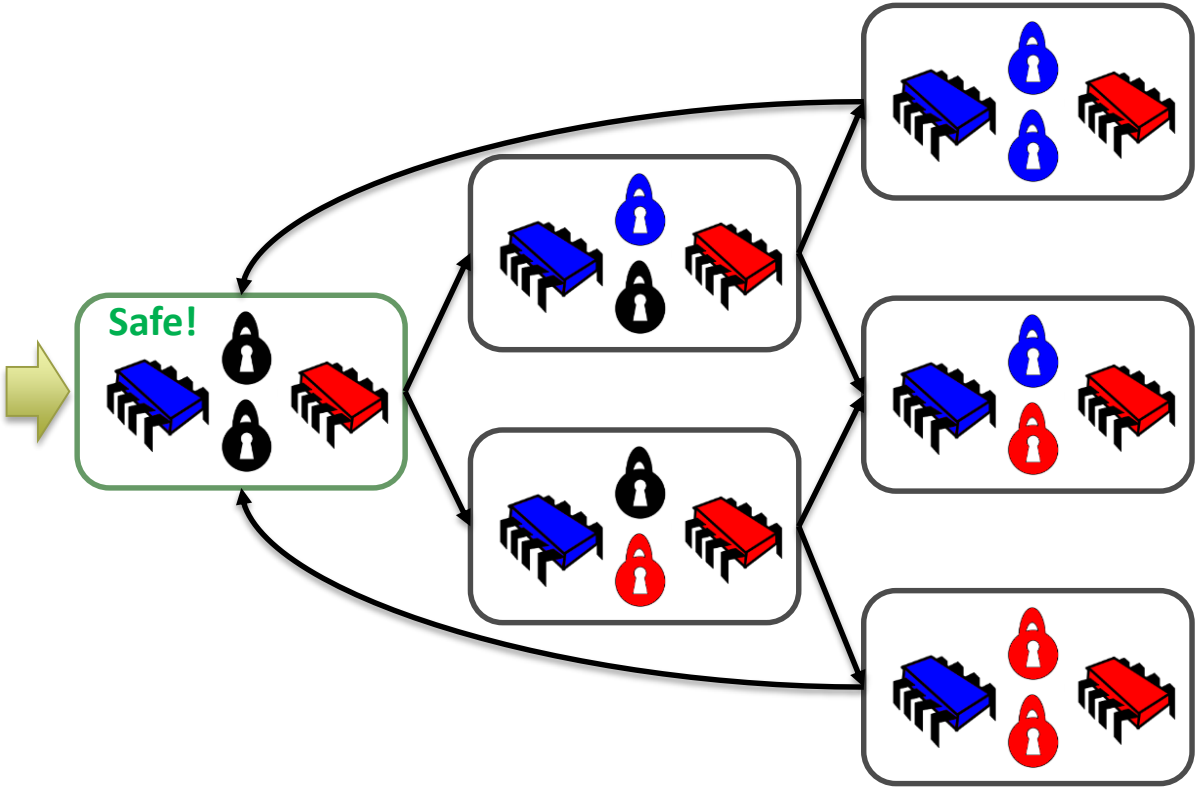


Now it’s a state space search problem!

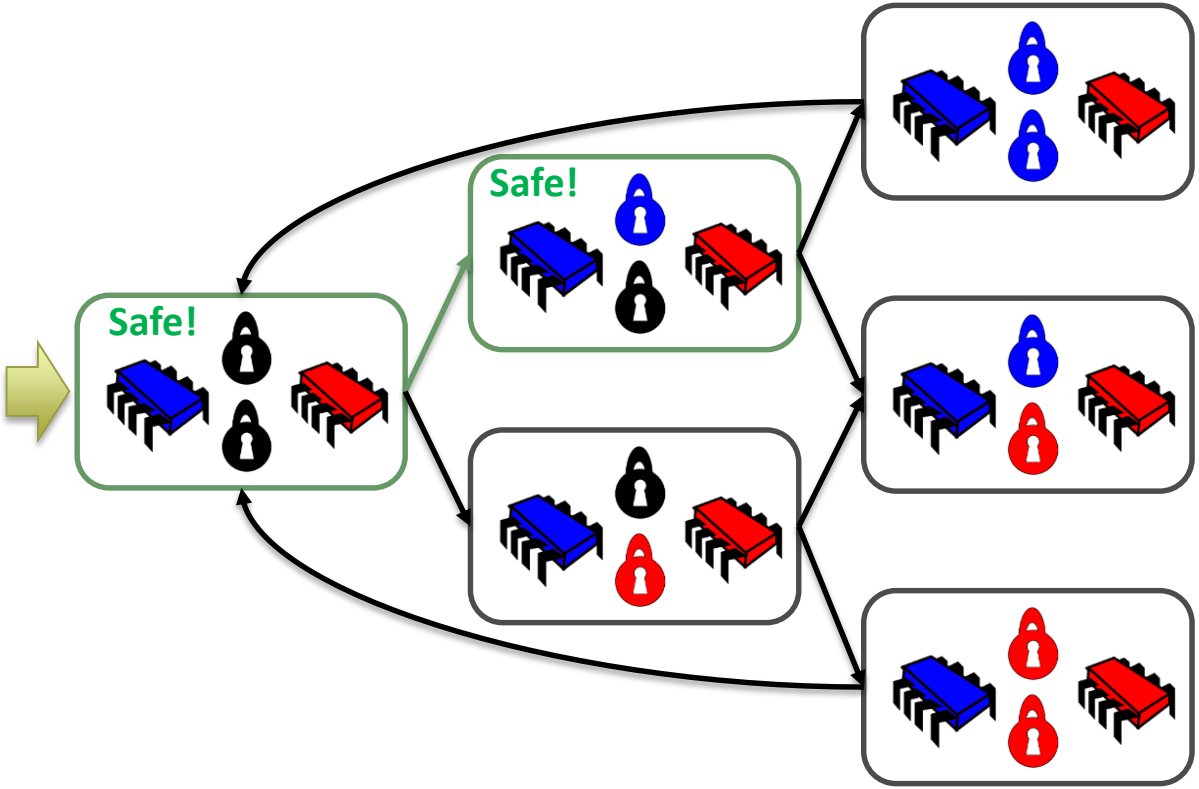
As a Search Problem, Model Checking is…

- **Fully Observable** or Partially Observable
- **Deterministic** or Stochastic
- Episodic or **Sequential**
- **Static** or Dynamic
- **Discrete** or Continuous
- **Single or Multi-Agent** (it depends on how you look at it)

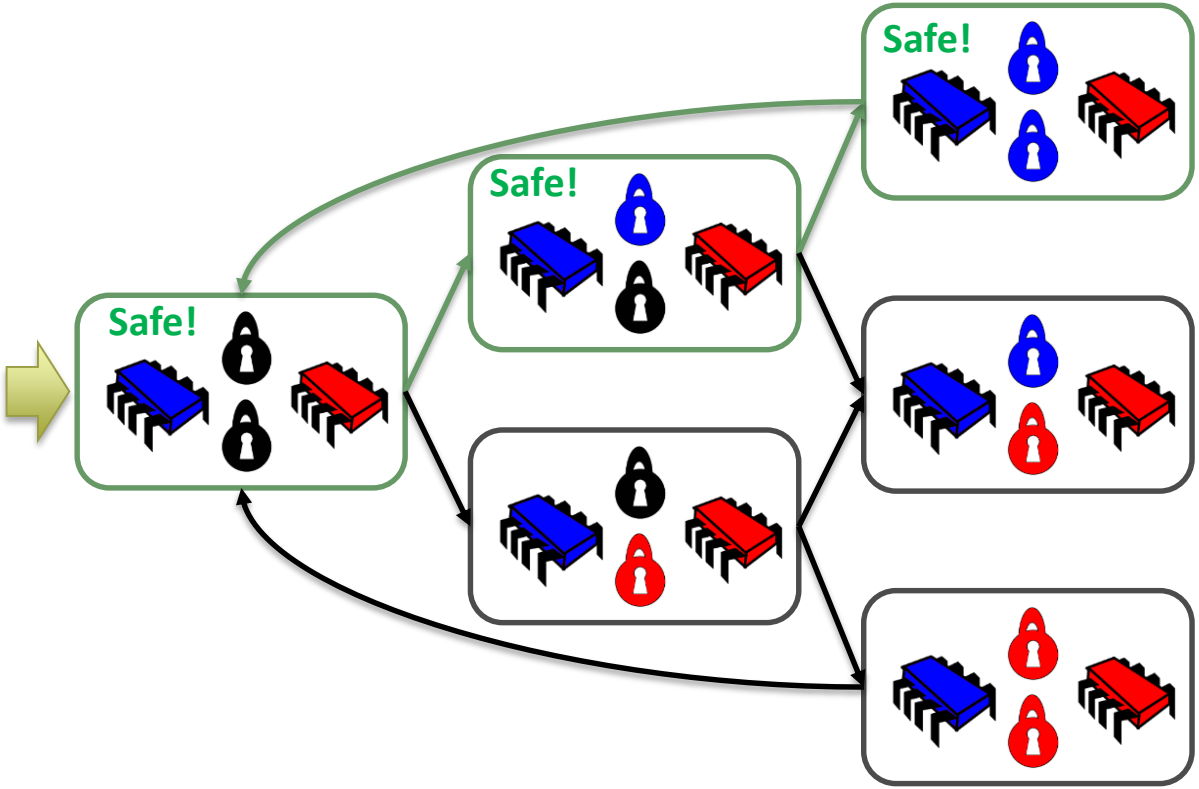
Depth-First Search on the State Space Representation



