Side Effect Monitoring for Java using Bytecode Rewriting

Manuel Geffken, Peter Thiemann

2014-09-19
Motivation
We care about Heap Side Effects!

Heap side effects

- Accessed heap memory locations
  - Read from field
  - Write to field
- A pure method does not write to any heap locations

Applications of side effect control

- Ensure purity of program assertions and specifications
- Validate correctness of program optimizations
- Improve program understanding
Introduction

Why Monitoring?

Monitoring is a successful approach to checking a specification.

- “Design by Contract”: JML toolchain uses combination of verification and run-time monitoring.
- **Monitoring allows fine grained properties to be checked** that cannot be verified statically.

Side effect monitoring

- Control side effects at run time
- Specify permitted side effects
- Report violations
Method contracts as specifications

@Grant access to memory locations

Access paths to specify memory locations

Example

class TreeNode {
    double mass;
    TreeNode left, right;

    @Grant("this.mass")
    void setMass (double mass) {
        this.mass = mass;
    }
}
Syntactic domains

access permission contracts \( c ::= x_1.e_1, \ldots, x_n.e_n \)

anchors \( x \in \text{this} + \text{Param} + \text{Class} \)

fields \( f \in \text{Field} \)

regular path expressions \( e ::= \varepsilon \mid f \mid e.e \mid e\mid e \mid e* \)
Path Expression Semantics

- We specify **read and write effects in single language**
- Access paths in the language describe **writable** locations
- Proper prefixes describe **read-only** locations
- Append `@` to path expression to give it **read-only** permission

class TreeNode {
    double mass; TreeNode left, right;

    @Grant("this.(left|right).mass")
    void setMassLeftRight (double mass) {
        this.left.mass = mass;
        this.right.mass = mass;
    }

    @Grant("this.mass.@")
    void getMass () {
        return this.mass;
    }
}
Semantic Principles

**Noninterference**  Instrumented program not violating any contract behaves as corresponding uninstrumented program

**Location-based semantics**  Permissions are associated with memory locations

**Prestate principle**  Access Permission Contracts refer to method’s prestate; locations in newer objects are unrestricted

**Dynamic extend**  Contract is in force from method entry to matching return
Global monitor state has **two stacks** that **grow in lockstep** as annotated methods are activated (dynamic extend) to

1. maintain **permission map** for each **active method contract** (location-based semantics)
   
   `permissions`  Maps memory locations to **access rights**

   ```java
   enum Access {READ, WRITE};
   Stack<Map<Object, Map<Field, EnumSet<Access>>>> permissions;
   ```

2. **grant full access to** locations in **new objects** (prestate principle)

   `newObjs`  Newly allocated objects since method activation

   ```java
   Stack<Set<Object>> newObjs;
   ```
Contract installation at method entry (prestate principle)

- Calculate new permission map based on contract and current heap (prestate)
- Make sure a new contract installation respects contracts that are already in force
- Install effective permissions

```java
void installContract(Contract c) {
    contractPerm = calculatePermissions(c);

    currentPerm = permissions.peek();
    nextPerm = contractPerm \ currentPerm;

    permissions.push(nextPerm);
    newObjs.push(∅);
}
```
calculatePermissions(Contract c)

- Construct automaton from contract c:
  \[ a \cdot b \cdot (c \mid d)^* \rightarrow s_0 \xrightarrow{a} s_1 \xrightarrow{b} s_2 \]

- Depth-first heap traversal from contract's anchors
  - Field names of visited fields as automaton input
  - Ensure termination: Maintain set of pairs of object and automaton states
Contract Installation
Depth-first Heap Traversal: Example

Visible heap:

Object stack:

Contract automaton:

Already visited pairs of object and automaton states:

\{ ⟨o_x, \{s_0\}⟩ \}
Visible heap:

Object stack:
\(o_x, o_1\)

Contract automaton:

Already visited pairs of object and automaton states:
\[\{\langle o_x, \{s_0\}\rangle, \langle o_1, \{s_1\}\rangle\}\]
Visible heap:

Object stack:
\( o_x, o_1, o_2 \)

Already visited pairs of object and automaton states:
\[ \{ \langle o_x, \{ s_0 \} \rangle, \langle o_1, \{ s_1 \} \rangle, \langle o_2, \{ s_2 \} \rangle \} \]
Visible heap:

Contract automaton:

Object stack:

Already visited pairs of object and automaton states:
Visible heap:

```
x → o_x → o_1 → o_2 → o_3 → o_4 → o_5 → ...  
  a   b   c   d  
  f
```

