Introduction to JavaPathfinder
Part 2

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Sagar Chaki
27 November 2007

Outline

Search Strategies

Partial-Order Reduction

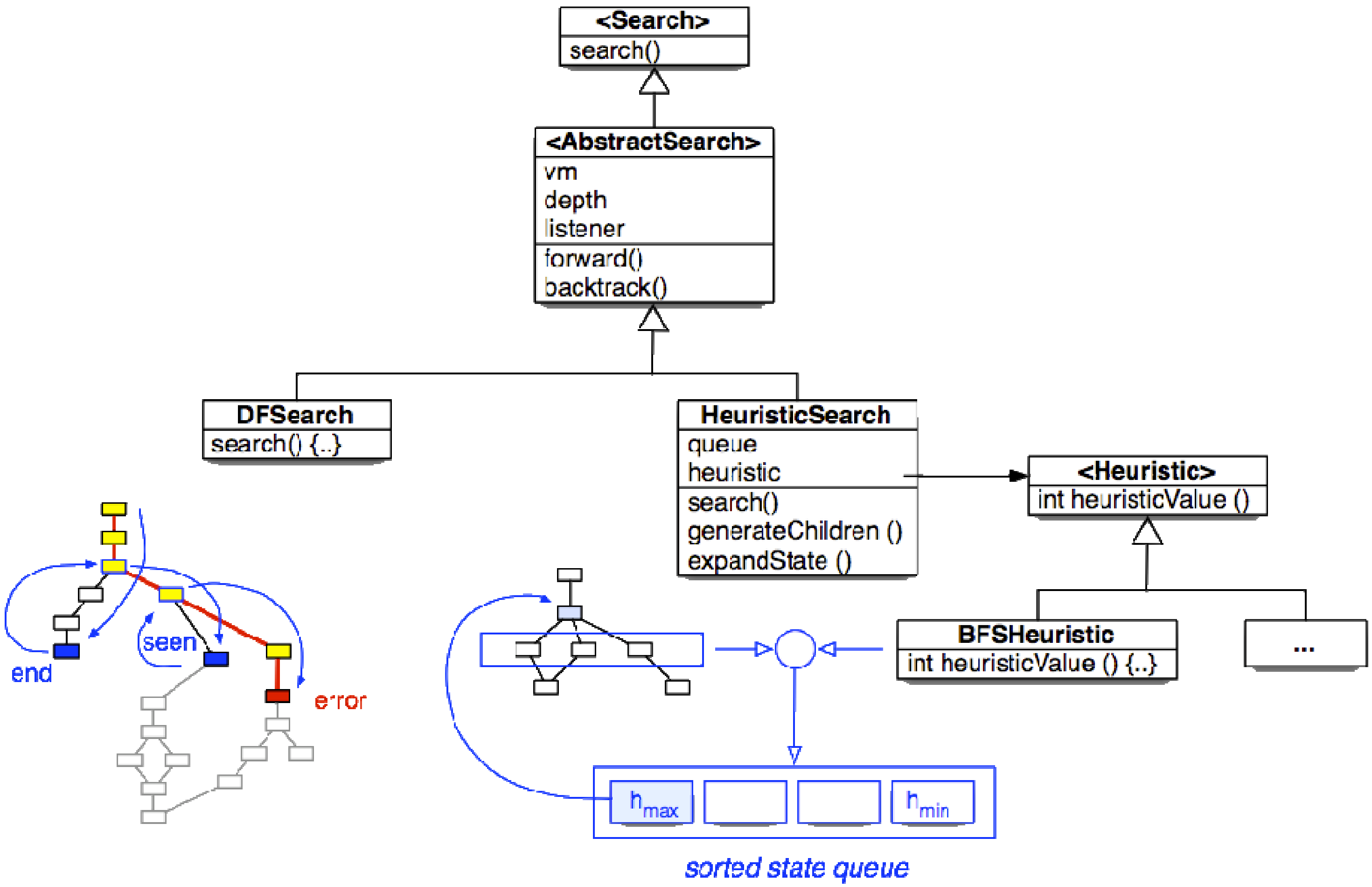
Exercises

Some slides borrowed from JavaPathFinder tutorial at ASE conference 2006

- http://www.visserhome.com/willem/presentations/ase06jpftut.ppt

Search Strategies

Under the Hood - Search

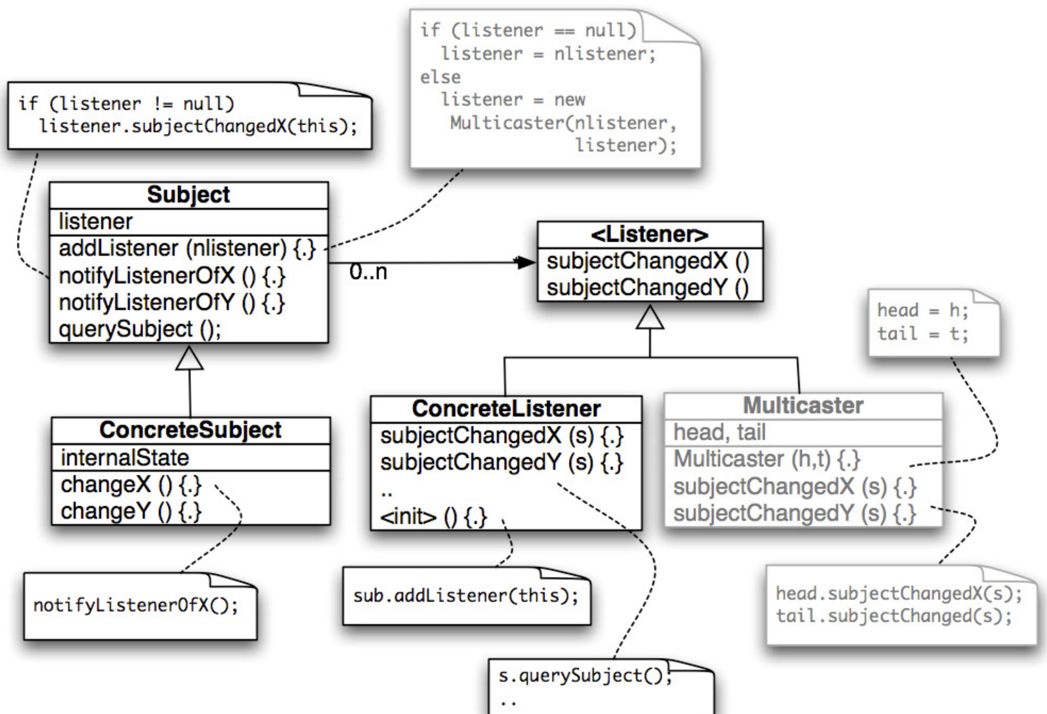


Extending JPF - Listeners

Preferred way of extending JPF: ‘Listener’ variant of the Observer pattern

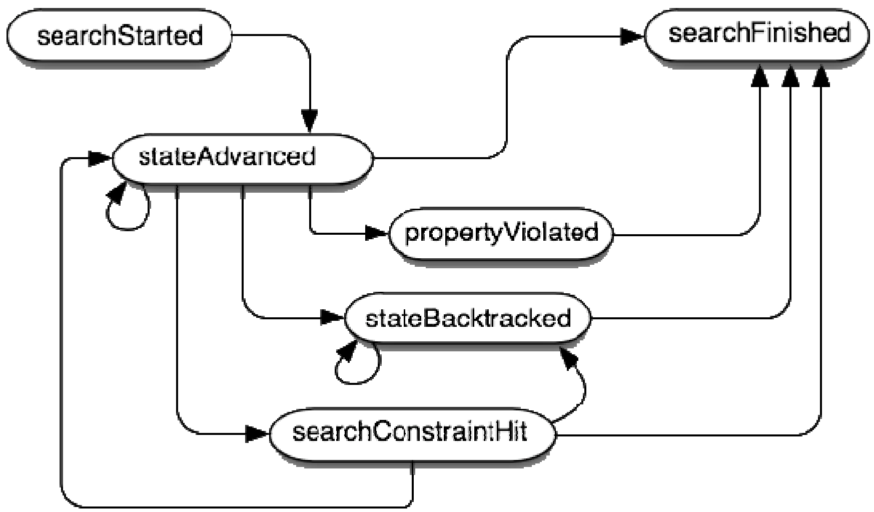
- Keep extensions out of the core classes

Listeners can subscribe to Search and VM events



Extending JPF - SearchListener

```
public interface SearchListener {
 /* got the next state */
 void stateAdvanced (Search search);
 /* state was backtracked one step */
 void stateBacktracked (Search search);
 /* a previously generated state was restored
(can be on a completely different path) */
 void stateRestored (Search search);
 /* JPF encountered a property violation */
void propertyViolated (Search search);
 /* we get this after we enter the search loop, but BEFORE the first
forward */
 void searchStarted (Search search);
 /* there was some contraint hit in the search, we back out could have
been turned into a property, but usually is an attribute of the search, not
the application */
 void searchConstraintHit (Search search);
 /* we're done, either with or without a preceeding error */
 void searchFinished (Search search);
}
```



Extending JPF - VMListener

```
public interface VMListener {
  void instructionExecuted (JVM vm); // VM has executed next instruction
  void threadStarted (JVM vm);       // new Thread entered run() method
  void threadTerminated (JVM vm);    // Thread exited run() method
  void classLoaded (JVM vm);         // new class was loaded
  void objectCreated (JVM vm);       // new object was created
  void objectReleased (JVM vm);      // object was garbage collected
  void gcBegin (JVM vm);             // garbage collection mark phase started
  void gcEnd (JVM vm);               // garbage collection sweep phase terminated
  void exceptionThrown (JVM vm);     // exception was thrown
  void nextChoice (JVM vm);          // choice generator returned new value
}
```