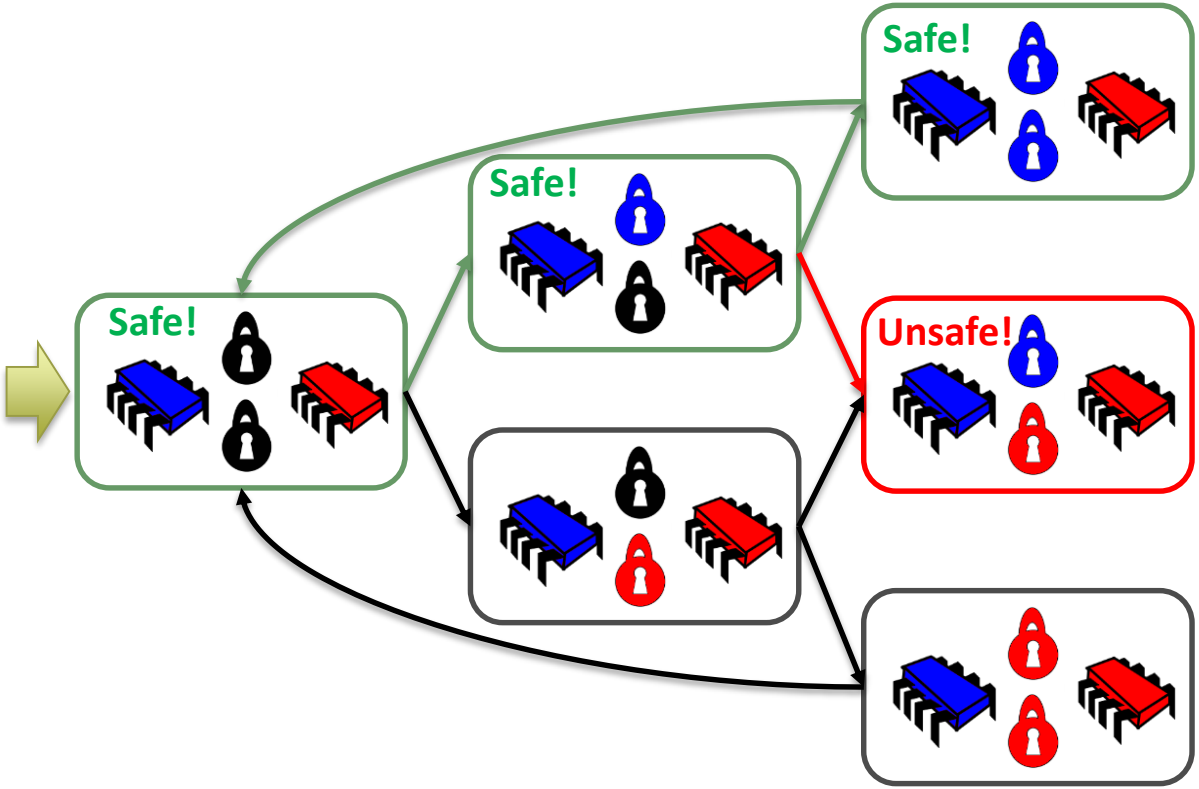
Depth-First Search on the State Space Representation



Depth-First Search on the State Space Representation



Depth-First Search on the State Space Representation



Property (Deadlock Freedom) Violated!

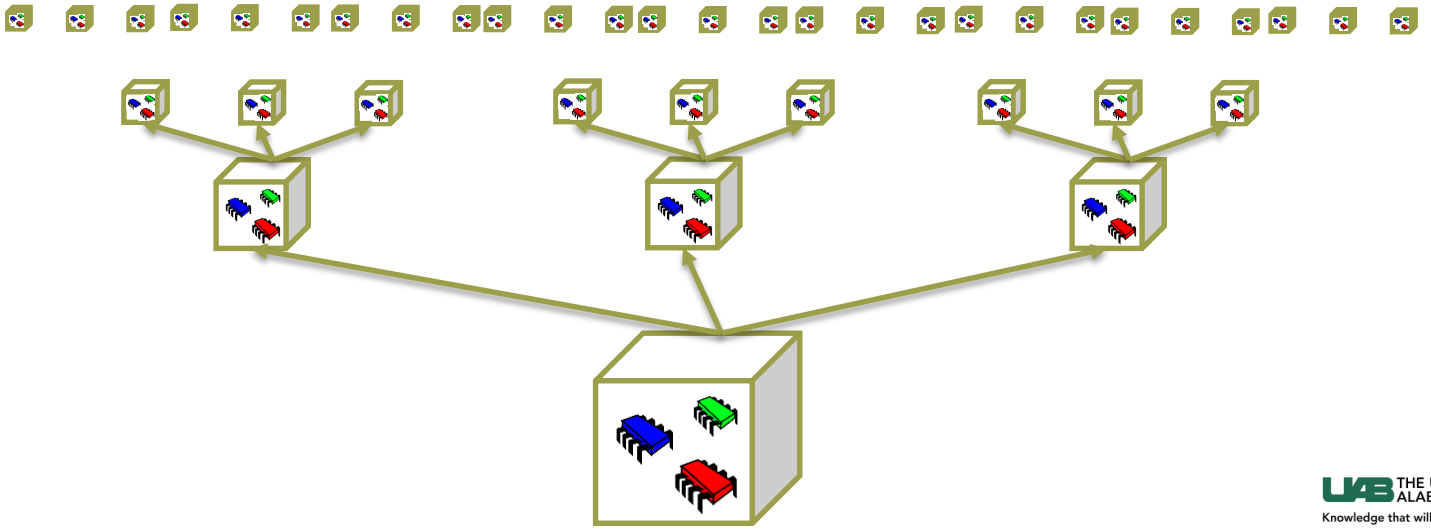
Counterexample:
1. Let blue grab left lock.
2. Let red grab right lock.

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

But how large can this state space be?

Number of Philosophers	Estimated Size of State Space (JPF)	If a state represented a 1 square meter of space…
1	13	Parking spot
2	170	Volleyball court
4	14516	Two soccer fields
8	1.001*10^8	Manhattan
16	4.753*10^15	9.3 x surface area of Earth
32	1.08*10^31	313 solar systems
64	5.5379*10^61	8.729 billion galaxies

Standing before the vast horizon of possible futures…



THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

How can we improve?

- Search less
  - **Partial order reduction:** The order of two actions by non-adjacent philosophers are independent and produce an identical state regardless of the order in which they were executed.
  - Almost all of the states in the dining philosopher's problem are mirrored. A more creative definition of state equivalence would prevent us from exploring redundant states.

How can we improve?

- Search smarter
  - Exhaustive search is impossible due to time and space constraints.
  - Using **heuristics** to help guide the search may allow us to find violations more efficiently. This brings us to **directed model checking**.

