Practical Model Checking

Dr. John Penix
Automated Software Engineering Group
NASA Ames Research Center

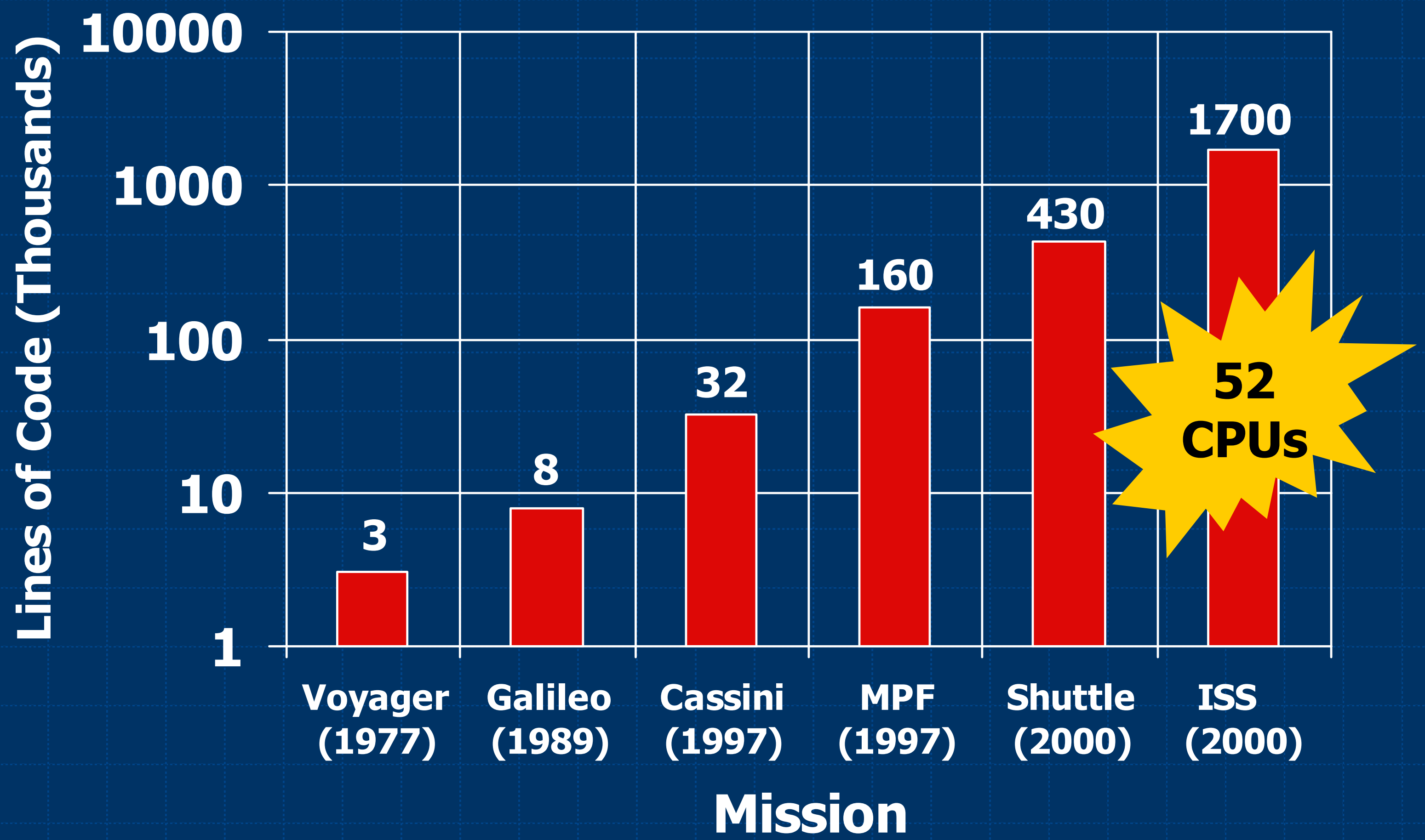
Outline

- NASA's Software Challenges
- Past Work: Program Model Checking
- Transition Challenges
  - Limited coverage of real systems
  - Languages
- Summary
- High-Dependability Computing Program