Object stack:

```
o_x, o_1, o_2, o_4
```

Contract automaton:

```
s_0 → a → s_1 → b → s_2 → c → d  
```

Already visited pairs of object and automaton states:

```
{ ⟨o_x, {s_0}⟩, ⟨o_1, {s_1}⟩, ⟨o_2, {s_2}⟩,  
  ⟨o_3, {s_2}⟩, ⟨o_4, {s_2}⟩ }  
```
Visible heap:

Contract automaton:

Object stack:

Already visited pairs of object and automaton states:

\{ \langle o_x, \{ s_0 \} \rangle, \langle o_1, \{ s_1 \} \rangle, \langle o_2, \{ s_2 \} \rangle, \\
\langle o_3, \{ s_2 \} \rangle, \langle o_4, \{ s_2 \} \rangle \}
**Contract Installation**
Depth-first Heap Traversal: Example

Visible heap:

```
Visible heap:
```

Object stack:
```
```

Contract automaton:
```
```

Already visited pairs of object and automaton states:
```
\{ \langle o_x, \{s_0\} \rangle, \langle o_1, \{s_1\} \rangle, \langle o_2, \{s_2\} \rangle, \\
\langle o_3, \{s_2\} \rangle, \langle o_4, \{s_2\} \rangle, \langle o_5, \emptyset \rangle \}\```
Contract Installation
Depth-first Heap Traversal: Example

Visible heap:

Object stack:

Contract automaton:

Already visited pairs of object and automaton states:

\[
\{ \langle o_x, \{ s_0 \} \rangle, \langle o_1, \{ s_1 \} \rangle, \langle o_2, \{ s_2 \} \rangle, \langle o_3, \{ s_2 \} \rangle, \langle o_4, \{ s_2 \} \rangle, \langle o_5, \emptyset \rangle \}\]
void visit(Object obj, String fieldname, Object fieldvalue) {
   // remember automaton state

   // step automaton according to field name to get access right

   // continue depth-first heap traversal
   ((Traversable)fieldvalue).traverse(this);

   // restore remembered automaton state
}

class C extends D {
   F f;
   G g;
}
Heap Traversal
Visitor Pattern with Specialized Traversal Code

```java
void visit(Object obj, String fieldname, Object fieldvalue) {
    // remember automaton state

    // step automaton according to field name to get access right

    // continue depth-first heap traversal
    ((Traversable)fieldvalue).traverse(this);

    // restore remembered automaton state
}
```

class C extends D implements Traversable {
    F f;
    G g;

    void traverse(Traversal t) {
        t.visit(this, "f", this.f);
        t.visit(this, "g", this.g);
        super.traverse(t);
    }
}
Major Instrumentation Points

- Method entry
  - Heap traversal
- Method exit
  - Pop from permissions stacks
  - Merge top set of newObjects stack into underlying entry
Major Instrumentation Points

- Method entry
  - Heap traversal
- Method exit
  - Pop from permissions stacks
  - Merge top set of newObjects stack into underlying entry

- Field access: Get access right for accessed location
- Object creation: Add object to top set of newObjects stack
Field Access Instrumentation

Example

A simple getter method

```java
public F getF() {
    // get access right for location to access
    EnumSet<Access> accessRight = getAccessRight(this, "f");
    // Is location's access right sufficient for this access?
    if (!accessRight.contains(Access.READ))
        throw new ReadViolation();

    return this.f;
}
```
A simple getter method

```java
public F getF() {
    // get access right for location to access
    EnumSet<Access> accessRight = getAccessRight(this, "f");

    // Is location's access right sufficient for this access?
    if (!accessRight.contains(Access.READ))
        throw new ReadViolation();

    return this.f;
}
```
Example

```java
Obj o = new Obj();
```
Example

```
Obj o = new Obj();

class Obj {
    Obj() {
        super();
        // add new object to top of newObjs stack
        newObjs.peek().add(this);

