Model checking

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

- thread state (current instruction, stack),
- global variables,
- heap references, and
- trail (path to the state)
Transitions

- Sequence of instructions
- End of transition determined by
  - Multiple successor states (choices)
  - Enforced by listeners (`vm.breakTransition();`)
  - Reached maximal length (configuration `vm.max_transition_length`)
  - End or blocking of current thread

[Diagram showing state transition and choice scheduling]

Scheduling Choice

```java
synchronized (...) {
    wait (...)
    x = mySharedObject
    ...
}
```

Data Choice

```java
boolean b = Verify.getBoolean();
double d = Verify.getDouble("MyHeuristic");
...
```

http://babelfish.arc.nasa.gov/trac/jpf/wiki
## Scheduling Choices

- Which other thread is runnable?
- Partial Order Reduction: Is this thread affected by the current transition?
- Controlled by search and VM

## Data Choices

- Which concrete value to choose for the inputs?
- Mostly configured by the user

## Control Choices

- Which branch in the program to take?
- Explicit invocation schedule by extensions
Implementing Choices

- choices encapsulated in ChoiceGenerators (CGs)
- registered by VM, instructions, extensions, or listeners
- `cg.randomize_choices` configures JPF to randomly explore choices

http://babelfish.arc.nasa.gov/trac/jpf/wiki
Applications, JPF, and JPF-Applications

JPF unaware programs
- runs on any JVM

JPF enabled programs
- annotate program
  - requirements
  - sequences (UML)
  - contracts (PbC)
  - tests
  - ... analyze program
  - symbolic exec
  - test data
  - thread safety / races

JPF dependent programs
- restricted application models
  - UML statemachines
  - does not run w/o JPF libraries
- initial domain impl. costs
  - domain libs can be tricky

Constraints
- runtime costs
  - order of magnitude slower
  - state storage memory
- standard library support
  - java.net, javax.swing, ...
  - (needs abstraction models)
- functional property impl. costs
  - listeners, MJ1 knowledge
- restricted choice types
  - scheduling sequences
  - java.util.Random

Benefits
- non-functional properties
  - unhandled exceptions
    - (incl. AssertionError)
  - deadlocks
  - races
- improved inspection
  - coverage statistics
  - exact object counts
  - execution costs
- low modeling costs
  - statemachine w/o layout hassle...
- functional (domain) properties
  - built-in into JPF libraries
- flexible state space
  - domain specific choices
    - (e.g. UML "enabling events")
- runtime costs & library support
  - usually not a problem, domain libs can control state space

http://babelfish.arc.nasa.gov/trac/jpf/wiki
**Interfering with the Search (1/2)**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getBoolean</code></td>
<td>Get a Boolean CG</td>
</tr>
<tr>
<td><code>getInt</code></td>
<td>Get a named integer CG</td>
</tr>
<tr>
<td><code>getIntFromList</code></td>
<td>Get an integer CG initialized from a list</td>
</tr>
<tr>
<td><code>getObject</code></td>
<td>Get a named object CG</td>
</tr>
<tr>
<td><code>getDouble</code></td>
<td>Get a named double CG</td>
</tr>
<tr>
<td><code>getDoubleFromList</code></td>
<td>Get a double CG initialized from a list</td>
</tr>
<tr>
<td><code>getLongFromList</code></td>
<td>Get a long CG initialized from a list</td>
</tr>
<tr>
<td><code>getFloatFromList</code></td>
<td>Get a float CG initialized from a list</td>
</tr>
<tr>
<td><code>random</code></td>
<td>Get a CG for random values</td>
</tr>
<tr>
<td><code>randomBool</code></td>
<td>Get a Boolean CG</td>
</tr>
</tbody>
</table>
### Interfering with the Search (2/2)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gov.nasa.jpf.jvm.Verify</code></td>
<td>For transitions and states</td>
</tr>
<tr>
<td><code>addComment</code></td>
<td>Add a comment to a state</td>
</tr>
<tr>
<td><code>instrumentPoint</code></td>
<td>Add a label to a state</td>
</tr>
<tr>
<td><code>atLabel</code></td>
<td>Check for a label</td>
</tr>
<tr>
<td><code>boring</code></td>
<td>Hint an uninteresting state</td>
</tr>
<tr>
<td><code>interesting</code></td>
<td>Conditionally hint an interesting state</td>
</tr>
<tr>
<td><code>ignoreIf</code></td>
<td>Conditionally prune the search space</td>
</tr>
<tr>
<td><code>beginAtomic</code></td>
<td>Start an atomic block</td>
</tr>
<tr>
<td><code>endAtomic</code></td>
<td>End an atomic block</td>
</tr>
<tr>
<td><code>breakTransition</code></td>
<td>End the current transition</td>
</tr>
</tbody>
</table>
Properties