My recommendation…

h(state) =

Total # of philosophers

-

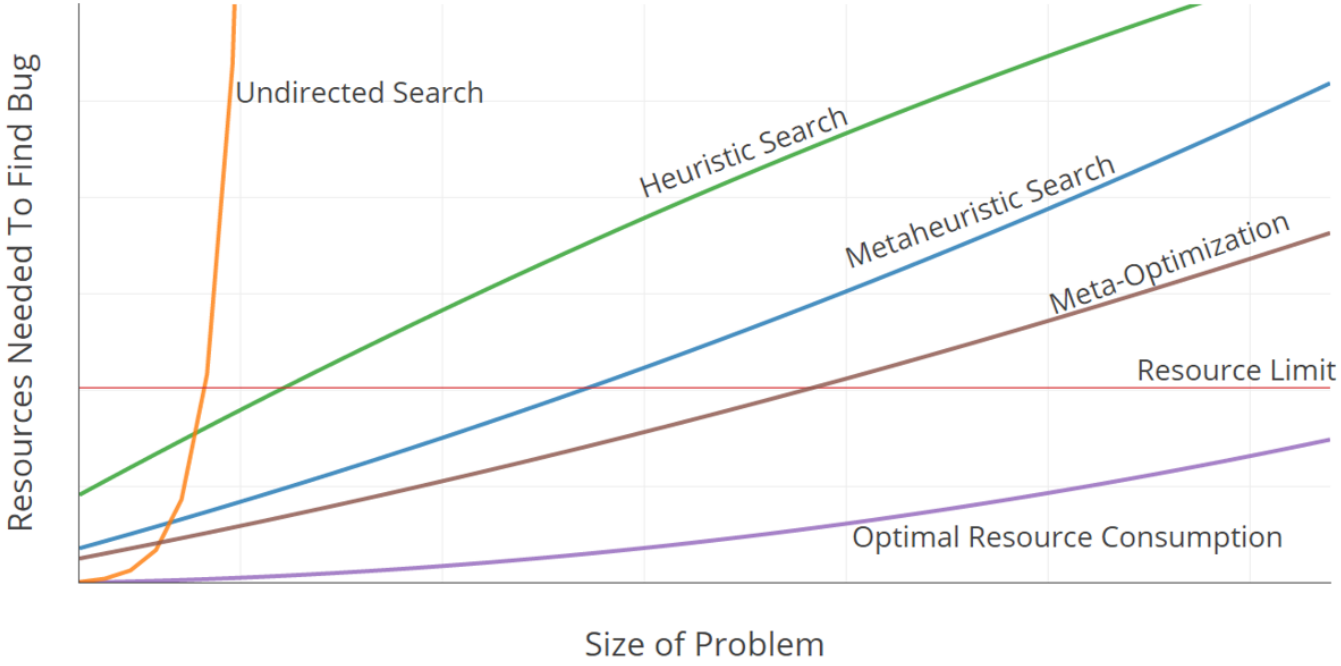
# of philosophers who hold no locks

Using our heuristic, we can find the bug quickly

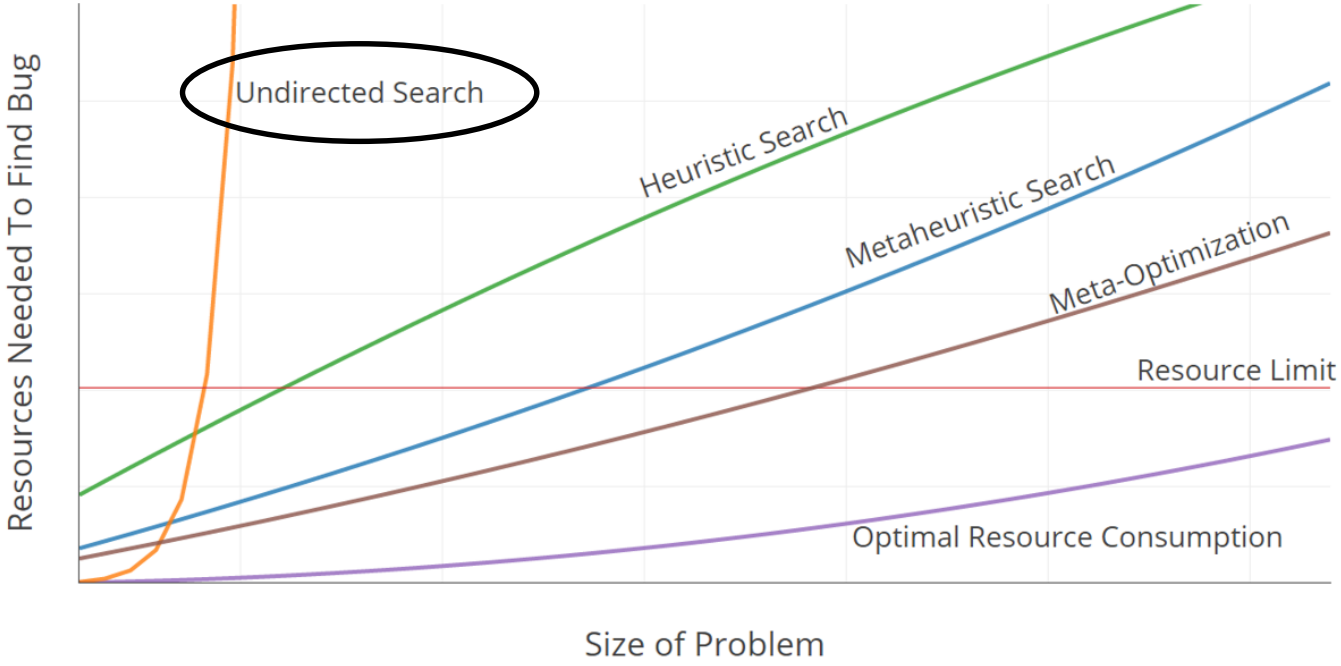
Number of Philosophers	Estimated Size of State Space (JPF)	Number of States Explored
1	13	N/A
2	170	23
4	14516	278
8	1.001*10^8	942
16	4.753*10^15	1278
32	1.08*10^31	1950
64	5.5379*10^61	3294

Model-checking

State of the Art for State Space Heuristics



State of the Art for State Space Heuristics



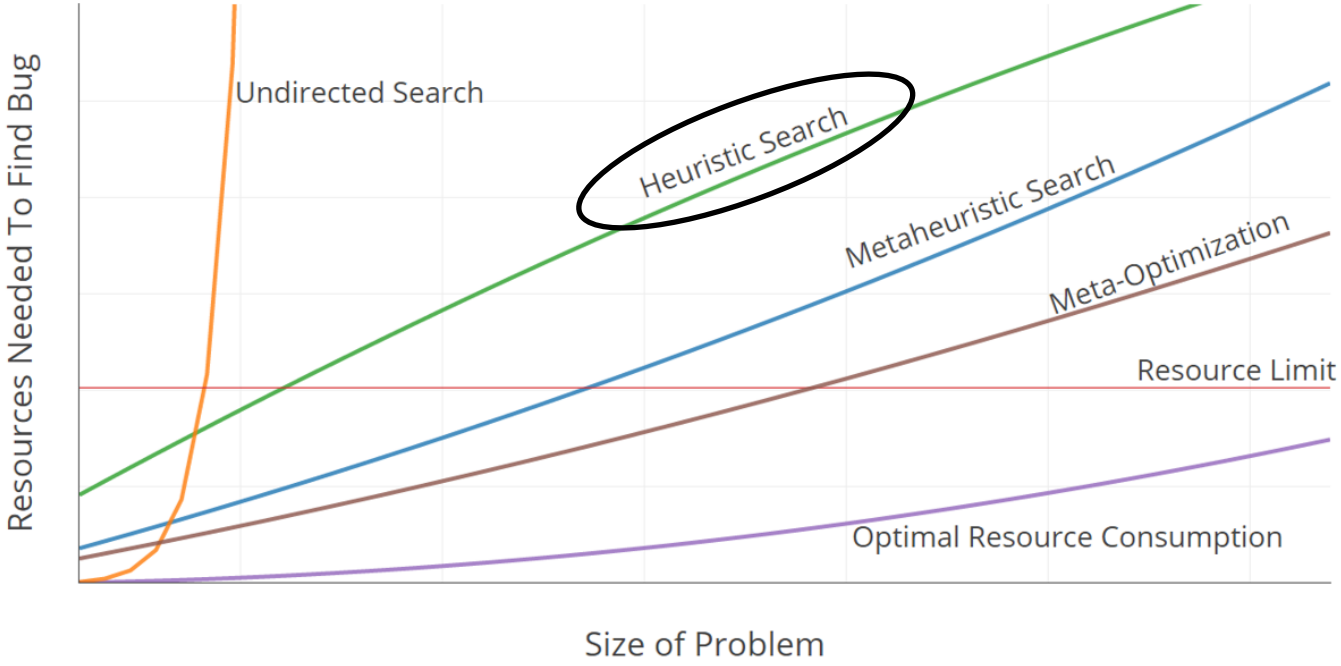
Examples

BFS

DFS

IDS

State of the Art for State Space Heuristics



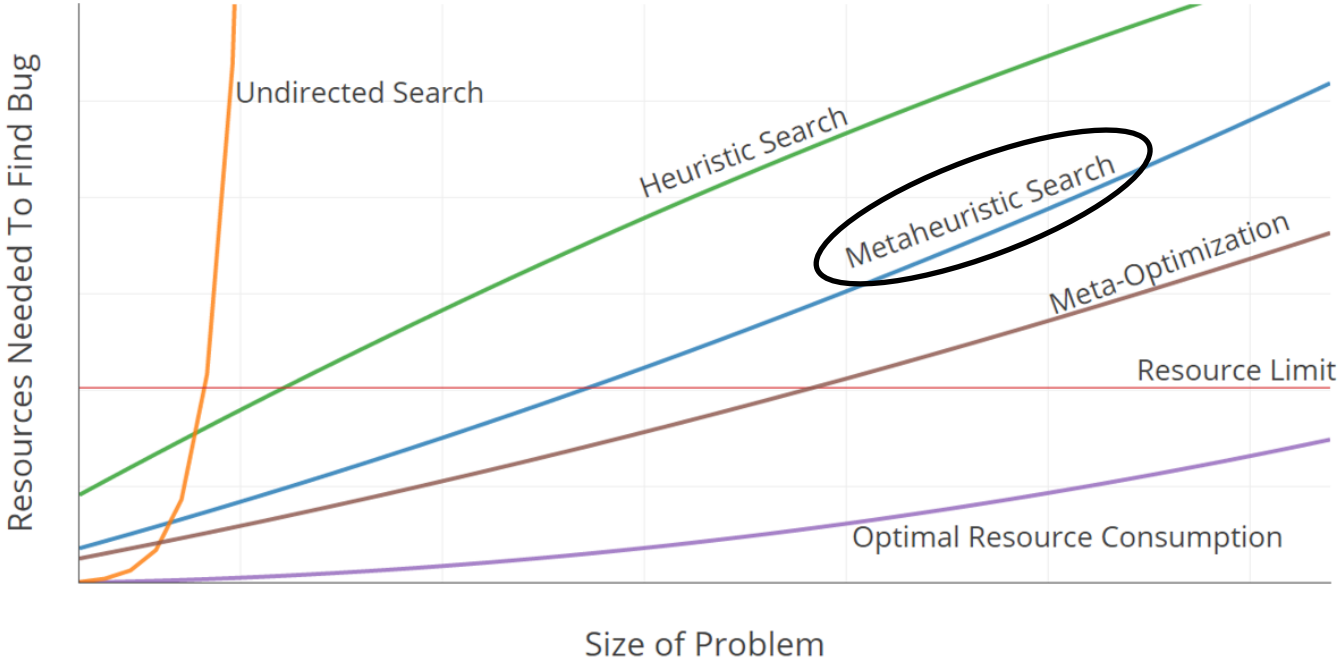
Examples

A*

Beam

(plus cost function)