NASA’s Software Challenges

- High-quality software must be delivered on schedule – astronauts are not beta-testers
- High dependability required over long mission lifecycles while utilizing cutting edge (unproven and risky) software technology
- Increasing system and software complexity pushes beyond the limits of conventional methods for assuring dependability
- Software is developed by interdisciplinary teams from distributed organizations

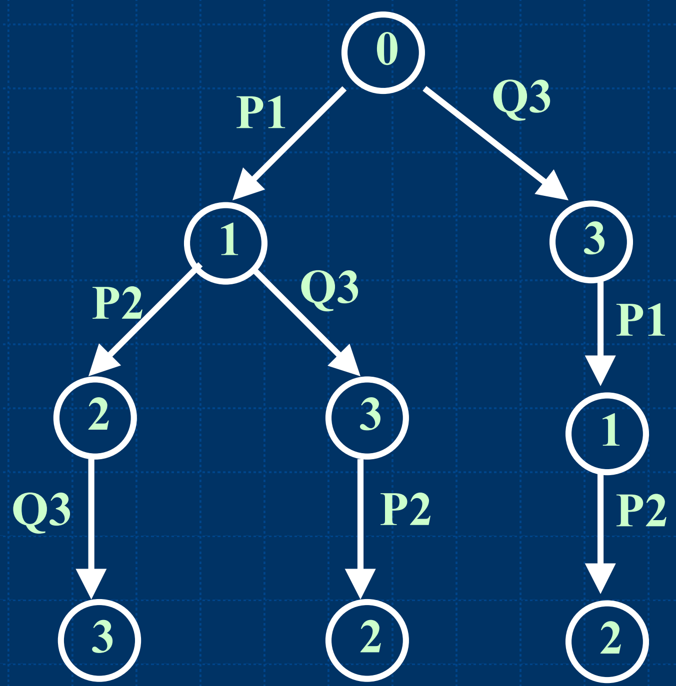
Growing Software Complexity



Testing Concurrent Programs

```
Program
  Var x:int;
  Parallel
     Block P
        x:= 1;
        x:= 2
     End P
  And
     Block Q
        x:=3
     End Q
End
```