        // monitor body but allow access to own fields
        ...
    }
}
```
Implementation

- **Javassist** for bytecode instrumentation
- **Offline transformation**: classfile $\rightarrow$ instrumented classfile
- **Online transformation**: Instrumentation at runtime using custom class loader
- **Weak collections** for permission maps and sets of new objects
@Grant("this.(left|right)*.force, this.(left|right)*.mass.@")

void computeForces() {
    this.force = this.mass * G;

    if (left != null)
        left.computeForces();

    if (right != null)
        right.computeForces();
}
balanced tree

\( t(s) \)

\begin{align*}
\text{nodes (1000)} \\
O(n \log n)
\end{align*}

degenerate tree

\( t(s) \)

\begin{align*}
\text{nodes (100)} \\
O(n^2)
\end{align*}
void computeForcesUnannotated() {
    this.force = this.mass * G;

    if (left != null)
        left.computeForces();

    if (right != null)
        right.computeForces();
}

@Grant("this.(left|right)*.force, this.(left|right)*.mass.@")
public void computeForcesDelegator() {
    computeForcesUnannotated();
}
@Grant("this.force, this.mass.@")
public void computeForce() {
    this.force = this.mass * G;
}

Accumulated runtime of 1000 runs
Evaluation: Micro Benchmarks
Uninstrumented and Instrumented (No Annotations)

- `computeForcesUnannotated`
- Accumulated runtime of 1000 runs
Major Instrumentation Points

Overhead

- Method entry
  - Heap traversal
    - Expensive for large heap and contracts covering large parts of the heap
    - Cheap for small heaps, flat (small) contracts, unannotated methods

- Method exit
  - Pop from permissions stacks
  - Merge top set of newObjects stack into underlying entry
    - Repeated union can be expensive for recursive methods

- Field access: Get access right for accessed location
- Object creation: Add object to top set of newObjects stack
Major Instrumentation Points

Overhead

- Method entry
  - Heap traversal
    - Expensive for large heap and contracts covering large parts of the heap ✗
    - Cheap for small heaps, flat (small) contracts, unannotated methods ✓
- Method exit
  - Pop from permissions stacks ✓
  - Merge top set of newObjects stack into underlying entry
    - Repeated union can be expensive e.g. for recursive methods
  - Avoid explicit union: Establish relative order (contract generation) between installed contracts and objects ✓
    - Only maintain each installed contract's generation on stack
- Field access: Get access right for accessed location ✓
- Object creation: Add object to top set of newObjects stack
  Add object to map from objects to their contract generations ✓
### Evaluation: Larger Benchmarks

#### Ashes Suite Collection

#### Evaluation

- **“ashesHardTestSuite”** benchmarks from the Ashes Suite Collection
- Annotations using default contracts covering the whole heap (worst case), e.g.

  ```java
  @Grant("Matrix.*, Test.*, this.*, a.*, b.*")
  public void matmul(Matrix a, Matrix b) {
      ...
  }
  ```

<table>
<thead>
<tr>
<th>Application</th>
<th>Methods</th>
<th>Baseline Run time (s)</th>
<th>Unannotated Instrumented run time (s)</th>
<th>Fully annotated Traversed objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>boyer</td>
<td>75</td>
<td>0.27</td>
<td>0.35</td>
<td>9733.82</td>
</tr>
<tr>
<td>decode</td>
<td>19</td>
<td>0.23</td>
<td>0.32</td>
<td>395.40</td>
</tr>
<tr>
<td>fft</td>
<td>10</td>
<td>0.06</td>
<td>0.08</td>
<td>0.10</td>
</tr>
<tr>
<td>illness</td>
<td>93</td>
<td>0.14</td>
<td>0.22</td>
<td>1923.01</td>
</tr>
<tr>
<td>lexgen</td>
<td>210</td>
<td>0.33</td>
<td>0.52</td>
<td>7634.90</td>
</tr>
<tr>
<td>lu</td>
<td>7</td>
<td>0.24</td>
<td>0.26</td>
<td>0.64</td>
</tr>
<tr>
<td>machineSim</td>
<td>108</td>
<td>0.12</td>
<td>0.16</td>
<td>23.42</td>
</tr>
<tr>
<td>matrix</td>
<td>10</td>
<td>0.17</td>
<td>0.26</td>
<td>792.57</td>
</tr>
<tr>
<td>probe</td>
<td>6</td>
<td>0.70</td>
<td>0.73</td>
<td>1.26</td>
</tr>
</tbody>
</table>

* We excluded two programs: javazoom (exceeded maximal method size) and puzzle (baseline did not run).
Future Work

- Reduce “worst case” overhead from heap traversal
  - Combination of dynamic monitoring and static side-effect analysis
  - Investigate “lazy” contract installation while adhering to the prestate-principle
  - Avoid repeated contract installations for recursive functions
- More realistic case studies
Questions?

Thank you for your interest.

Any questions?