State of the Art for State Space Heuristics



Examples

GA

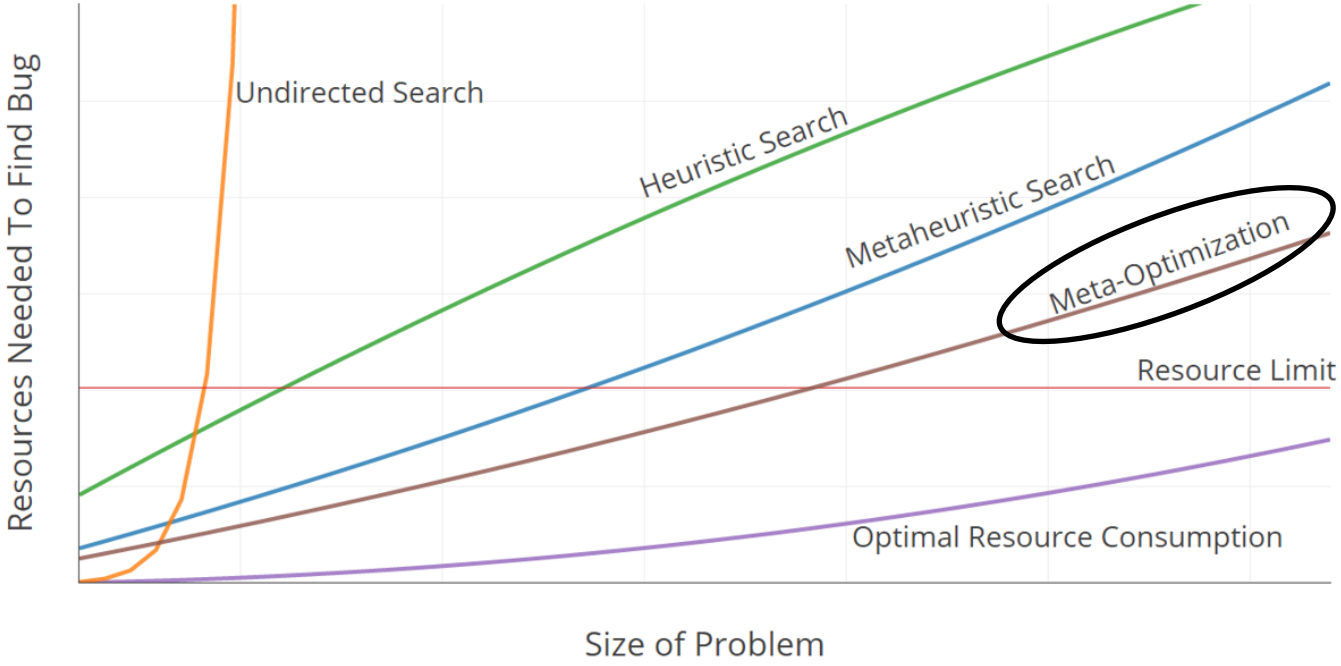
ACO

SA

EDA

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

State of the Art for State Space Heuristics



Examples

Param.Tuning

MGA

Limitations

- In this problem domain, optimality of an informed search strategy is algorithmically undecidable.
- For example, assume that A* plus a heuristic function H is optimal for finding bugs in a class of programs C.
- But for any program p, membership of p in C is algorithmically undecidable (see Rice's Theorem).

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

Limitations

- Metaheuristics are usually designed with optimization problems in mind.
- Limited transferability of heuristics from one problem to the next.
- Most meta-optimizations effectively require that we solve easier versions of a problem.
- But most importantly…

No Free Lunch Theorem

(Wolpert and Macready 1995)

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

No Free Lunch Theorem: An Analogy

- A group of friends enjoys going to eat at restaurants. They all have different dietary preferences.
- Every restaurant has an identical menu, but their prices may differ.
- Objective: Invent a decision process that finds an optimal selection of food that makes everyone happy and minimizes the average cost for each person.

No Free Lunch Theorem: An Analogy

- If the prices obey consistent patterns, we can exploit those patterns (e.g. the salad is always cheaper than the steak, which is usually cheaper than the lobster).
- But if the prices are uniformly random, then **it is impossible** to create such a decision procedure.

No Free Lunch Theorem

- Programming languages are Turing complete. The state spaces are both infinite and infinitely varied.
- For every program for which our chosen heuristic works well, we can construct another program for which our heuristic performs terribly.
- In short, **every** heuristic performs *just as well* as any other heuristic **on average**.

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

No Free Lunch Theorem

- But the distribution of real programs is non-uniform.
- Programs as purposeful artifacts
- Likewise, studies suggest that concurrency bugs are non-random occurrences.
- Therefore, the strength of our search algorithms depends upon **the strength of our priors**.

What do we know?

What do we know about concurrency bugs? (Lu et al. 2008)

- 96% of concurrency bugs involve only two threads.
- 66% of concurrency bugs involve concurrent accesses to only one variable.
- 92% of concurrency bugs involve no more than four memory accesses.
- 97% of non-deadlock concurrency bugs are either atomicity violations or order violations.
- 97% of deadlock concurrency bugs involve two threads circularly waiting for at most two resources.

What do we know about human programmers?

- The design of the program, made visible to us through the source code, is undergirded by a “typology of schemas” (Detienne 1990).
- Bugs are mismatches between the intended and actual semantics of the program.
- Even if programmers did have the knowledge we needed, getting that knowledge would be burdensome (Engler and Musuvathi 2004).

What do we know about programs?

- They’re multi-modal! We can reverse engineer the priors we need by studying…
  - Source Code
  - Bytecode
  - Execution traces
  - Comments/Documentation
  - Formal specifications

My work

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

My work

- Data mining execution traces to inform structural heuristics
- Hybridizing static analysis with model checking
- Substituting static analysis with machine-learned approximations

Refinement of Structural Heuristics for Model Checking of Concurrent Programs through Data Mining

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

Structural Heuristics (Groce and Visser 2002/2004)

- Structural heuristics are a highly generalizable/transferable class of heuristics that aim to be agnostic to the specifics of the program.
- Examples:
  - Take as few/many branches as possible.
  - Schedule as few/many threads as possible.
  - Have as many threads as possible be running/blocked/suspended/etc.

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

Structural Heuristics (Groce and Visser 2002/2004)

$$h_{ti}\,(path, limit) = \sum_{i=path.length-limit}^{path.length} \begin{cases} path.length * N_{aliveThreads} & \text{if } path[i].tid == path[i-1].tid \\ 0 & \text{otherwise} \end{cases}$$

- Maximizes the number of different interleavings of threads that we explore.

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

Problem: Generalizability vs. Scalability

- Assume we have a state space of a program P that we are exploring with A^* and h_{ti} with a history limit of L. P consists of a set of threads $thrds$. Assume that $L < \|thrds\|$.

Problem: Generalizability vs. Scalability

$thrds = \{\text{Alice}, \text{Bob}, \text{Cathy}, \text{Dave}\}$

$L = 3$

$$h_{ti}([\text{Alice}, \text{Bob}, \mathbf{Dave}, \mathbf{Alice}, \boldsymbol{Alice}], 3) = 0 + 5 * 4 = 20$$

$$h_{ti}([\text{Alice}, \text{Bob}, \mathbf{Alice}, \mathbf{Alice}, \boldsymbol{Bob}], 3) = 0 + 0 = 0$$

$$h_{ti}([\text{Alice}, \text{Bob}, \mathbf{Bob}, \mathbf{Alice}, \boldsymbol{Cathy}], 3) = 0 + 0 = 0$$

- There are up to $\|thrds\|^L$ different histories of length L, and of these, there are $\|thrds\| * (\|thrds\| - 1)^{L-1}$ histories where the most recent thread has been scheduled just once (i.e. where $h_{ti}\ (path, L) = 0$)

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

Problem: Generalizability vs. Scalability

- For a fixed L, we know that…

$$\lim_{\|threads\|\to\infty} \frac{\|thrds\| * (\|thrds\| - 1)^{L-1}}{\|thrds\|^{L}} = 1$$

- which means that almost all paths are optimally interesting at high thread counts.
- Increasing L to compensate has the effect narrows and deepens the search, which can cause us to miss violations at shallow depths.

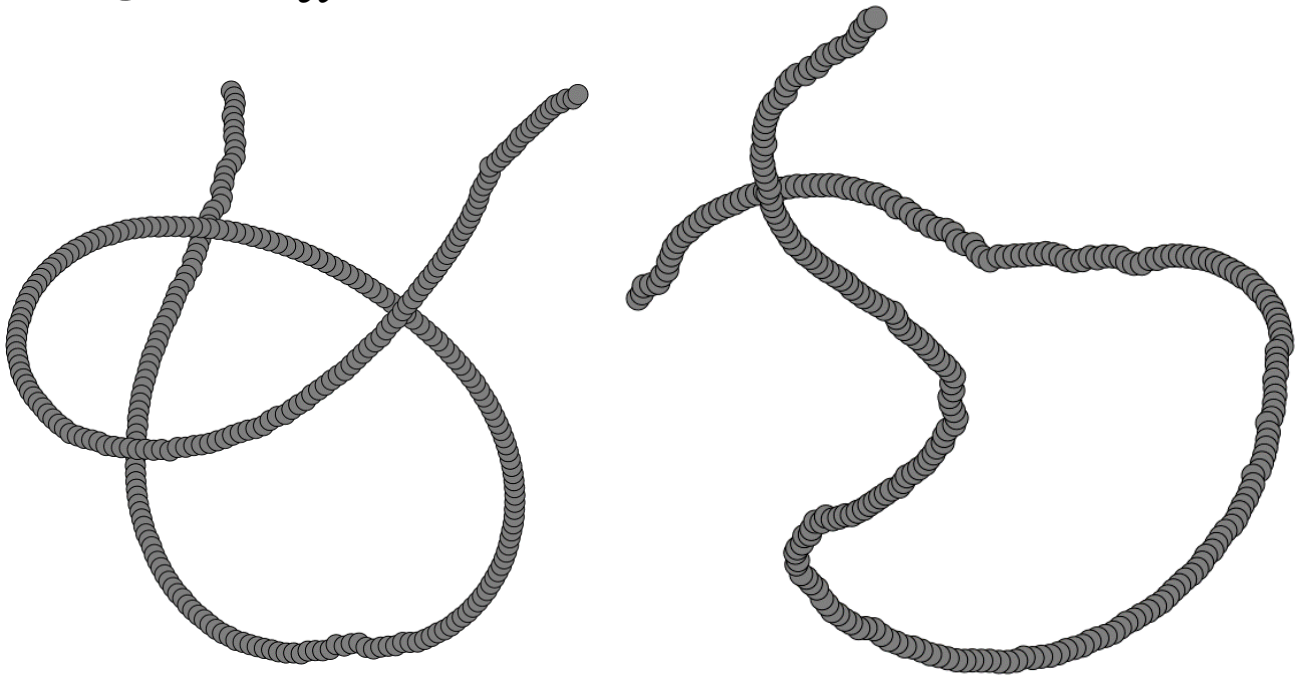
Specializing the h_{ti} Heuristic

- Assume we have a producer-consumer queue. It has a race condition bug in it somewhere.
- 96% of concurrency bugs are 2-thread bugs, and there are three possible flavors:
  - Producer/Producer
  - Producer/Consumer
  - Consumer/Consumer

