Automated Construction of Reasonable Environment for Java Components

Pavel Parízek, Jiří Adámek, Tomáš Kalibera

DISTRIBUTED SYSTEMS RESEARCH GROUP
http://dsrg.mff.cuni.cz

CHARLES UNIVERSITY PRAGUE
Faculty of Mathematics and Physics
Software quality

• Key attribute of software quality
  - Absence of errors (bugs) in code

• How to discover errors in code
  - Software testing
  - Formal verification and analysis
    • model checking, theorem proving, static analysis
Model checking

• Automated verification technique
  ▪ Good at detection of errors in concurrent and reactive systems
  ▪ Used for a long-time in verification of both software and hardware systems

• Issue
  ▪ Does not scale well to large systems
    • State explosion
Model checking of component systems

• Two aspects
  ▪ Compositional approach to verification
  ▪ Independent development of components
Compositional model checking

- Key idea
  - Components are checked separately (one at a time)
- Benefit
  - Model checking is less prone to state explosion
Independent development of components

- Each component may be developed by a different team (or at a different time)
  - Rest of a system may not be available upon completion of a component
Model checking of component systems

• Two aspects
  ▪ Compositional approach to verification
  ▪ Independent development of components

• Key requirement on techniques and tools
  ▪ Ability to verify an isolated component
Our research

• Java component
  ▪ Fragment of Java code with well-defined interface
  ▪ Example: HashMap with interface Map
    ```java
    public interface Map<K,V> {
      V get(Object key);
      V put(K key, V value);
      V remove(Object key);
      ...
    }
    ```

• Our goal: discovery of concurrency errors in isolated Java components
  ▪ Using Java PathFinder model checker (JPF)
Model checking of isolated Java components

- General issue: problem of missing environment
  - JPF works only for complete Java programs (with `main`)

- Solution: construction of artificial environment
  - Fragment of Java code with `main`
    - Simulates behavior of a specific real environment for C
  - Artificial environment + component = complete program

```
Java program

C  →  C               E
```

Automated Construction of Reasonable Environment
FESCA 2009
class EnvThread extends Thread
{
    public void run()
    {
        while (!done)
        {
            map.put(rndKey(), rndValue());
            Object o = map.get(rndKey());
            map.remove(rndKey());
        }
    }
}

public static void main(String[])
{
    Map map = new HashMap();

    Thread th1 =
        new EnvThread(map).start();
    Thread th2 =
        new EnvThread(map).start();

    // wait for threads to finish
    th1.join();
    th2.join();
}
Construction of artificial environment

• Code of the artificial environment
  ▪ Written completely by the developer (by hand)
  ▪ Generated from a high-level model
    • Model of environment’s behavior: sequences and parallel executions of methods
    • Values of method parameters

• High-level model
  ▪ Again written by the developer
  ▪ Constructed in an automated way
Criteria for definition of environment’s model

- Coverage of component’s code
  - Artificial environment should trigger most errors in code
- Performance and scalability of checking
  - Model checker should find at least some errors
    - Before running out of memory because of state explosion

- Goal of application of a model checker
  - Verify that the component is free of errors
    - High coverage, low performance and scalability
  - Discover at least some errors in code
    - Low coverage, better performance and scalability
Model of environment’s behavior

• Typical options
  - Universal environment
    
    \[
    \text{get}^* \mid .. \mid \text{get}^* \mid \text{put}^* \mid .. \mid \text{remove}^* \mid ..
    \]
  - Context-specific environment
  - Environment optimized for discovery of specific errors in code
Environment for discovery of concurrency errors

• Options
  - Randomly ordered sequence of method pairs
    (put | put); (get | remove); (get | put); ...
  - List of method pairs selected via error patterns
    (remove | put); (remove | remove); (remove | get); ...

  Assumption: access to an important variable is not synchronized in the remove method

• Key idea
  - Selection of method pairs to be executed in parallel
  - Rationale
    • Most concurrency errors can be triggered by two threads
Our approach

- **Goal**
  - Efficient discovery of concurrency errors in Java code of isolated components
  - Addressing drawbacks of existing techniques
    - Including our earlier work (error patterns)

- **Model of environment’s behavior**
  - Automatically constructed on the basis of static analysis and a software metric

- **Values of method parameters**
  - Defined manually by the user
Key ideas

• Concurrency error can be triggered only if
  ▪ Methods are executed in parallel (in different threads)
  ▪ Methods interact via concurrency constructs of Java
    • Accesses to shared variables (fields, objects, …)
    • Synchronization (locks, calls of wait and notify)

• High degree of interaction between methods implies higher chance of an error
  ▪ Assessment of the interaction → interaction metric
Interaction between Java methods – example

```java
class HashMap {
    private final int capacity;
    private int size;
    private V[] data;

    public V put(K key, V value) {
        int h = hash(key);
        V old = data[h];
        data[h] = value;
        size++;
        return old;
    }

    public V remove(Object key) {
        int h = hash(key);
        V value = data[h];
        data[h] = null;
        size--;
        return value;
    }

    public V get(Object key) {
        int h = hash(key);
        V value = data[h];
        return value;
    }
}
```
Our approach – algorithm

• Input
  - List of all possible sets of component’s methods
  - Example: \{put, get\}, \{put, put\}, \{put, remove\}, ...

• Two steps are performed for each method set
  1) Static analysis of methods’ code
     - Identifies accesses to shared variables in methods in each set
  2) Application of interaction metric
     - Computes the degree of mutual interaction among methods on the basis of numbers of accesses to shared variables

• Output
  - Sequence of method sets that is ordered according to the metric
  - Example: \{put, remove\}, ..., \{put, get\}, ...
Automated Construction of Reasonable Environment

**Whole picture – verification process**

- **Isolated component (Java code)**
- **JPF**
- **Reasonable artificial environment (Java code)**
- **Verification results**
- **Static analysis**
- **Environment generator**
- **Reasonable behavior model of artificial environment**
- **Artificial environment construction**
- **Interaction metric computation**
- **Information on concurrent method interaction**
- **Specification of method parameter values (Java class)**

---

Automated Construction of Reasonable Environment
FESCA 2009
Experiments

• Interaction metric is configurable
  ▪ Different configurations may work better in different cases

• Non-trivial components
  ▪ `ConcurrentHashMap` from `java.util.concurrent`
  ▪ Daisy file system
    ▪ Used as an assignment for challenge problem at CAV/ISSTA 2004

<table>
<thead>
<tr>
<th></th>
<th>Daisy file system</th>
<th>ConcurrentHashMap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Memory</td>
</tr>
<tr>
<td>Metric cfg 1 (EEN)</td>
<td>1 s</td>
<td>52 MB</td>
</tr>
<tr>
<td>Metric cfg 2 (UUS)</td>
<td>9782 s</td>
<td>1023 MB</td>
</tr>
<tr>
<td>Random pairs</td>
<td>1715 s</td>
<td>310 MB</td>
</tr>
</tbody>
</table>
Evaluation

• Results of experiments show that our approach
  ▪ Significantly reduces time and memory needed for
ergency of errors in some components
  ▪ Does not help for some other components

• Different configurations of the interaction metric
  yield very different results
  ▪ Results depend on the metric’s configuration
Future work

- Use of more precise static analysis of Java code
  - Goal: more precise assessment of the degree of interaction among Java methods
    - Values of variables (data-flow) should be taken into account
    - Options: points-to analysis, shape analysis

- Design of several metrics and their comparison
  - Several different metric functions

- Extensive evaluation on many components
  - Problem: how to get complex Java components with concurrency errors
    - We plan to use fault injection (seeding) techniques