Program Executions



Combinations to Test

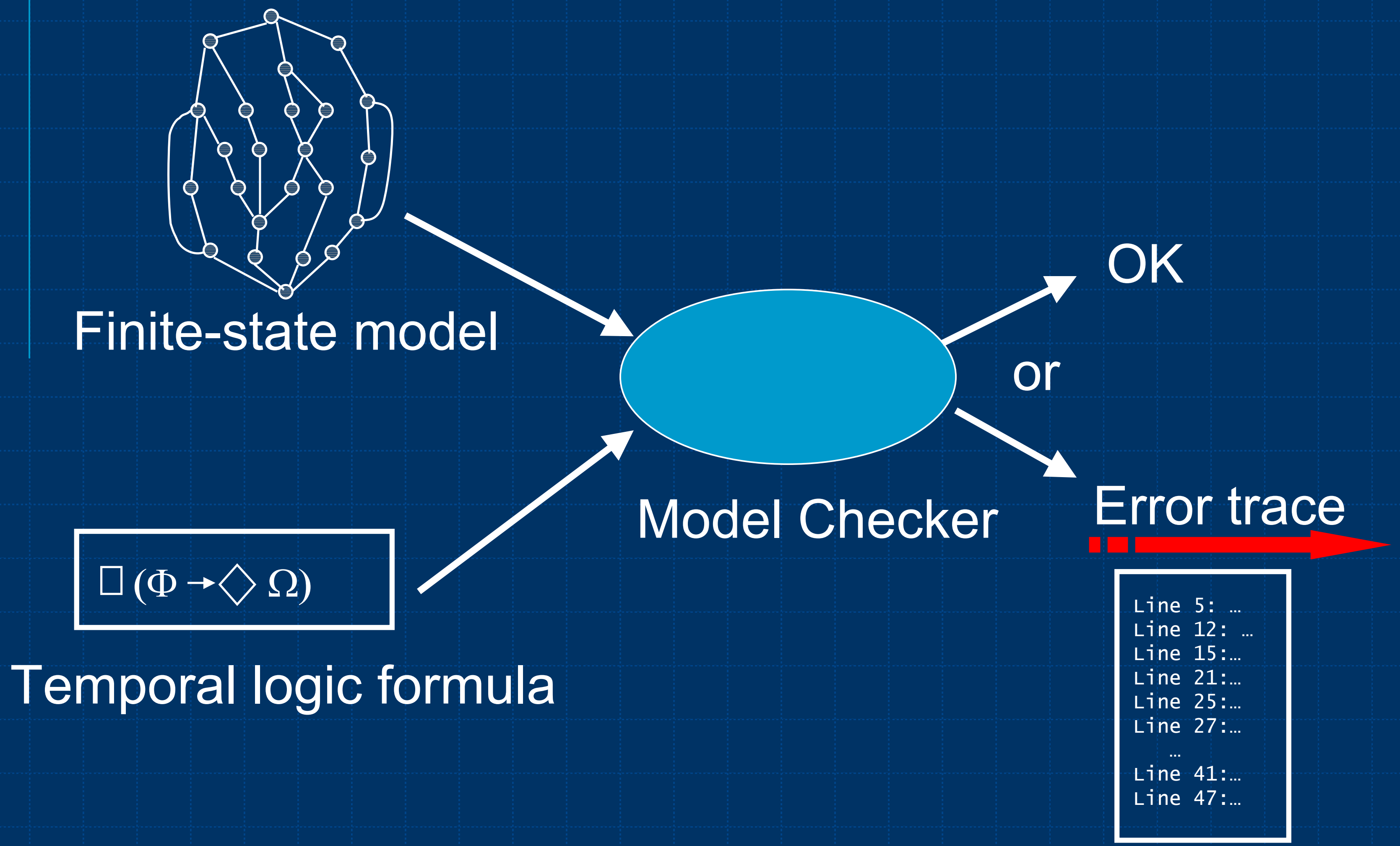
$$\frac{[(\mathrm{size}(P) + \mathrm{size}(Q)]\,!}{\mathrm{size}(P)\,!\ \mathrm{size}(Q)\,!}$$

10,10 : 10^5

100,100 : 10^{59}

1000,1000 : 10^{600}

Model Checking

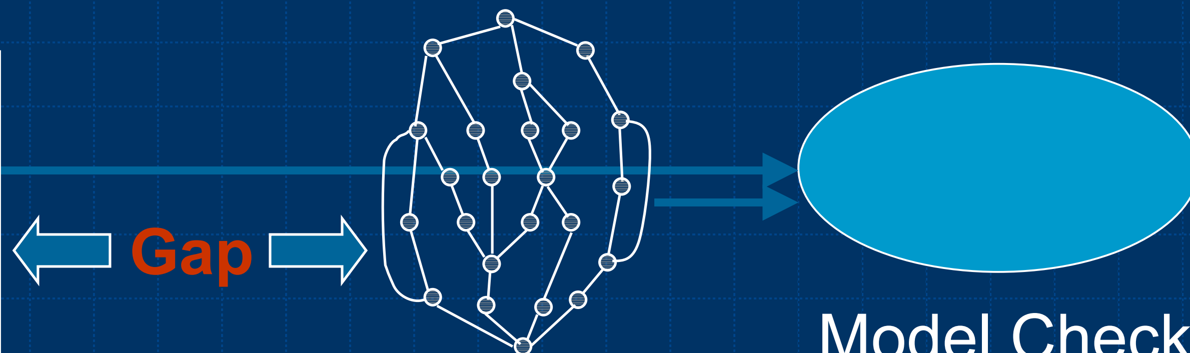


Model Construction Problem

```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

Object take() {
 …
 tail=(tail+1)%size;
 return buffer[tail];
}
```

Program



Model Checker

Model Description

- **Semantic gap:**

  Programming Languages

  methods, inheritance, dynamic creation, exceptions, etc.