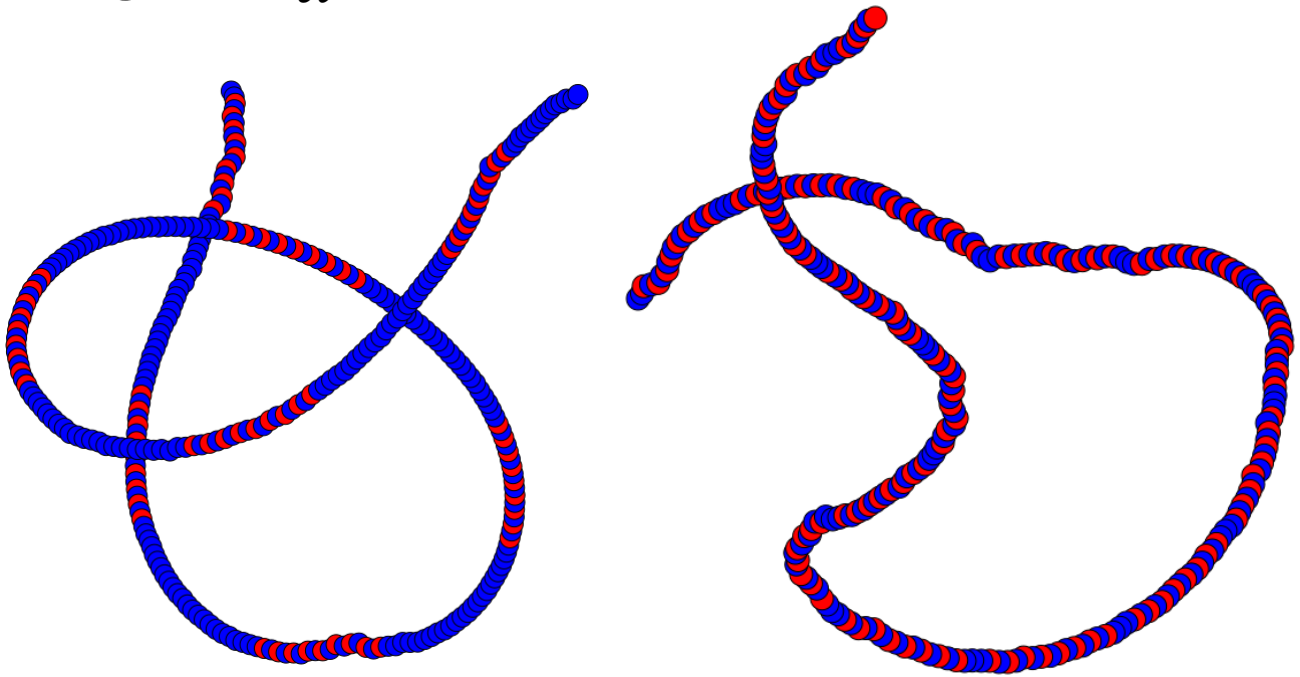
Specializing the h_{ti} Heuristic

- Two mutually exclusive hypotheses:
  - The bug occurs between threads executing different code (producer/consumer)
  - The bug occurs between threads executing the same code (producer/producer or consumer/consumer)

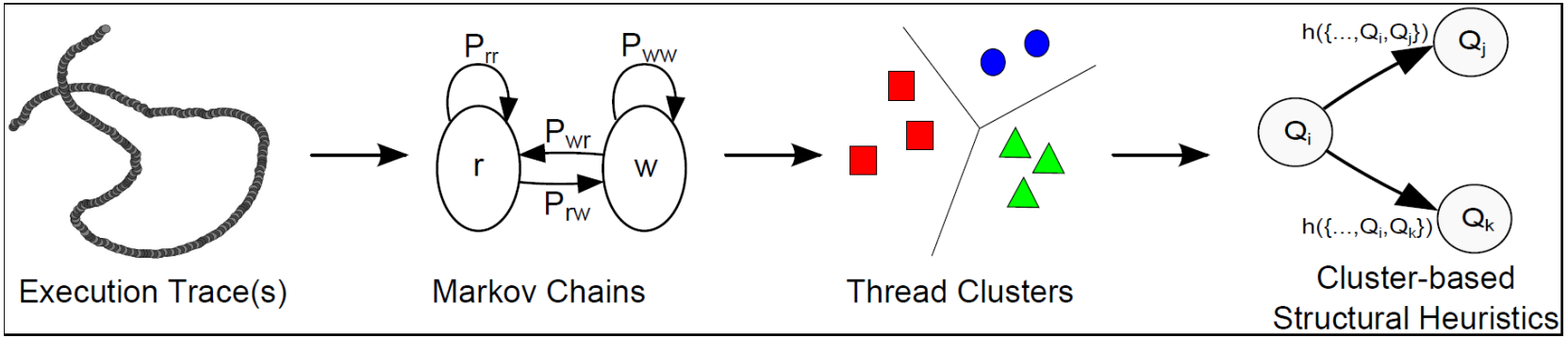
Specializing the h_{ti} Heuristic



Specializing the h_{ti} Heuristic

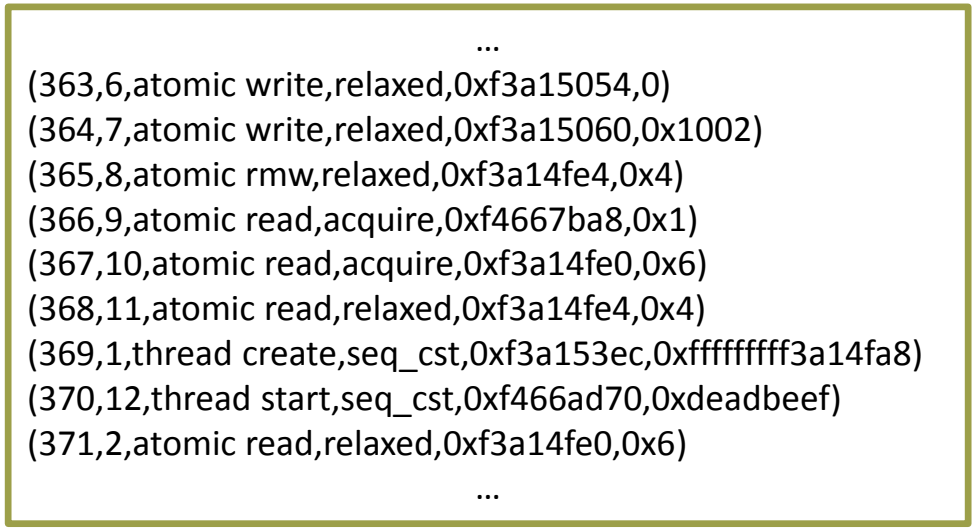


How it works: Overview



THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

How it works: Data Mining on Traces



Execution Trace

(load,store,store,load,...)

(load,load,load,load,...)

(store,store,store,load,...)

Load/Store Sequences (per thread)

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

How it works: Markov Chain Representation

$$p_{rr} = \frac{N_{read,read}}{N_{read,write} + N_{read,read}}$$

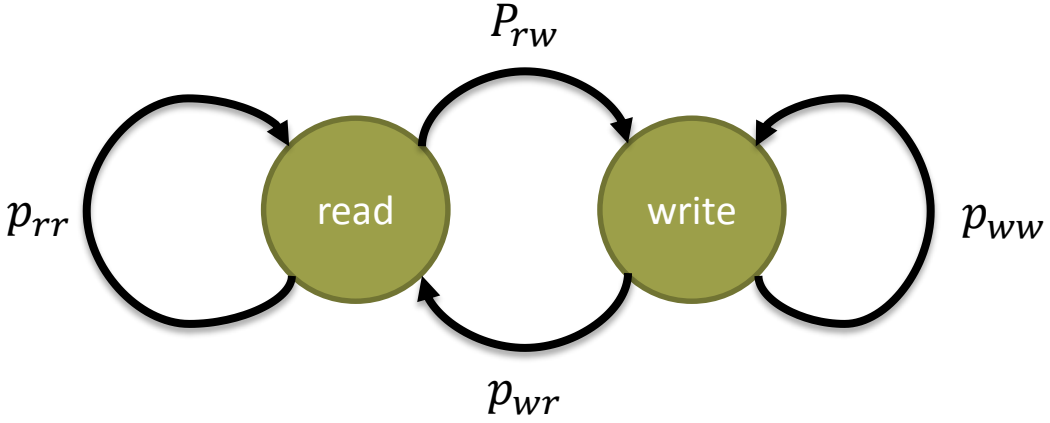
$$p_{rw} = \frac{N_{read,write}}{N_{read,write} + N_{read,read}}$$

$$p_{rw} = \frac{N_{write,read}}{N_{write,read} + N_{write,write}}$$

$$p_{ww} = \frac{N_{write,write}}{N_{write,read} + N_{write,write}}$$

$$S = \begin{bmatrix} p_{rr} & p_{rw} \\ p_{wr} & p_{ww} \end{bmatrix}$$

$$Q = \left\langle q_{read} = \frac{N_{read}}{N_{read} + N_{write}} \middle| q_{write} = \frac{N_{write}}{N_{read} + N_{write}} \right\rangle$$

