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

Jeremy S. Bradbury[α], Itai Segall[β], Eitan Farchi[β], Kevin Jalbert[α], David Kelk[α]

[α]Software Quality Research Group
University of Ontario Institute of Technology
Oshawa, Ontario, Canada

[β]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

Funding partially provided by: NSERC CRSNG

Background

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

[EFN+02] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. IBM Systems Journal, 41(1):111{125, 2002.

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]

[HP04] D. Hovemeyer and W. Pugh. Finding bugs is easy. SIGPLAN Not., 39(12):92{106, 2004.

[Art01] C. Artho. Finding faults in multi-threaded programs. Master's thesis, Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin, 2001.

[NPSG09] M. Naik, C.-S. Park, K. Sen, and D. Gay. Eective static deadlock detection. In 2009 IEEE 31st Int.Conf. on Software Engineering (ICSE), pages 386-396, 2009.

[JSu] JSure for concurrency. Web page: http://www.surelogic.com/concurrency-tools.html

[Par] Parsoft JTest - Java testing, static analysis, code review. Web page: http://www.parasoft.com/jsp/products/jtest.jsp/

[LHDQ10] Z. Luo, L. Hillis, R. Das, and Y. Qi. Eective static analysis to find concurrency bugs in Java. In Proc. Of 10th IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM), 2010.

Background

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]

[HP00] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. Int. Journal on Software Tools for Technology Transfer (STTT), 2(4), 2000.

[RDH03] Robby, M. B. Dwyer, and J. Hatclliff. Bogor: an extensible and highly-modular software model checking framework. In Proc. of ESEC/FSE-11, 267-276, 2003.

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 depsite(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.depsite(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
```

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 depsite(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
```

```java
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.deposite(10);
  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
```

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 depsite(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.depsite(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

[SEH03] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: a challenge to software engineering. In Proc. ICSE 2003, pages 74-83, May 2003

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].

[DKL+99] S. R. Dalal, et al. Model-based testing in practice. In Proc. of ICSE '99, pages 285-294, 1999.
[TL02] K. C. Tai and Y. Lie. A test generation strategy for pairwise testing. IEEE Trans. Softw. Eng., 28(1):109{111, Jan. 2002.

Building a Combinatorial Model (2)

Parameter & Value Selection

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

Program Size

Parameter	Small	Medium	Large
Number of statements	< 10k	10k-100k	> 100k
Number of critical regions	< 5	5-20	>20
Percentage of statements in critical regions	< 5%	5-15%	> 15%

Building a Combinatorial Model (3)

Number of Threads

- Maximum number of threads executing in parallel during program execution

Small	Medium	Large	Very Large
< 5	5-10	10-100	> 100

Building a Combinatorial Model (4)

Path Error Density

- The probability of a thread interleaving manifesting a bug

Very Low	Low	Medium	High
< 2%	2-25%	25-75%	> 75%

Bug Depth

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

Low	Medium	High
< 25	25-50	> 50

Building a Combinatorial Model (5)

Bug Pattern Type

- The kind of bug exhibited by the program.

Bug Pattern Name
• Nonatomic operations assumed to be atomic • Two-state access • Wrong long or no lock • Double-checked lock • Sleep • Losing a notify • Blocking critical section • Orphaned thread • Notify instead of notify all • Interference • Deadlock (deadly embrace)