  Model Description Languages

  automata

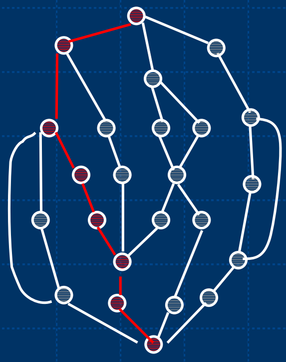
Java PathFinder

```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

Object take() {
 …
 tail=(tail+1)%size;
 return buffer[tail];
}
```

```
0:    iconst_0
1:    istore_2
2:    goto    #39
5:    getstatic
8:    aload_0
9:    iload_2
10:   aaload
```

JVM

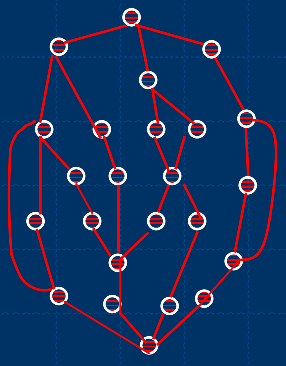


Controllable Scheduler

Special JVM ⟷ Model Checker

Memory not Speed

Modular Design



JPF Highlights

- Models can be infinite state
  - Depth-first state graph generation (Explicit-state model checking)
  - Errors are real
  - Verification can be problematic (Abstraction required)
- All of Java is handled except native code
- Nondeterministic Environments
  - JPF traps special nondeterministic methods
- Properties
  - User-defined assertions and deadlock
  - LTL properties (integrated with Bandera)
- Source level error analysis (with Bandera tool)

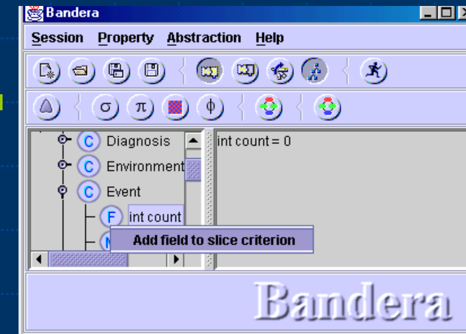
Enabling Technologies

- Remove *irrelevant* code
- Reduce sizes:
  e.g. Queues, arrays etc.
- Reduce variable ranges
  to singleton
- Group statements
  together in atomic blocks
  to reduce interleaving

- Property Preserving Slicing
- Abstraction
  - Under-approximations
  - Over-approximations

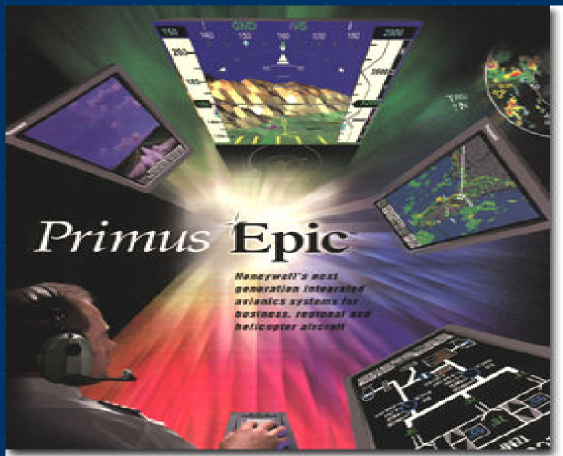
- Partial-order Reductions
- State Compression
- Heuristic Search

Bandera code-level debugging of error-path



Repair

Combined techniques allows $O(10^2)$ source line and $O(10^6)$ state-space increase



DEOS
10000 lines to 1500

3x Slicing 30x

Property preserving

```
Case 0:
new();
Case 1:
Stop();
Case 2:
Remove();
Case 3:
Wait();
```