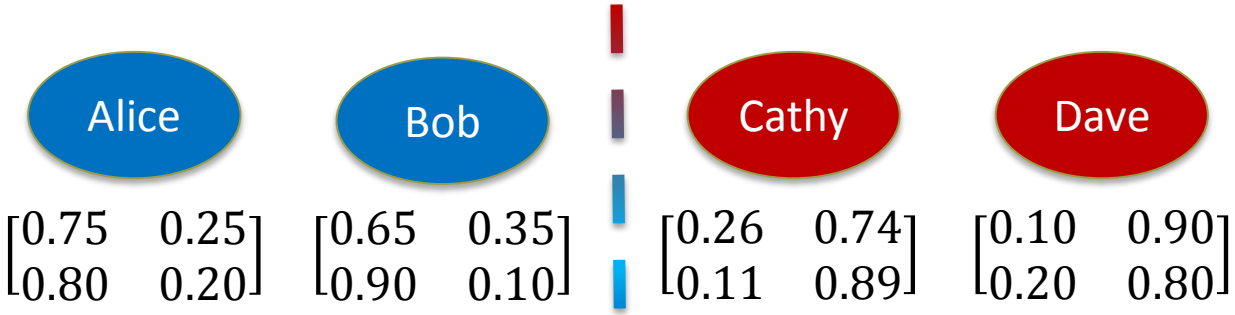


Markov Chain Representation

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

How it works: K-means Clustering

- Constructing a behavioral model for threads by clustering on read/write patterns (using Lloyd's algorithm).



How it works: Inventing novel heuristics

$$h_{ci}\,(path, limit) = \sum_{i=path.length-limit}^{path.length} \begin{cases} path.length * N_{aliveClusters} & \text{if } path[i].cluster == path[i-1].cluster \\ 0 & \text{otherwise} \end{cases}$$

$$h_{ti}\,(path, limit) = \sum_{i=path.length-limit}^{path.length} \begin{cases} path.length * N_{aliveThreads} & \text{if } path[i].tid == path[i-1].tid \\ 0 & \text{otherwise} \end{cases}$$

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

How it works: Inventing novel heuristics

$thrds = \{\text{Alice}, \text{Bob}, \text{Cathy}, \text{Dave}\}$

$L = 3$

$h_{ti}([\text{Alice}, \text{Bob}, \mathbf{Dave}, \mathbf{Alice}, \boldsymbol{Alice}], 3) = 0 + 5 * 4 = 20$

$h_{ti}([\text{Alice}, \text{Bob}, \mathbf{Alice}, \mathbf{Alice}, \boldsymbol{Bob}], 3) = 0 + 0 = 0$

$h_{ti}([\text{Alice}, \text{Bob}, \mathbf{Bob}, \mathbf{Alice}, \boldsymbol{Cathy}], 3) = 0 + 0 = 0$

$h_{ci}([\text{Alice}, \text{Bob}, \mathbf{Dave}, \mathbf{Alice}, \boldsymbol{Alice}], 3) = 0 + 5 * 2 = 10$

$h_{ci}([\text{Alice}, \text{Bob}, \mathbf{Alice}, \mathbf{Alice}, \boldsymbol{Bob}], 3) = 5 * 2 + 5 * 2 = 20$

$h_{ci}([\text{Alice}, \text{Bob}, \mathbf{Bob}, \mathbf{Alice}, \boldsymbol{Cathy}], 3) = 0 + 0 = 0$

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

How it works: Inventing novel heuristics

$$h_{tici}\,(path, limit) = h_{ti}(path, limit) + K_{ci} * h_{ci}(path, limit)$$

$K_{ci} < 0$: Favor intra-cluster interleaving.

$K_{ci} = 0$: Identical to TI alone.

$K_{ci} > 0$: Favor extra-cluster interleaving.

Results

- Cluster assignments on MC representations achieved an 82% correspondence on average with thread categories picked by programmers (perfect correspondence 32% of the time, 90% correspondence nearly half the time).

Name	Cluster #	Job	Name	Cluster #	Job
Alice	0	Accounting	Dave	1	Marketing
Bob	0	Accounting	Earl	0	Marketing
Cathy	1	Marketing	Frank	1	Executive

Results

- In **95%** of performance tests, picking K_{ci} to be -1 or 1 caused h_{tici} to outperform h_{ti}.
- h_{ti} succeeded in finding the bug 41% of the time within resource bounds. Given the right value of K_{ci}, h_{tici} found the bug **100%** of the time.

Results

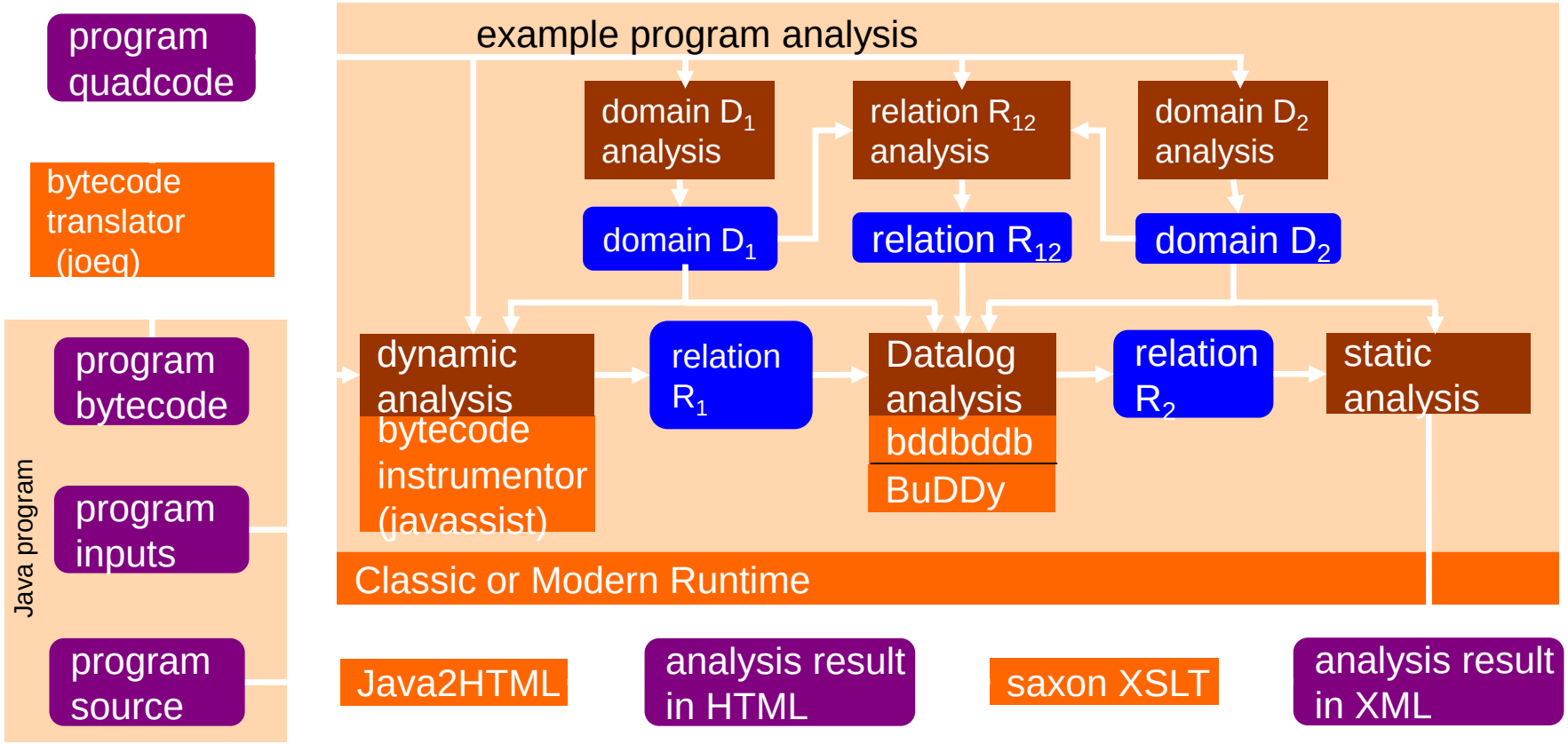
- On average, h_{ti} found a bug after exploring 14001 states. When picking the right K_{ci}, h_{tici} found a bug after exploring **1758 states**. Picking the wrong value caused h_{tici} to find the bug after exploring **27001 states.**
- Cluster analysis, on average, took **96 milliseconds**. Every millisecond spent training saved an average of **5337 milliseconds** on search (for best K_{ci}), and cost an average of **624 milliseconds** on search (for worst K_{ci}).

