What Have We Seen?

- JML Tools: Runtime assertion checking
- ESC/Java: Static checking of JML annotations and runtime constraints
- KeY: Formal proof of JML annotations
- Jahob: Data structure verification

Symbolic state representation and reasoning
Explicit State Model Checking
Now: Explicit State

- Concrete representation of states, e.g., $x = 4, y = 3$
- Transitions produce new concrete states, e.g.,
  $x = 4, y = 3 \xrightarrow{x=x+1} x = 5, y = 3$
- System model: Transition System (TS)
- Graph search algorithms used to search for property violations
Transition Systems (TS)

Definition (Transition System)
A transition system ($TS$) is a structure $TS = (Q, Act, \to)$, where
- $Q$ is a set of states,
- $Act$ a set of actions,
- $\to \subseteq Q \times Act \times Q$ the transition relation.

$Q = \{ q_0, q_1, q_2, q_3 \}$
$I = \{ q_0 \}$
$
\to = \{(q_0, x++, q_1), (q_1, y++, q_3), (q_0, y++, q_2), (q_2, x++, q_3)\}$

$\text{start} \quad \rightarrow \quad q_0 \quad \rightarrow \quad q_1 \quad \rightarrow \quad q_3 \quad \rightarrow \quad q_2$
Exploring Transition Systems

- Treat transition system as graph
- Use graph search algorithm to explore states
- Different search strategies:
  - Depth-First-Search (DFS)
  - Breath-First-Search (BFS)
  - Greedy Search

Goal: Find error fast ("before running out of memory")

More debugging than verification
Searching
Basics

- Explore states in a graph.
- Unify states.
- Keep “pending list” of nodes yet to explore.
- Keep “closed list” of already explored states.

Theory

Explore all possible states.

Practice

Heuristic cutoff:

- bounded number of states
- bounded path length
- ...
Abstract Searching

1. Choose and remove next state $s$.
2. If $s$ is already closed, goto Step 1
3. Evaluate $s$.
4. Add all successors of $s$ onto the pending list
5. Move $s$ to closed list

Main Operations
- State evaluation
- Creation of successor states
- State unification
Different Types

Uninformed Searches
- Exploration order determined by graph structure.
- Not goal-directed.

Informed Searches
- Exploration order guided by heuristics and/or path length.
- “Prefer short paths.”
- Heuristic value = estimate of distance to goal.
- uninformed search
- first explore the successor nodes, then the siblings
- **Pending list**: LIFO (e.g., stack)
Breath-First-Search (BFS)

- uninformed search
- first explore the siblings, then the successor nodes
- Pending list: FIFO (e.g., Queue)
Greedy Search

- informed search
- heuristic estimate of the minimal distance of a state to a goal
- expand state with minimal value of the heuristic
- Pending list: Ordered list (e.g., priority queue or Heap)

Problems

- Highly sensitive to heuristic
- Plateaus
- Found error path might still be long

... but highly efficient in practice
A* Search

- informed search
- use heuristic,
- but also consider the cost of the path to the current state
- expand state with minimal sum of heuristic value and path cost
- Pending list: Ordered list (e.g., priority queue or Heap)

Admissible heuristics

Let $n$ be a node and $d(n)$ be the exact distance of node $n$ to the goal. Heuristic $h$ is admissible if and only if

$$\forall v. \ h(v) \leq d(v)$$

A* search with admissible heuristic ensures shortest path to goal!
Observation

Search procedures only differ in the order in which they explore the state space.

We can express all these search methods using two functions over states \( s \) (and a bound on the length of paths):

- \( d(s) \) - a distance function
- \( h(s) \) - a heuristic function

Choose \( s \) that minimizes \( d(s) + h(s) \).

<table>
<thead>
<tr>
<th></th>
<th>( d(s) )</th>
<th>( h(s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>(-\text{pathlength}(s))</td>
<td>0</td>
</tr>
<tr>
<td>BFS</td>
<td>\text{pathlength}(s)</td>
<td>0</td>
</tr>
<tr>
<td>Greedy Search</td>
<td>0</td>
<td>\text{heuristic}(s)</td>
</tr>
<tr>
<td>A*</td>
<td>\text{pathlength}(s)</td>
<td>\text{heuristic}(s)</td>
</tr>
</tbody>
</table>
Java Virtual Machine
Virtual vs. Concrete Machine

- Programs
  - Virtual Machine
    - Machine independent code
    - Machine dependent interpreter in machine code
    - Machine code interpreter
  - Concrete Machine
    - Rebuild for every concrete machine
    - Compile once — Run everywhere
JVM Basics

- JVM interprets .class files
- .class files contain
  - a description of classes (name, fields, methods, inheritance relationships, referenced classes, ...)
  - a description of fields (name, type, attributes (visibility, volatile, transient, ...))
  - bytecode for the methods

- Stack machine
- Typed instructions
- Bytecode verifier to ensure type safety
Different Memory Areas

Java separates between

- a **Java stack**
  - Used for method calls and expression evaluation
  - One per thread
  - Checked for overflows

- a **native stack**
  - Used for native calls using JNI
  - Not directly usable by the bytecode
  - Not checked for overflows

- a **heap**
  - Used for dynamic allocation
  - Managed by garbage collectors
  - Shared between all threads
  - Size limited by JVM configuration
**Calling Methods**

**Activation Frame** contains:

- Variables local to the called method
- Stack space for instruction execution (**Operand Stack**)

One activation frame per method call: $x.foo()$

1. pushes new activation frame
2. calls the method $foo$
3. pops the activation frame
Executing Instructions

- Arguments are on the operand stack
  - Some instructions move local variables or constants to the stack
- Most instructions pop topmost arguments from the stack and push result onto the stack
Example: lcmp

Compare two long values l1 and l2.

long l2 = popLong();
long l1 = popLong();
if (l1 < l2)
    push(-1);
if (l1 == l2)
    push(0);
if (l1 > l2)
    push(1);
Java Native Interface (JNI)

- foreign function interface
- execution jumps to non-Java code
- runs outside of VM
- uses native stack
- but can access JVM through JNIEnv structure
  - JNIEnv needed to translate between native stack and heap
- useful to access native OS libraries or optimize certain computation tasks
  - Assumption: Native code is faster than Java code
  - Note: Native code breaks platform independence