DEOS
Infinite state to 1,000,000 states

5x Abstraction 100x

Environment Generation

Semi-automated: requires domain knowledge

```
Case 0:
new();
Case 2:
Remove();
```

State compression
2x 15x

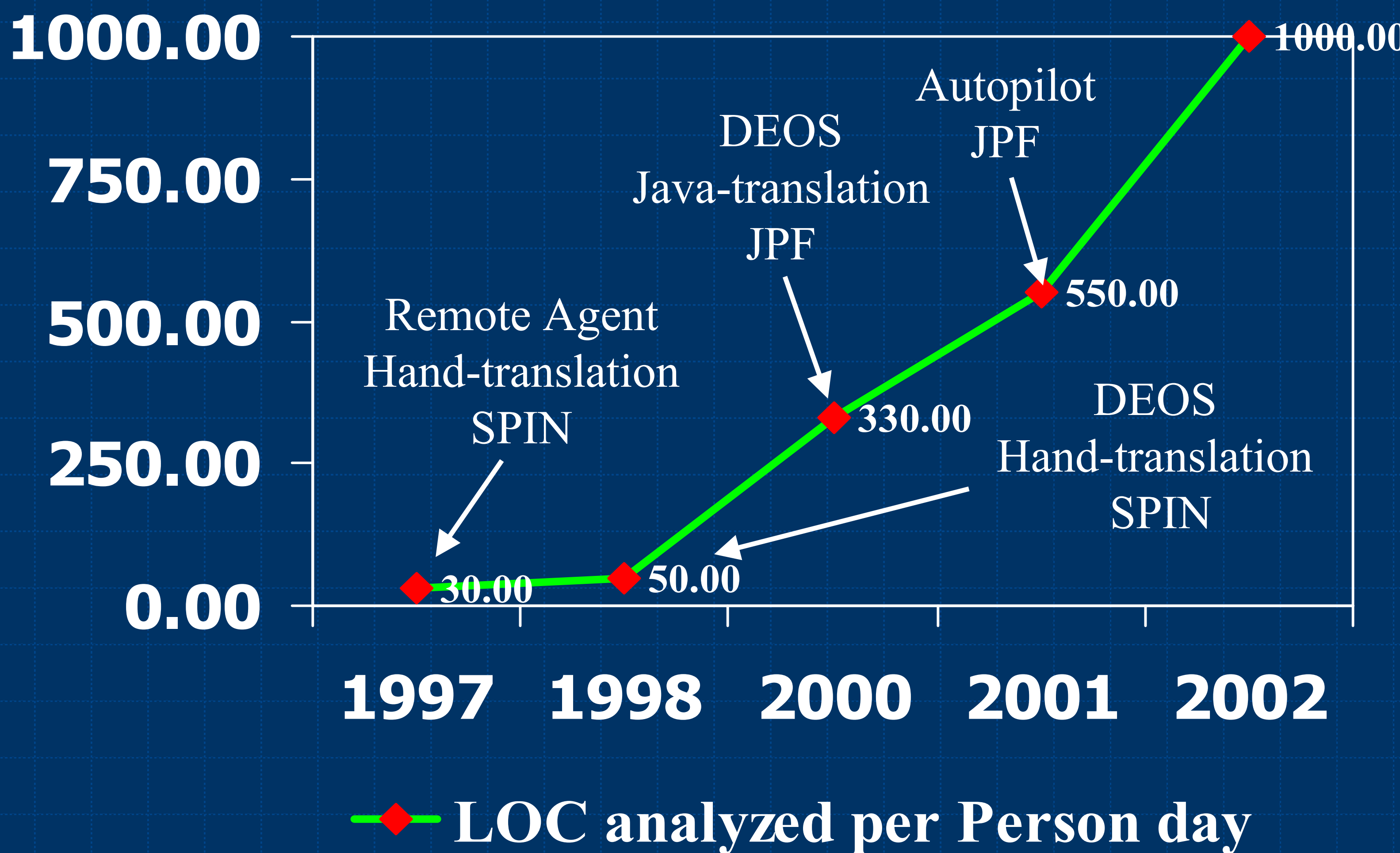
Partial-order reduction
2x 10x

JPF

Spurious error elimination during abstraction

2x 10x
Heuristic search
Focused search for errors

Scaling Program Model Checking