Ariadne: Hybridizing Static Analysis and Directed Model Checking

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

Chord: Static Analysis for Java

- Chord is an open-source static analysis tool for Java
- Static analysis: Studying the behavior of a program without executing it.
- Provides various off-the-shelf analyses (e.g., various may-alias and call-graph analyses; thread-escape analysis; static and dynamic concurrency analyses for finding races, deadlocks, atomicity violations; etc.), and supports extension.



THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

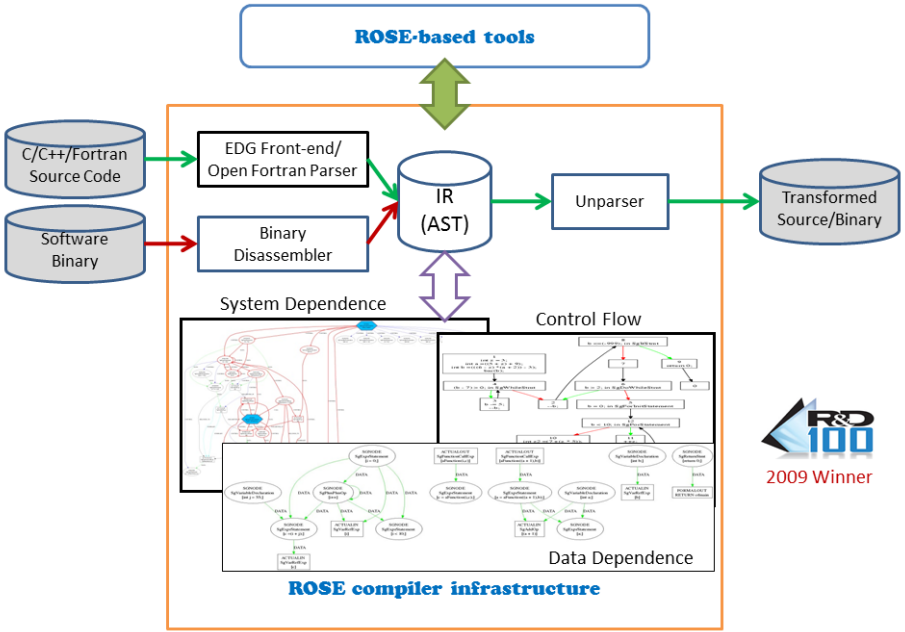
What we want

- Chord produces a bug report indicating where it believes bugs may be in your program.
- We want to translate that knowledge into a form that a state space heuristic can use.

How we do it

- Parse the XML documents provided by Chord.
- Use the ROSE compiler framework to inject instrumentation into the source code that gives hints to the search when threads reach that instrumentation.
- Develop heuristics that exploit that information.

How it works: ROSE



How it works

```
static int x; //global variable

public void f() { g(); }
private void g() { x = x + 1; }
public void h() { i(); }
private void i() { int y = x; }
```

Race condition RC0 (as reported by Chord):

- Thread #1 calls f(), which calls g().
- Thread #2 calls h(), which calls i().
- Thread #1 unsynchronized write, Thread #2 unsynchronized read.

Race condition RC1 (as reported by Chord):

- Thread #1 calls f(), which calls g().
- Thread #2 calls f(), which calls g().
- Thread #1 unsynchronized write, Thread #2 unsynchronized write.

How it works: Developing an annotation scheme

RC0_L: reach code in g()
RC0_R: reach code in i()
RC1_L: reach code in g()
RC1_R: reach code in g()

Seeking another thread on path…

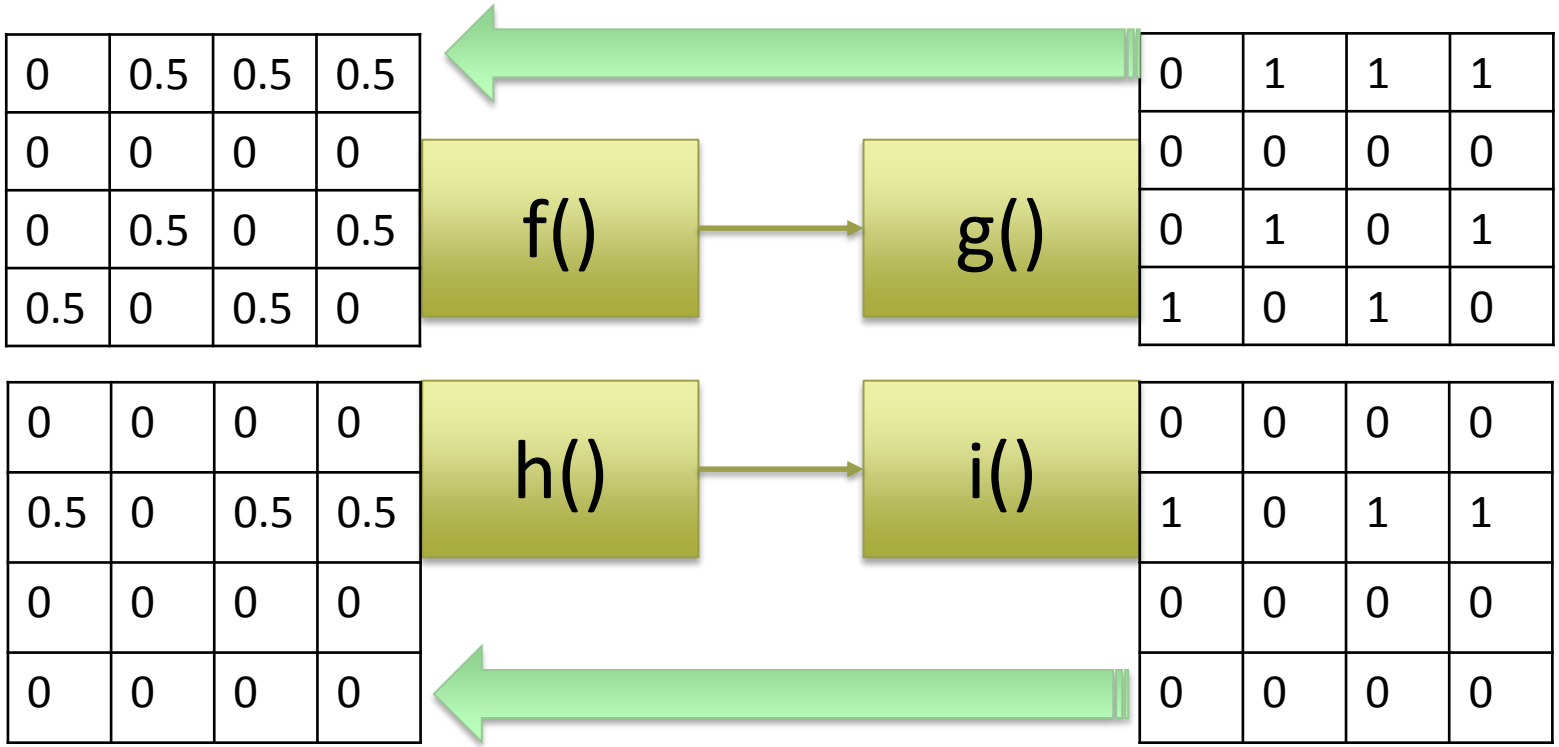
Thread is On path…

	0L	0R	1L	1R
0L	0	0	0	0
0R	1	0	1	1
1L	0	0	0	0
1R	0	0	0	0

Translation: The thread is currently executing code in i(), and is seeking threads executing code in g() to cause the suspected race condition

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

How it works: Developing an annotation scheme



THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

How it works: Developing an annotation scheme

```
static int x; //global variable

public void f() {
        g();
}
private void g() {
        x = x + 1;
}
public void h() {
        i();
}
private void i() {
        int y = x;
}
```

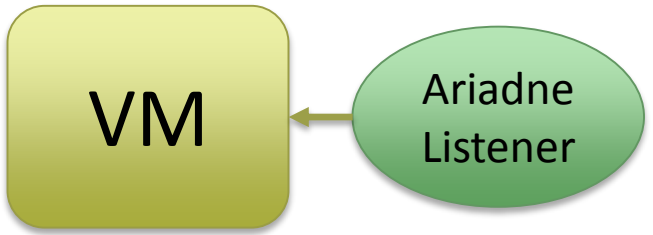
```
static int x; //global variable

public void f() {
        Ariadne.annotation({0,0.5,0.5…})
        g();
}
private void g() {
        Ariadne.annotation({0,1,1…})
        x = x + 1;
}
public void h() {
        Ariadne.annotation({0,0,0…})
        i();
}
private void i() {
        Ariadne.annotation({0,0,0…})
        int y = x;
}
```