- Configured with `search.properties`
- Evaluated after every transition
- Base class: `gov.nasa.jpf.Property`
- Properties shipped with JPF Core:
  - `gov.nasa.jpf.jvm.IsEndStateProperty`
  - `gov.nasa.jpf.jvm.NoOutOfMemoryErrorProperty`
  - `gov.nasa.jpf.jvm.NotDeadlockedProperty`
  - `gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty`
Listener

- Configured with listener and listener.autoload

- Different types:
  - `VMLListener` notified about executed instructions, threads state changes, loaded classes, created objects, object monitor events, garbage collections, choice generators, and method enter and exit events
  - `SearchListener` notified about state changes, property violations, and search related events

- Implementation basis for many extensions

- Idea: JPF can check what you can program

- JPF Core comes with many listeners in package `gov.nasa.jpf.listener`
How Listeners Work

- VM or search notifies listener about next or previous event.
- Listener can act upon this event.
- Listeners can influence VM or search.
- Can annotate objects, fields, operands, and variables with attributes
Writing Our First Listener
A *user-specified set of fields and variables* *should never be assigned to* `null`.

**Chopped into Pieces**

- configurable field and variable description
- check for variable and field assignment
Desired property can be violated by writing a field or variable. This does not necessarily break a transition. We need a listener to break the transition and report an error.
Utility for specifying field descriptions:

- `x.y.Foo.bar` field `bar` in class `x.y.Foo`
- `x.y.Foo+.bar` all `bar` fields in `x.y.Foo` and all its supertypes
- `x.y.Foo.*` all fields of `x.y.Foo`
- `*.myData` all fields names `myData`
- `!x.y.*` all fields of types outside types in package `x.y`
### Using Utilities (2/2)

#### MethodSpec

Utility for specifying methods:

- `x.y.Foo.*` all methods of class `x.y.Foo`
- `*(x.y.MyClass)` all methods that take exactly one parameter which is of type `x.y.MyClass`
- `!x.y.*.*(int)` no method of any class in package `x.y` or any subpackage that takes exactly one argument that is an `int`

#### VarSpec

Utility for specifying local variable descriptions:

Syntax: `MethodSpec:VariableName`
public NonNullChecker(Config conf) {
    Set<String> spec = conf.getStringSet("nnc.fields");
    if (spec == null)
        spec = Collections.emptySet();
    nonNullableFields = new FieldSpec[spec.size()];
    int i = -1;
    for (String field : spec)
        nonNullableFields[++] = FieldSpec.createFieldSpec(field);
    spec = conf.getStringSet("nnc.vars");
    if (spec == null)
        spec = Collections.emptySet();
    nonNullableVars = new VarSpec[spec.size()];
    i = -1;
    for (String var : spec)
        nonNullableVars[++] = VarSpec.createVarSpec(var);
}
Observation
Only two instructions can assign \texttt{null} to a field:
- putfield
- putstatic

Basic Idea
If such an instruction wrote to a field we are interested in, check value of that field.

\texttt{instructionExecuted} notification
private void checkFieldInsn(FieldInstruction insn) {
    if (isRelevantField(insn)) {
        if (isNullFieldStore(insn)) {
            storeError(vm, insn);
            vm.breakTransition();
        }
    }
}

private boolean isRelevantField(FieldInstruction insn) {
    if (!insn.isReferenceField())
        return false;
    FieldInfo fi = insn.getFieldInfo();
    for (FieldSpec fieldSpec : nonNullableFields) {
        if (fieldSpec.matches(fi)) {
            return true;
        }
    }
    return false;
}

private boolean isNullFieldStore(FieldInstruction insn) {
    FieldInfo fi = insn.getFieldInfo();
    ElementInfo ei = insn.getLastElementInfo();
    return ei.getFieldValueObject(fi.getName()) == null;
}
Observation

Only one instruction can assign null to a local variable:

- astore

We can use our method from before to check that.
private void checkLocalVarInsn(ASTORE insn) {
    if (isRelevantVar(insn)) {
        if (isNullVarStore(insn)) {
            storeError(vm, insn);
            vm.breakTransition();
        }
    }
}

private boolean isRelevantVar(ASTORE insn) {
    int slotIdx = insn.getLocalVariableIndex();
    MethodInfo mi = insn.getMethodInfo();
    int pc = insn.getPosition() + 1;

    for (VarSpec varSpec : nonNullableVars) {
        if (varSpec.getMatchingLocalVarInfo(mi, pc, slotIdx) != null)
            return true;
    }
    return false;
}

private boolean isNullVarStore(ASTORE insn) {
    ThreadInfo ti = vm.getLastThreadInfo();
    int slotIdx = insn.getLocalVariableIndex();
    return ti.getObjectLocal(slotIdx) == null;
}
Demo