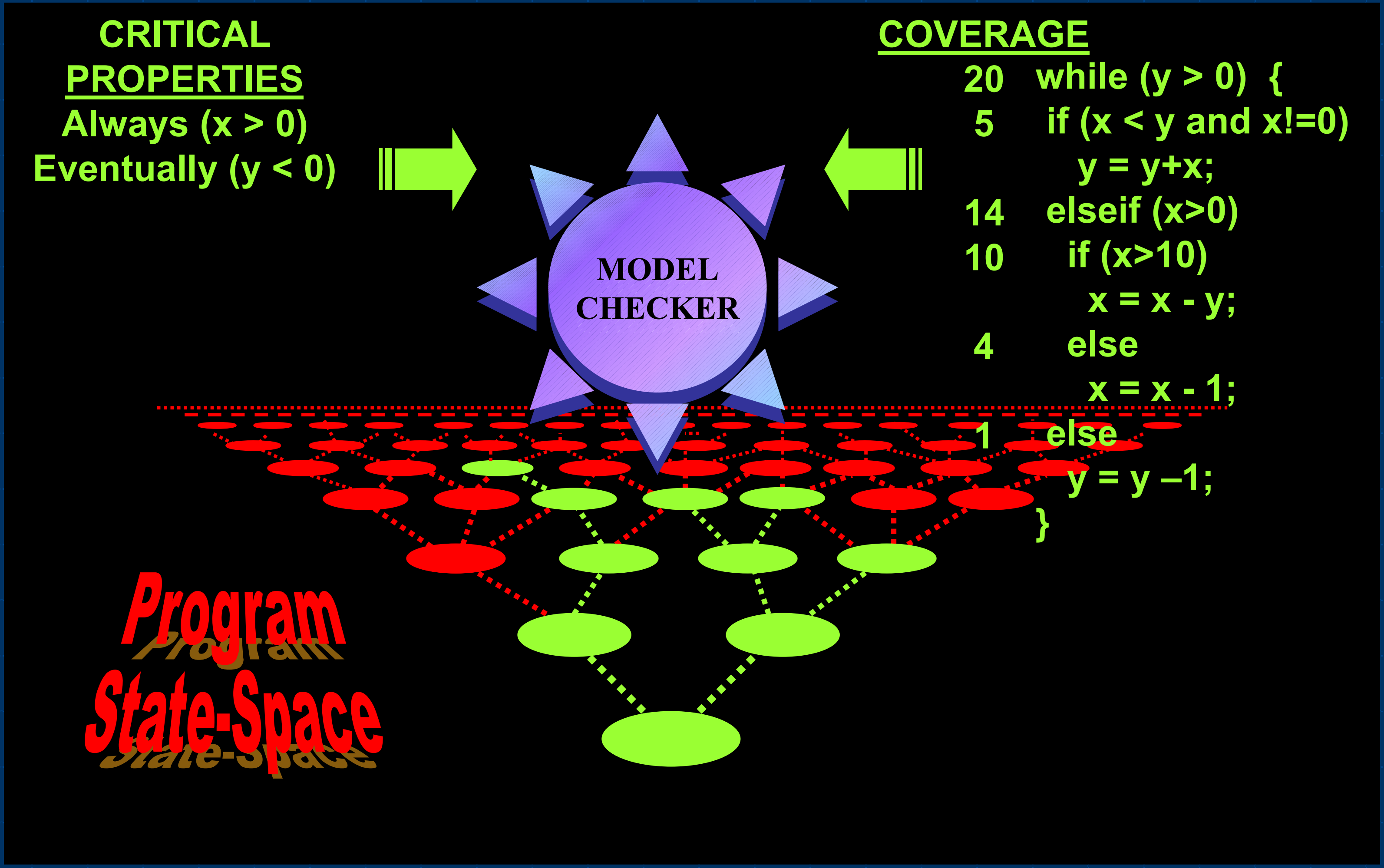
Challenges to Adoption

- State space limitation
  - Verification context limited due to memory
  - But, our success stories were about error detection, not full verification
- Java only used in limited contexts:
  - Java for data monitoring and visualization
  - Embedded Java not picked up at NASA
  - C/C++ used for control applications

Out of Memory… Exception?