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

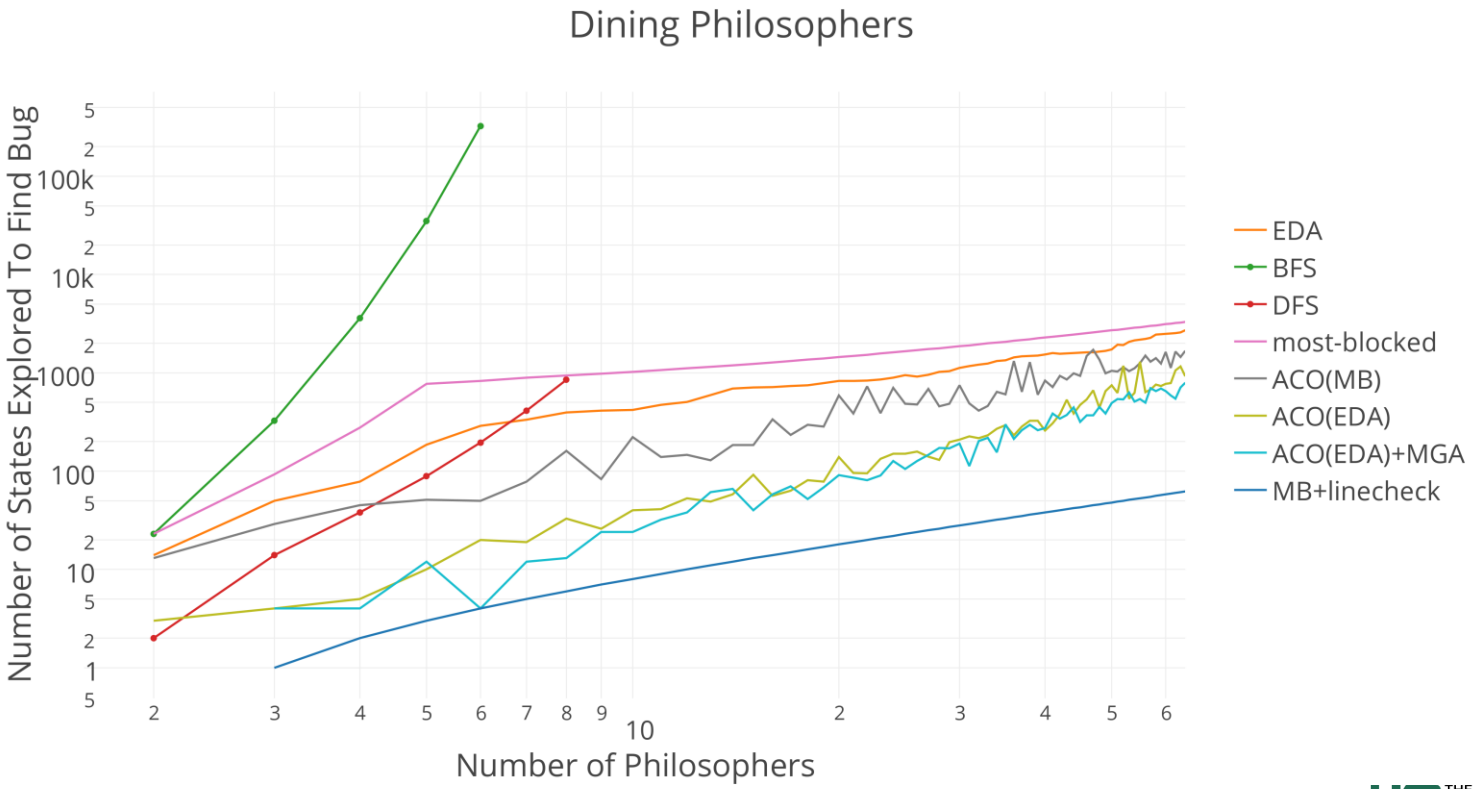
How it works: Listening for Calls with Java Pathfinder

```
public class Ariadne {
        public static void annotation(double[] values){
                //This does nothing.
        }
}
```



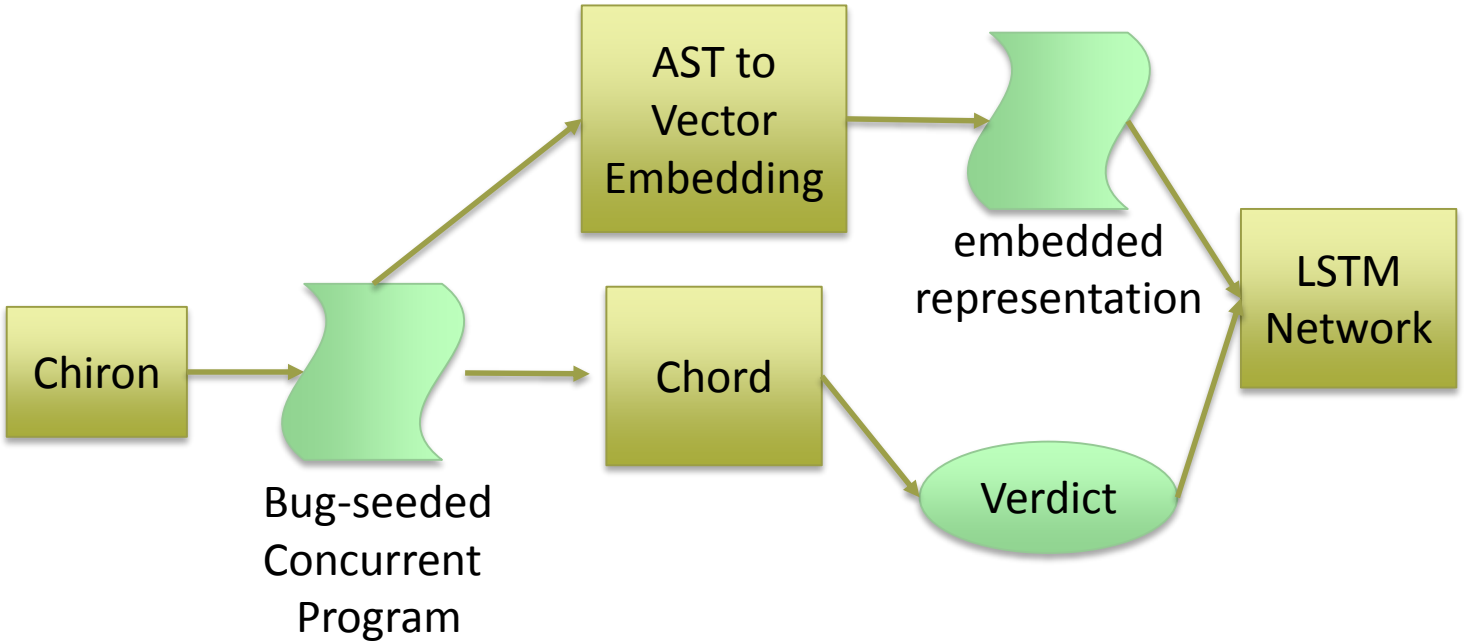
The listener intercepts the execution of the annotation method, and records the ID of the thread that executed it and the values that were passed.

How it works: Potential for Results

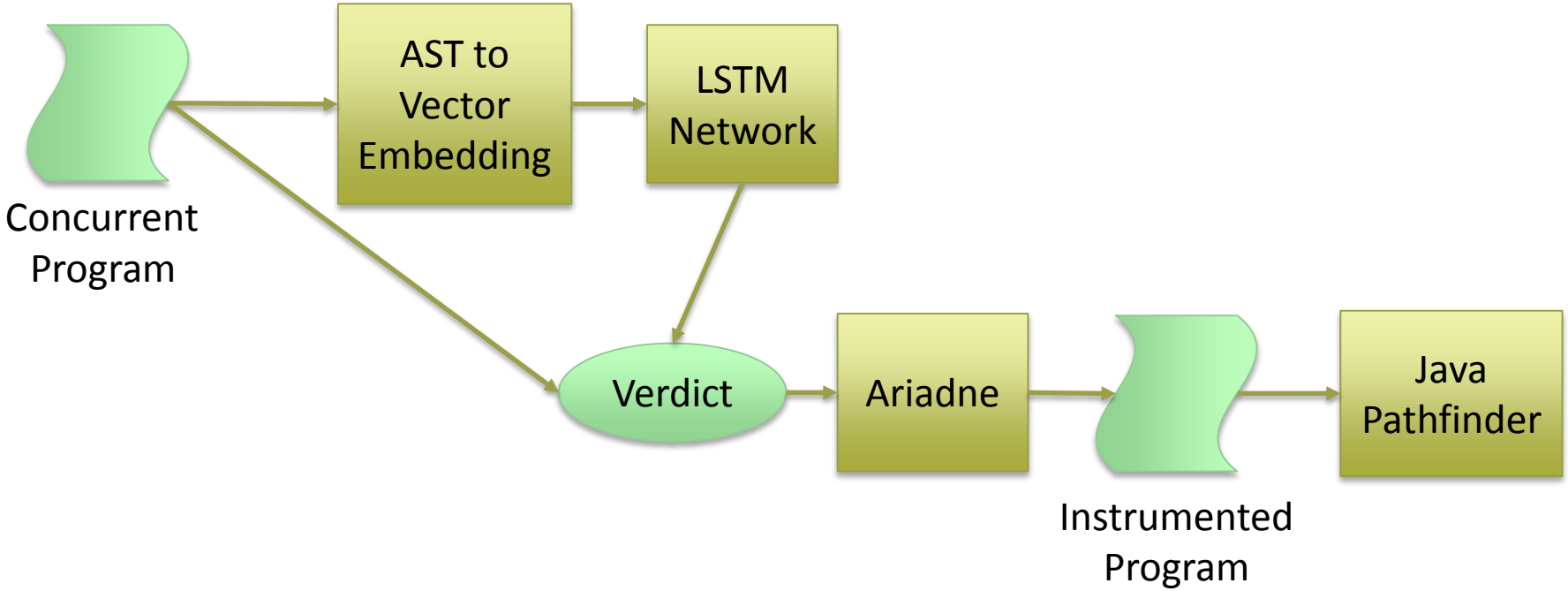


THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

Work in Progress



Work in Progress



THE UNIVERSITY OF ALABAMA AT BIRMINGHAM
Knowledge that will change your world

Questions?

- David H Wolpert and William G Macready. No free lunch theorems for search. Technical report, Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.
- Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pages 329-339, New York, NY, USA, 2008. ACM.
- Alex Groce and Willem Visser. Model checking java programs using structural heuristics. In ACM SIGSOFT Software Engineering Notes, volume 27, pages 12-21. ACM, 2002.
- Alex Groce and Willem Visser. Heuristics for model checking java programs. International Journal on Software Tools for Technology Transfer, 6(4):260-276, 2004.