Pairwise Construction

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

	Program Size – # Statements	Program Size – # Critical Regions	Program Size – % Statements in Critical Regions	# Threads	Path Error Density	Bug Depth	Bug Pattern
1	Small	Small	Large	Small	Medium	High	TwoStageAccess
2	Medium	Small	Medium	Large	VeryLow	Medium	NonAtomicAssumedAtomic
3	Large	Medium	Small	Small	Low	Medium	BlockingCriticalSection
4	Medium	Large	Large	Medium	Low	Low	Interference
5	Small	Medium	Medium	VeryLarge	High	Low	OrphanedThread
6	Large	Large	Small	Large	High	High	NoLock
7	Medium	Large	Small	VeryLarge	Medium	Medium	LostNotify
8	Small	Medium	Small	Medium	VeryLow	High	NotifyInsteadOfNotifyAll
9	Large	Small	Medium	Medium	Medium	Low	SleepInsteadOfJoin
10	Large	Large	Large	VeryLarge	VeryLow	High	Deadlock
11	Medium	Small	Large	Medium	High	Medium	DoubleCheckedLocking
12	Small	Large	Medium	Small	VeryLow	Low	DoubleCheckedLocking
13	Small	Medium	Large	Large	Low	Medium	SleepInsteadOfJoin
14	Medium	Medium	Small	Large	Medium	Low	Deadlock
15	Large	Small	Medium	VeryLarge	Low	Medium	NotifyInsteadOfNotifyAll
16	Large	Small	Medium	Small	High	High	LostNotify
17	Medium	Small	Small	Small	Low	High	OrphanedThread
18	Small	Large	Medium	VeryLarge	VeryLow	Low	BlockingCriticalSection
19	Small	Medium	Large	VeryLarge	Low	Low	NonAtomicAssumedAtomic
20	Large	Medium	Medium	Small	VeryLow	Medium	Interference
21	Large	Large	Small	Small	Medium	High	NonAtomicAssumedAtomic
22	Small	Medium	Large	Large	VeryLow	Low	LostNotify
23	Medium	Small	Large	Medium	Medium	Medium	NoLock
24	Small	Medium	Medium	VeryLarge	Low	Low	NoLock
25	Small	Small	Medium	Small	Low	Medium	Deadlock
26	Small	Small	Small	Large	High	High	Interference
27	Large	Large	Small	Medium	High	Medium	TwoStageAccess
28	Medium	Medium	Medium	Large	VeryLow	Low	TwoStageAccess
29	Large	Large	Large	Large	Medium	Medium	OrphanedThread
30	Large	Medium	Small	Large	Medium	High	DoubleCheckedLocking
31	Medium	Large	Large	Small	High	Low	NotifyInsteadOfNotifyAll
32	Medium	Small	Large	Large	Medium	High	BlockingCriticalSection
33	Medium	Large	Small	VeryLarge	VeryLow	High	SleepInsteadOfJoin
34	Small	Large	Small	VeryLarge	Low	High	DoubleCheckedLocking
35	Small	Large	Small	Large	Medium	Low	NotifyInsteadOfNotifyAll
36	Small	Medium	Large	Small	High	Medium	SleepInsteadOfJoin
37	Medium	Medium	Medium	Small	VeryLow	High	NoLock
38	Small	Large	Small	VeryLarge	Low	Low	TwoStageAccess
39	Small	Large	Large	VeryLarge	Medium	High	Interference
40	Large	Medium	Large	Medium	VeryLow	High	OrphanedThread
41	Medium	Medium	Large	Medium	High	High	Deadlock
42	Small	Medium	Small	Medium	High	High	NonAtomicAssumedAtomic
43	Small	Small	Large	Medium	Low	Medium	LostNotify
44	Small	Medium	Medium	Medium	High	High	BlockingCriticalSection

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

[EU04] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In Proc. of PADTAD 2004.

[RM07] N. Rungta and E. G. Mercer. Understanding hardness in models used for benchmarking model checking techniques, Proc. of the 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), pages 247-256, 2007.

[LLQ+05] S. Lu, et al. Bugbench: Benchmarks for evaluating bug detection tools. In Proc. of the Workshop on the Evaluation of Software Defect Detection Tools, 2005.

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

[Con] ConMAn: Concurrency mutation analysis operators. Web page: https://github.com/sqrg-uoit/ConMAn

[BCD06] J. S. Bradbury, J. R. Cordy, and J. Dingel. Mutation operators for concurrent Java (J2SE 5.0). In Proc. of the 2nd Workshop on Mutation Analysis (Mutation 2006), pages 83-92, 2006.

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 =
the total time to detect the bug by a tool t

path cost to detect a bug by t =
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[α], Itai Segall[β], Eitan Farchi[β], Kevin Jalbert[α], David Kelk[α]

[α]Software Quality Research Group
University of Ontario Institute of Technology
Oshawa, ON, Canada

[β]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

Funding partially provided by: NSERC CRSNG