- On most real programs, the model checker is going to run out of memory.
  - Program slicing & abstraction are helpful, but effort is required to make them sufficient
- Then, what claim can be made when a model checker only gets partial coverage?
- Furthermore: Can coverage metrics be used to guide the model checker to find errors?

Guided State-Space Analysis



Correctness by Coverage

- If a certain structure (branch, condition, DUpath) has **not** been covered
- Then there is no evidence for claiming that part of the program behavior is working correctly – or is free of errors

poor coverage => weak claim for error-freeness

Correctness by Coverage

Coverage => stronger claim for error-freeness

Coverage => **strong** claim for error-freeness**?**

For the claim to be strong:

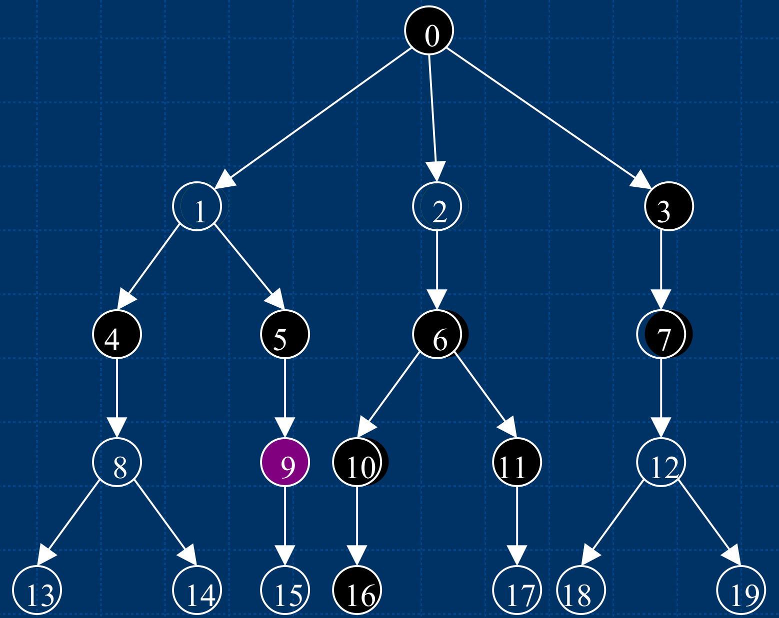
- Metric has a strong correlation to class of errors: coverage-based testing will find **all** errors in the class
- Any set of test cases which provides coverage is equally likely to find an error

Which Metrics for Model Checking?

