Formal Methods for Java

Lecture 19: Explicit State Model Checking and JVM

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Jul 5, 2017

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

➥ Symbolic state representation and reasoning

Explicit State Model Checking

Now: Explicit State

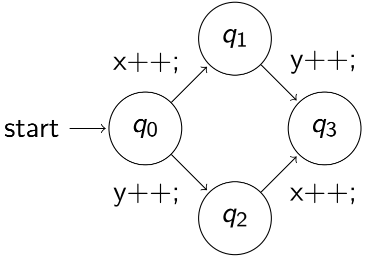
- Concrete representation of states, e.g., $\boxed{x = 4, y = 3}$
- Transitions produce new concrete states, e.g.,
  $\boxed{x = 4, y = 3} \xrightarrow{x=x+1} \boxed{x = 5, y = 3}$
- System model: Transition System (TS)
- Graph search algorithms used to search for property violations

Transition Systems (Reminder)

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.



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

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

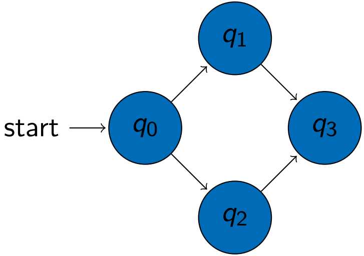
- Exploration order determined by graph structure.
- Not goal-directed.

Informed Searches

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

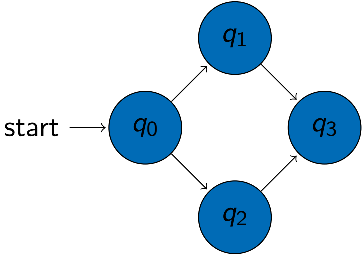
Depth-First-Search (DFS)

- 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):

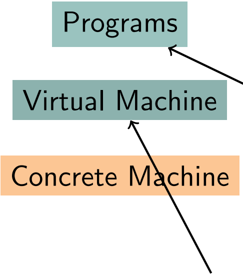
- $d(s)$ - a distance function
- $h(s)$ - a heuristic function

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*	$pathlength(s)$	$heuristic(s)$

Java Virtual Machine

Virtual vs. Concrete Machine



- Machine independent code
- Machine dependent interpreter in machine code
- Machine code interpreter

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

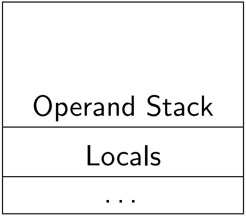
Java separates between

- 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

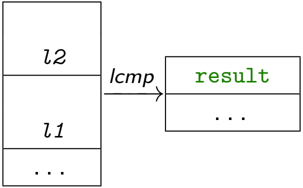
Executing Instructions

- 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 trough *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

JVM Instructions

- Most instructions are typed,
- but internally, only `int`, `long`, and `double` matter.
- Other types only used by the bytecode verifier
- Instructions can be grouped

Instruction Group "Load Instructions"

- `tload` where $t \in \{a, i, l, f, d\}$
  Stores local variable on the operand stack
- `taload` where $t \in \{a, b, s, i, l, f, d\}$
  Stores element of an array on the operand stack
- `aconst_null`
  Stores `null` on the operand stack
- `tconst_<n>` where $t \in \{i, l, d\}$
  Stores constant on the operand stack (only limited values possible)
- `bipush`, `sipush`
  Push `byte` resp. `short` constant on the operand stack
- `ldc`
  Load constant from the constant pool

Instruction Group “Store Instructions”

- `tstore` where $t \in \{a, i, l, f, d\}$
  Store top of operand stack into local variable
- `tastore` where $t \in \{a, b, s, i, l, f, d\}$
  Store top of operand stack into array

Instruction Group “Stack Manipulation”

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

Instruction Group “Conversion Instructions”

- `i2t` where $t \in \{b, c, d, f, l, s\}$
  Convert `int`
- `l2t` where $t \in \{d, f, i\}$
  Convert `long`
- `f2t` where $t \in \{d, i, l\}$
  Convert `float`
- `d2t` where $t \in \{f, i, l\}$
  Convert `double`

Instruction Group "Branching Instructions"

- `if_acomp`
  Compare two references and jump on success
- `if_icomp`
  Compare two `int`s and jump on success
- `if`
  Compare against 0 and jump on success
- `tcmp` where $t \in \{f, d\}$
  Compare two floating point numbers (don't jump)
- `ifnonnull`
  Jump if reference is not `null`
- `ifnull`
  Jump if reference is `null`
- `goto`
  Unconditional jump
- `jsr`
  Jump to subroutine

Instruction Group “Switch Instructions”

- `lookupswitch`
  Switch based upon a search in an ordered offset table
- `tableswitch`
  Switch based on index into an offset table

Instruction Group “Return Instructions”

- `treturn` where $t \in \{a, i, l, f, d\}$
  Return a value from a method
- `return`
  Return from a `void` method
- `ret`
  Return from subroutine

Instruction Group "Arithmetic Instructions"

- `tneg` with $t \in \{i, l, f, d\}$
  Negate a number
- `tadd` with $t \in \{i, l, f, d\}$
  Add two numbers
- `tsub` with $t \in \{i, l, f, d\}$
  Subtract two numbers
- `tmul` with $t \in \{i, l, f, d\}$
  Multiply two numbers
- `tdiv` with $t \in \{i, l, f, d\}$
  Divide two numbers
- `trem` with $t \in \{i, l, f, d\}$
  Compute the remainder of a division ($result = value_1 - (value_2 * q)$)
- `iinc`
  Increment integer by constant

Instruction Group "Logic Instructions"

- `tand` where $t \in \{i, l\}$
  Bitwise and
- `tor` where $t \in \{i, l\}$
  Bitwise or
- `txor` where $t \in \{i, l\}$
  Bitwise xor
- `tshr` where $t \in \{i, l\}$
  Logical shift right with sign extension
- `tushr` where $t \in \{i, l\}$
  Logical shift right with zero extension
- `tshl` where $t \in \{i, l\}$
  Logical shift left

Instruction Group “Object Creation Instructions”

- `new`
  Create a new object on the heap
- `newarray`
  Create a new array containing only elements of a primitive type on the heap
- `anewarray`
  Create a new array containing only elements of a reference type on the heap
- `multianewarray`
  Create a new multi-dimensional array on the heap

Instruction Group “Field Access Instructions”

- `getfield`
  Get the value of an instance field
- `getstatic`
  Get the value of a static field
- `putfield`
  Write the value of an instance field
- `putstatic`
  Write the value of a static field

Instruction Group "Method Invocation"

- `invokeinterface`
  Invoke method with polymorphic resolution
- `invokespecial`
  Invoke method without polymorphic resolution
- `invokestatic`
  Invoke a static method
- `invokevirtual`
  Invoke method with polymorphic resolution.

Instruction Group “Monitor Instructions”

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

Instruction Group “Miscellaneous”

- `arraylength`
  Get the length of an array
- `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
- `wide`
  Enable bigger operands