Formal Methods for Java

Lecture 7: Explicit State Model Checking and JVM

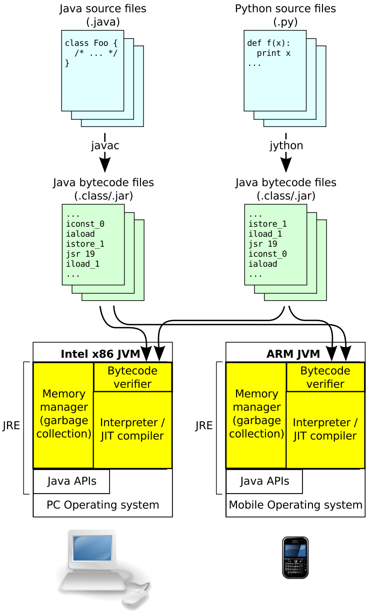
Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Nov 13, 2012

Java and the Virtual Machine

- Programs are written in Java or some other language
- Compiler translates this to Java Bytecode.
- Platform-specific Java Virtual Machine executes the code.



Java Virtual Machine (JVM)

- 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
- Integer stack
- Typed instructions
- Bytecode verifier to ensure type safety

Executing Instructions

- Arguments are on the operand stack
- Most instructions pop the topmost arguments from the stack and push result onto the stack
- Some instructions read/write local variables or object fields.

Instruction Group "Local Variable Instructions"

- `aload, iload, lload, fload, dload`
  Stores local variable on the operand stack
- `astore, istore, lstore, fstore, dstore`
  Stores top of operand stack into a local variable
- `iinc`
  Increments a local variable (does not touch the operand stack).

Example

Let x, y be the first and second integer variables.
Then x=y is compiled to the bytecode

```
iload_2
istore_1
```

Instruction Group "Constant value Instructions"

- `iconst, lconst, fconst, dconst, aconst_null`
  Pushes a fixed constant value on the operand stack
- `bipush, sipush`
  Pushes a byte or short constant value (given as parameter of the instruction) on the operand stack
- `ldc, ldc_w, ldc2_w`
  Pushes a constant value from the constant pool (part of the class file) on the operand stack.

Example

Let x, y, z be integer variables.

Then x=5, y=10000, z=1000000 is compiled to the bytecode

```
iconst_5
istore_1
sipush 10000
istore_2
ldc    #2;  //int 1000000
istore_3
```

Instruction Group "Stack Manipulation"

- `pop` and `pop2`
  Remove the topmost (2) elements from the operand stack
- `dup`, `dup2`, `dup_x1` ...
  Duplicate the top element(s) of the stack
- `swap`
  Exchange the topmost two elements on the operand stack

Example

The code

```
return a[i] += 1;
```

is translated as

```
aload_1   // load a
iload_2   // load i
dup2      // duplicate, stack contains a,i,a,i
iaload    // read a[i], stack now contains a,i,a[i]
iconst_1
iadd      // add one
dup_x2    // duplicate, stack contains a[i]+1,a,i,a[i]+1
iastore   // store a[i]+1 into a[i].
ireturn   // return duplicated result.
```

Instruction Group "Field Access Instructions"

- `getfield`
  Takes the object o from the operand stack and
  puts the value of an instance field of o onto the stack.
- `getstatic`
  Puts the value of a static field onto the stack.
- `putfield`
  Takes an object o and a value from the stack and
  writes the value of into the instance field of o.
- `putstatic`
  Takes a value from the stack and
  writes it into a static field.

Instruction Group “Method Invocation”

- `invokespecial`
  Invoke method without polymorphic resolution.
  Object and parameters are taken from the stack.
- `invokestatic`
  Invoke a static method. Parameters are taken from the stack.
- `invokevirtual`
  Invoke method with polymorphic resolution.
  Object and parameters are taken from the stack.
- `invokeinterface`
  Like `invokevirtual` but used for interface methods.

Example

The code

```
return new Integer(this.value);
```

is translated as

```
new java.lang.Integer
dup
aload_0 // load this
getfield MyClass.value
invokespecial java.lang.Integer.<init>(int)
areturn
```

Instruction Group "Monitor Instructions"

- `monitorenter`
  Enter a critical section
- `monitorexit`
  Leave a critical section

Instruction Group “Miscellaneous”

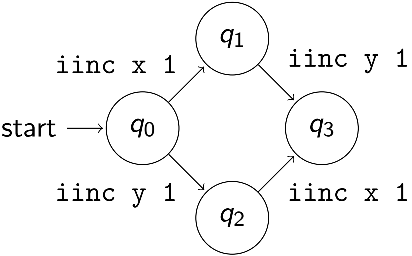
- `checkcast`
  Check a cast and throw a *ClassCastException* if cast fails
- `instanceof`
  Check if reference points to an instance of the specified class
- `athrow`
  Throw an exception or an error
- `nop`
  Do nothing

Transition Systems (TS)

Definition (Transition System)

A transition system (TS) is a structure $TS = (Q, Act, \rightarrow)$, where

- Q is a set of states,
- Act a set of actions,
- $\rightarrow \subseteq Q \times Act \times Q$ the transition relation.



$$
\begin{aligned}
Q &= \{q_0, q_1, q_2, q_3\} \\
I &= \{q_0\} \\
\rightarrow &= \{(q_0, \texttt{iinc x 1}, q_1), \\
&\quad (q_1, \texttt{iinc y 1}, q_3), \\
&\quad (q_0, \texttt{iinc y 1}, q_2), \\
&\quad (q_2, \texttt{iinc x 1}, q_3)\}
\end{aligned}
$$

Operational semantics for the JVM

- State consists of heap and sequence of activation frames.
- An action is the execution of a single bytecode instruction.

Explicit State Model Checking

Model checking

- Idea: exhaustively check the system
- Try all possible paths/all possible input values.
- Use search strategies to find errors fast.

Runtime checking vs. Model checking vs. Verification

Runtime Checking	Model Checking	Verification
JML Tools	JPF	ESC/Java2
		KeY
		Jahob

Now: Explicit State

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

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

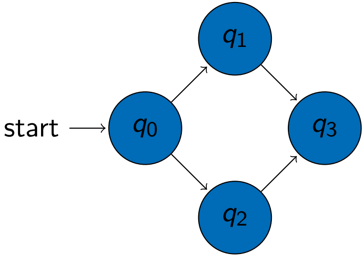
- Exploration order determined by graph structure.
- Not goal-directed.

Informed Searches

- Exploration order guided by heuristics and/or path length.
- "Prefer short paths."
- Heuristic value = estimate of distance to goal.

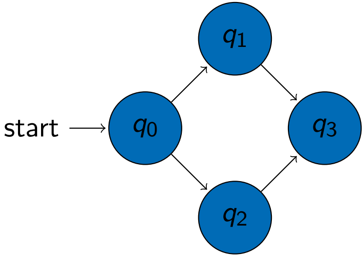
Depth-First-Search (DFS)

- 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!

A Unified Search Framework

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 $d(s) + h(s)$.

	$d(s)$	$h(s)$
DFS	$-pathlength(s)$	0
BFS	$pathlength(s)$	0
Greedy Search	0	$heuristic(s)$
A*	$pathlength(s)$	$heuristic(s)$