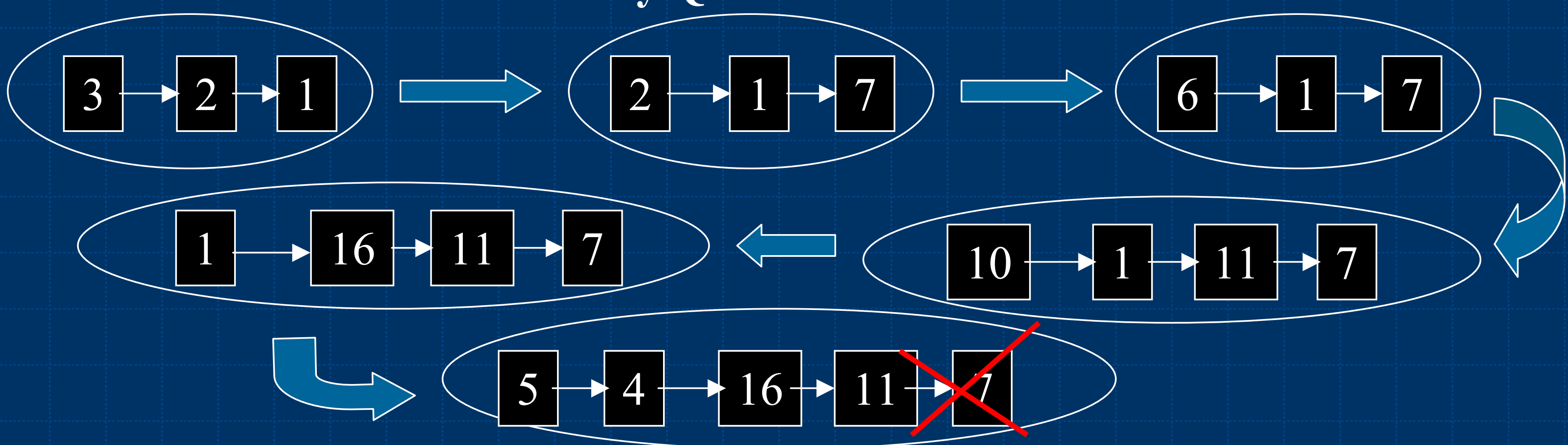
- Decision (Branch), Condition, Condition/Decision, MC/DC?
- Definition/Use and Concurrency Graph coverage?
- Relevant paths coverage?
- Coverage for valid properties
  - what ***is*** the model checker doing???

Directed Search

- Breadth-first (BFS) like state-generation
- Priority queue according to fitness function
- Queue limit parameter



Priority Queue with limit 4



Search Tactics

- Best-First, Beam and A* Search
- Heuristics on structure of Program
  - Branch Exploration: Maximize the coverage of new branches
  - Choose-free heuristic: Minimize non-deterministic choice
  - Assertions: Minimize distance to assertion
- Heuristics based on error classes
  - deadlock: Maximize number of blocked threads
  - Race conditions: Maximize thread interleavings
- User-defined heuristics
  - Full access to JVM's state via API
- Combine heuristics

C++ PathFinder

- Building a C++ to bytecode compiler based on Apogee C++ compilers
- Extensions to JPF JVM
- Challenges:
  - Pointers and memory model
  - Type systems

Pointers and memory

- Assuming that pointer arithmetic is array indexing:
  - malloc → new Array
  - pointer → (ref Array, index) (in complier)

Type systems

- Parameterized types (templates) are handled by the compiler front-end
- Extending the JPF JVM to support multiple inheritance – compiler passes superclass info via class file attributes

C++ Front-end Status

- End to end C++ → bytecode working on small examples
- JPF JVM extensions underway
- Integration in April

Conclusion

- Many barriers to having practical tools
  - Languages, Performance, Coverage
- Have to get something out there:
  - Knobs & Dials to trade cost/benefits
  - What properties are important?
  - Property checking vs. error detection?
  - Integration with life-cycle: unit testing? design checking? code reviews?

JPF info

http://ase.arc.nasa.gov/jpf/

High-Dependability Computing Program

Dr. Michael R. Lowry
NASA Ames Research Center

HDCP Purpose

- Develop scientific basis for engineering high-dependability computing systems (software and systems) through an experimental test-bed facility.
- Provide researchers a national facility for experimenting with technology to improve dependability on realistic systems at significant scale.
- Provide NASA and IT industry empirically validated methods to *predict* dependability and to *achieve* dependability. Anticipate that IT industry will increasingly provide components to Aerospace integrators - but these components must be highly dependable.

HDCP Components

- High-Dependability Computing Program:
  - CMU West (NASA Ames Research Park)
  - CMU Pittsburgh
  - USC, MIT, UMD, Washington & Wisconsin
- NASA collaborators
- Openly competed university research NASA + NSF + NSA funding
- Industry Consortium: Adobe, Cisco, Compaq, HP, IBM, ILOG, Marimba, Microsoft, Novell, Oracle, SGI, Siebel Systems, Sybase, Sun Microsystems