Using Combinatorial Benchmark Construction to Improve the Assessment of Concurrency Bug Detection Tools

Jeremy S. Bradbury\textsuperscript{a}, Itai Segall\textsuperscript{b}, Eitan Farchi\textsuperscript{b}, Kevin Jalbert\textsuperscript{a}, David Kelk\textsuperscript{a}

\textsuperscript{a}Software Quality Research Group
University of Ontario Institute of Technology
Oshawa, Ontario, Canada

\textsuperscript{b}IBM Haifa Research Laboratory
Haifa, Israel

jeremy.bradbury@uoit.ca, itais@il.ibm.com, farchi@il.ibm.com, kevin.jalbert@uoit.ca, david.kelk@uoit.ca
Concurrency Testing

• Provide code/requirements coverage metrics plus coverage of interleaving space!
  1. Coverage-based testing with manual interleaving explorations
    • Examples: using different OS, different hardware configurations, hand-instrumentation of delays.
  2. Testing with noise makers
    • Example: IBM’s Concurrent Testing Tool [EFN+02] which automatically instruments source code delays

Background

Static Analysis

• Use techniques like call-graph analysis and lock analysis to identify potential bugs without executing the software
• Trade-off compared to testing – improved performance but possibility of spurious results.
• **Examples:** FindBugs [HP04], JLint [Art01], Chord [NPSG09], JSure [JSu], JTest [Par] and RSAR [LHDQ10]

Software Model Checking

• A formal methods approach involving finite state modelling of a software system

• Uses exhaustive state space search to explore all interleavings – can also use heuristic search

• **Examples:** Java Pathfinder (JPF) [HP00], Bogor [RDH03]


Comparing Concurrency Bug Detection Tools

```java
package account;

//import java.lang;

public class Account {
    double amount;
    String name;

    //constructor
    public Account(String nm, double amnt) {
        amount = amnt;
        name = nm;
    }

    //functions
    synchronized void deposit(double money) {
        amount += money;
    }

    synchronized void withdraw(double money) {
        amount -= money;
    }

    synchronized void transfer(Account ac, double mn) {
        amount -= mn;
        ac.amount += mn;
    }

    synchronized void print() {
        System.out.println(name + "--" + amount);
    }

} //end of class Account

package account;

public class ManageAccount extends Thread {
    Account account;
    static Account[] accounts = new Account[10];
    static int num = 2; //the number of the accounts
    static int accNum = 0;
    int i; //the index

    public ManageAccount(String name, double amount) {
        account = new Account(name, amount);
        i = accNum;
        accounts[i] = account;
        accNum = (accNum + 1) % num;
    }

    public void run() {
        account.deposit(300);
        account.withdraw(100);
        Account acc = accounts[(i + 1) % num];
        account.transfer(acc, 99); // remain in here
    }

    static public void printAllAccounts() {
        for (int j = 0; j < num; j++) {
            if (ManageAccount.accounts[j] != null) {
                ManageAccount.accounts[j].print();
            }
        }
    }
}
```
Is this a good program to assess bug detection tool fitness?
Comparing Concurrency Bug Detection Tools

package account;

//import java.lang.*;

public class Account {
    double amount;
    String name;

    //constructor
    public Account(String nm, double amnt) {
        amount = amnt;
        name = nm;
    }

    //functions
    synchronized void deposit(double money) {
        amount += money;
    }

    synchronized void withdraw(double money) {
        amount -= money;
    }

    synchronized void transfer(Account ac, double mn) {
        amount -= mn;
        ac.amount += mn;
    }

    synchronized void print() {
        System.out.println(name + "-" + amount);
    }

} //end of class Account

package account;

public class ManageAccount extends Thread {

    Account account;
    static Account[] accounts = new Account[10];
    static int num = 2; // the number of the accounts
    static int accNum = 0;
    int i; // the index

    public ManageAccount(String name, double amount) {
        account = new Account(name, amount);
        i = accNum;
        accounts[i] = account;
        accNum = (accNum + 1) % num;
    }

    public void run() {
        account.deposit(300);
        account.withdraw(100);
        Account acc = accounts[i + 1] % num;
        account.transfer(acc, 99);

        static public void printAllAccounts() {
            for (int j = 0; j < num; j++) {
                if (ManageAccount.accounts[j] != null) {
                    ManageAccount.accounts[j].print();
                }
            }
        }

    }

} //end of class ManageAccount
Motivation

Challenge

• How do we assess the fitness of a particular concurrency bug detection tool?
  • How do we compare it with other tools?

Solution

• Empirical methods + unbiased data
• Where can we get unbiased data (i.e., programs with real concurrency bugs?) – need a benchmark!
What is a benchmark?

A benchmark is composed of three parts [SEH03]:

1. Creation of a motivating comparison

2. Development of a task sample (i.e., benchmark data)

3. Identification or development of performance measures

Developing a Concurrency Benchmark

1. Building a combinatorial model

2. Pairwise construction

3. Acquiring benchmark examples
Building a Combinatorial Model

- **Combinatorial Test Design (CTD)** is a well-known test planning technique
  - The test space is modelled by a set of **parameters**, their respective **values**, and restrictions on the value combinations.
  - The most common application of CTD is **pairwise testing** – covers the interaction of every pair of parameters
  - A test set that covers all possible pairs of parameter values can typically detect **50%-75%** of the bugs in a program [DKL+99, TL02].