Extending JPF - Listener Example

```
public class HeapTracker extends GenericProperty implements VMListener, SearchListener {
  class PathStat { .. int heapSize = 0; .. }    // helper to store additional state info
  PathStat stat = new PathStat();
  Stack pathStats = new Stack();
  public boolean check (JVM vm, Object arg) {        // GenericProperty
    return (stat.heapSize <= maxHeapSizeLimit);
  }
  public void stateAdvanced (Search search) {       // SearchListener
    if (search.isNewState()) {..
      pathStats.push(stat);
      stat = (PathStat)stat.clone(); ..
  }
  public void stateBacktracked (Search search) {    // SearchListener
    .. if (!pathStats.isEmpty())  stat = (PathStat) pathStats.pop();
  }
  public void objectCreated (JVM vm) {..             // VMListener
    ElementInfo ei = vm.getLastElementInfo();
    ..stat.heapSize += ei.getHeapSize(); ..
  }
  public void objectReleased (JVM vm) {              // VMListener
    ElementInfo ei = vm.getLastElementInfo();
    ..stat.heapSize -= ei.getHeapSize(); ..
  }
}
```

Extending JPF - Listener Configuration

Listeners are usually configured, not hard coded

Per configuration file:

```
search.listener = MySearchListener
vm.listener = MyVMListener
jpf.listener = MyCombinedListener:MySecondListener...
```

Per command line:

```
jpf ... +jpf.listener=MyCombinedListener  ...
```

Hard coded:

```
MyListener listener= new MyListener(..);
..
Config config = JPF.createConfig( args);
JPF jpf = new JPF( config);
jpf. addSearchListener (listener);
jpf. addVMListener ( listener);
jpf.run();
..
```

Partial-Order Reduction

Partial-Order Reduction (POR)

The number of different scheduling combinations is the prevalent factor for the state space size of concurrent programs.

Fortunately, for most practical purposes it is not necessary to explore all possible instruction interleavings for all threads.

The number of scheduling induced states can be significantly reduced by grouping all instruction sequences in a thread that cannot have effects outside this thread itself, collapsing them into a single transition.

This technique is called Partial Order Reduction (POR), and typically results in more than 70% reduction of state spaces.

On-the-Fly POR in JPF

JPF employs an on-the-fly POR that does not rely on user instrumentation or static analysis. JPF automatically determines at runtime which instructions have to be treated as state transition boundaries.

If POR is enabled (configured via *vm.por property*), a forward request to the VM executes all instructions in the current thread until one of the following conditions is met:

1. the next instruction is scheduling relevant
2. the next instruction yields a "nondeterministic" result (i.e. simulates random value data acquisition)

Detection of both conditions are delegated to the instruction object itself (*Instruction.execute(..)*), passing down information about the current VM execution state and threading context.

If the instruction is a transition breaker, it creates a *ChoiceGenerator* and schedules itself for re-execution.

Determining Scheduling Relevance (1)

Each bytecode instruction type corresponds to a concrete *gov.nasa.jpf.Instruction* subclass that determines scheduling relevance based on the following factors:

Instruction Type

Due to the stack based nature of the JVM, only about 10% of the Java bytecode instructions are scheduling relevant, i.e. can have effects across thread boundaries.

The interesting instructions include direct synchronization (*monitorEnter, monitorExit, invokeX* on synchronized methods), field access (*putX, getX*), array element access (*Xaload, Xastore*), and invoke calls of certain Thread (*start(), sleep(), yield(), join()*) and Object methods (*wait(), notify()*).

Determining Scheduling Relevance (2)

Object Reachability

Besides direct synchronization instructions, field access is the major type of interaction between threads.

However, not all *putX / getX* instructions have to be considered, only the ones referring to objects that are reachable by at least two threads can cause data races.

While reachability analysis is an expensive operation, the VM already performs a similar task during garbage collection, which is extended to support POR.

Determining Scheduling Relevance (3)

Thread and Lock Information

Even if the instruction type and the object reachability suggest scheduling relevance, there is no need to break the current transition in case there is no other runnable thread.

In addition, lock acquisition and release (*monitorEnter, monitorExit*) do not have to be considered as transition boundaries if there they happen recursively - only the first and the last lock operation can lead to rescheduling.

Controlling Thread Scheduling (1)

While JPF uses these information to automatically deduce scheduling relevance, there exist three mechanisms to explicitly control transition boundaries (i.e. potential thread interleavings)

Attributor

A configurable concrete class of this type is used by JPF during class loading to determine object, method and field attributes of selected classes and class sets.

The most important attributes with respect to POR are method atomicity and scheduling relevance levels: (a) never relevant, (b) always scheduling relevant, (c) only relevant in the context of other runnables. (d) only relevant of top-level lock.

The default Attributor executes all java.* code atomically, which is can be too aggressive (i.e. can cause BlockedAtomicExceptions).

Controlling Thread Scheduling (2)

VMListener

A listener can explicitly request a reschedule by calling ThreadInfo.yield() in response of a instruction execution notification.

Verify

The Verify class serves as an API to communicate between the test application and JPF, and contains beginAtomic(), endAtomic() functions to control thread interleaving

Software Engineering Institute | Carnegie Mellon