Parameter & Value Selection

- We need to identify the set of parameters and values that characterize programs in the benchmark

Program Size

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of statements</td>
<td>&lt; 10k</td>
<td>10k-100k</td>
<td>&gt; 100k</td>
</tr>
<tr>
<td>Number of critical regions</td>
<td>&lt; 5</td>
<td>5-20</td>
<td>&gt; 20</td>
</tr>
<tr>
<td>Percentage of statements in critical regions</td>
<td>&lt; 5%</td>
<td>5-15%</td>
<td>&gt; 15%</td>
</tr>
</tbody>
</table>
Building a Combinatorial Model (3)

Number of Threads

- Maximum number of threads executing in parallel during program execution

<table>
<thead>
<tr>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
<th>Very Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 5</td>
<td>5-10</td>
<td>10-100</td>
<td>&gt; 100</td>
</tr>
</tbody>
</table>
Building a Combinatorial Model (4)

Path Error Density
• The probability of a thread interleaving manifesting a bug

<table>
<thead>
<tr>
<th>Very Low</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 2%</td>
<td>2-25%</td>
<td>25-75%</td>
<td>&gt; 75%</td>
</tr>
</tbody>
</table>

Bug Depth
• Minimum depth along a path that a bug can be exhibited. Measured in number of context switches.

<table>
<thead>
<tr>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 25</td>
<td>25-50</td>
<td>&gt; 50</td>
</tr>
</tbody>
</table>
Building a Combinatorial Model (5)

Bug Pattern Type
• The kind of bug exhibited by the program.

<table>
<thead>
<tr>
<th>Bug Pattern Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Nonatomic operations assumed to be atomic</td>
</tr>
<tr>
<td>• Two-state access</td>
</tr>
<tr>
<td>• Wrong long or no lock</td>
</tr>
<tr>
<td>• Double-checked lock</td>
</tr>
<tr>
<td>• Sleep</td>
</tr>
<tr>
<td>• Losing a notify</td>
</tr>
<tr>
<td>• Blocking critical section</td>
</tr>
<tr>
<td>• Orphaned thread</td>
</tr>
<tr>
<td>• Notify instead of notify all</td>
</tr>
<tr>
<td>• Interference</td>
</tr>
<tr>
<td>• Deadlock (deadly embrace)</td>
</tr>
</tbody>
</table>
Pairwise Construction

- Using our combinatorial model we get 44 task samples
- The task samples are selected out of over 14,000 different programs
Acquiring Benchmark Examples

Leveraging Existing Programs

- The IBM Concurrency Benchmark [EU04]
  - 28 Java concurrency programs
- The Rungta and Mercer Model Checking Benchmark [RM07]
  - Mix of Java programs from IBM and other sources
- BugBench [LLQ+05]
  - 4 C++ concurrency programs
- Open-source repositories

Acquiring Benchmark Examples

Using Program Mutation

• Mutation analysis uses a set of mutation operators in which each operator corresponds to a syntactic bug pattern.

• A mutation operator is applied to a program and generates a set of mutant programs

• To generate additional examples we plan to use the concurrency mutation tool - ConMAN [Con, BCD06]
  • We will apply ConMAN to existing programs


Performance Measures

- We need to assess the fitness of a given tool with respect to its ability to find bugs (effectiveness) and its efficiency with which the bug detection is carried out.
- Performance measures are necessary to achieve this!

How can we measure effectiveness?

How can we measure efficiency?
Performance Measures

Effectiveness

**bug detection rate of t =**
the percentage of bugs detected by a tool t.

**ease to kill a kind of bug by t =**
the percentage of bugs of a given kind that are detected by a tool t.
Performance Measures

Efficiency

cost (in time) to detect a bug by $t = \text{the total time to detect the bug by a tool } t$

path cost to detect a bug by $t = \text{the number of interleaving schedules analyzed/executed in order to find the bug by a tool } t$
Conclusions & Future Work

• We have proposed a new benchmark to assess the fitness of a concurrency bug detection tool and to compare it with other tools.
• We have also developed a new approach to benchmark construction based on combinatorial test design (CTD).

NEXT STEP:
Select 44 example programs/task samples for benchmark – but first get feedback on construction from community!
Using Combinatorial Benchmark Construction to Improve the Assessment of Concurrency Bug Detection Tools

Jeremy S. Bradbury\textsuperscript{a}, Itai Segall\textsuperscript{b}, Eitan Farchi\textsuperscript{b}, Kevin Jalbert\textsuperscript{a}, David Kelk\textsuperscript{a}

\textsuperscript{a}Software Quality Research Group
University of Ontario Institute of Technology
Oshawa, ON, Canada

\textsuperscript{b}IBM Haifa Research Laboratory
Haifa, Israel

jeremy.bradbury@uoit.ca, itais@il.ibm.com, farchi@il.ibm.com,
kevin.jalbert@uoit.ca, david.kelk@uoit.ca