# 并发软件适应性随机测试方法①

岳 翰[1,2], 吴 鹏[1]

[1](中国科学院软件研究所 计算机科学国家重点实验室, 北京 100190)
[2](中国科学院大学, 北京 100190)

**摘 要**: 测试用例选择是软件测试中的关键问题之一. 目前, 测试用例选择在并发软件测试方面鲜有涉及. 以多线程并发程序为研究对象, 提出面向并发软件的适应性随机测试方法, 通过优化测试用例选择, 来提高并发软件测试的效率和错误发现能力. 根据实验结果, 我们提出的并发软件适应性随机测试方法比随机测试方法的测试效率更高, 错误发现能力也更强.

**关键词**: 并发测试; 适应性随机测试; 测试用例选择

## Adaptive Random Testing for Multi-Threaded Concurrent Programs

YUE Han[1,2], WU Peng[2]

[1](University of Chinese Academy of Sciences, Beijing 100190, China)
[2](State Key Laboratory of Computer Science, Institute of Software Chinese Academy of Sciences, Beijing 100190, China)

**Abstract**: Test case selection is an important part in software testing. However, there is little research about test case selection in testing of concurrent programs. We propose an adaptive random testing approach for multi-threaded concurrent programs. By selecting test cases with heuristics, experimental results show that our approach can improve the efficiency and fault detection capability of random testing of concurrent programs.

**Key words**: concurrent testing; adaptive random testing; test case selection

## 1 引言

随着多核处理器体系结构和分布式系统的发展, 并发软件应用逐渐广泛. 但并发软件由于其内在的不确定性, 如并发线程间的交互等, 导致其状态空间“组合爆炸”、运行结果不确定且不可预知. 这使并发错误非常难于发现, 而且更难于重现, 从而对软件自动化测试提出了新的挑战.

目前, 并发软件测试方法往往只针对给定的多线程并发程序和输入数据进行测试, 而不关心测试输入的选择问题, 即测试输入仍由用户给定. 这主要是基于并发错误的“数据无关性假设”, 即线程调度与输入数据无关. 但事实上, 并发软件的数据相关性是不可忽视的. 例如, 并发程序中的循环次数往往与输入数据有关. 如何在资源有限的条件下选择更“好”的测试用例是并发软件测试亟待解决的关键问题. 目前, 测试用例选择在并发软件测试方面还鲜有涉及. 我们以多线程并发程序为研究对象, 提出面向并发软件的适应性随机测试方法.

面向顺序程序的适应性随机测试方法[1]是指在随机测试中, 选择多样化的测试用例, 使测试输入能够均匀地覆盖输入数据域. 适应性随机测试方法在发现一个软件错误所需执行测试输入的平均数量上比随机测试方法本身减少多达 50%[2]. 这是因为软件输入域上可能导致软件故障的区域(即“故障区域”)往往是毗连的. 对顺序程序而言, 测试用例多样性度量指标可以直观地定义在输入数据域上.

但并发软件(如多线程程序)测试主要关注并发程序各个线程之间的交互行为. 因此, 并发测试用例的多样性度量指标应着眼于对并发线程调度的均匀覆盖. 我们以线程间可能的交错模式(interleaving idiom) 和交

① 基金项目:国家自然科学基金(61100069,61161130530,61472405)
收稿时间:2015-03-06;收到修改稿时间:2015-04-17

错实例(iRoot)[3]为基准, 提出并发测试用例的多样性度量指标, 即通过比较不同线程调度所能覆盖的线程交错实例来衡量不同并发测试用例之间的差异(或称之为“距离”); 并在此基础上, 提出两种并发测试用例的最优选择策略: 即以所覆盖且仍未执行到的线程交错实例最多者为最优, 或者在对所有已知仍未执行到的线程交错实例构成最小覆盖的测试用例中选择覆盖最多者为最优.

实验结果表明, 我们提出的测试方法相对于随机测试在效率上有一定幅度的提升. 在所需执行的测试用例个数方面, 我们的测试方法相比随机测试方法有显著的进步. 而且在测试时间方面, 我们提出的测试方法也优于随机测试方法.

本文第 2 节介绍相关工作; 第 3 节阐述我们提出的并发程序适应性随机测试方法; 第 4 节介绍工具实现, 并通过实例研究, 与随机测试方法进行对比分析; 最后一节对全文进行总结.

## 2 相关工作

### 2.1 适应性随机测试

通常, 软件测试的过程就是从所有可能的输入(即软件输入域)中取出若干样本, 分别执行被测程序, 并判断其运行结果是否满足软件规范. 若不满足, 则称为软件故障, 即表明被测程序中存在错误. 经验表明, 软件输入域上可能导致软件故障的区域(即“故障区域”)往往是毗连的.

基于这一经验规律, Chen TY 等人[1]提出了适应性随机测试方法(Adaptive Random Testing, 简称 ART), 以此来提高随机测试的效率. 该方法是指在随机测试中, 选择多样化的测试用例, 以使得测试输入尽量均匀地覆盖输入域. 从较早提出的 Fixed-Sized-Candidate-Set-ART[1]算法(简称 FSCS-ART 算法), 到之后的 Restricted Random Testing[4]、Quasi-Random Testing[5]和 Lattice-Based ART[6]算法, 都对随机测试的效率有较大幅度的提升. Chen TY 等人[7]证明, 就数值程序而言, 如果不事先知道故障区域, 适应性随机测试算法可将随机测试发现第一个错误的时间平均至多缩短 50%.

以FSCS-ART算法为例, 如图1所示. 该算法每次在候选随机测试用例中, 选择与已执行测试用例的“距离”最远的一个测试用例进行测试.

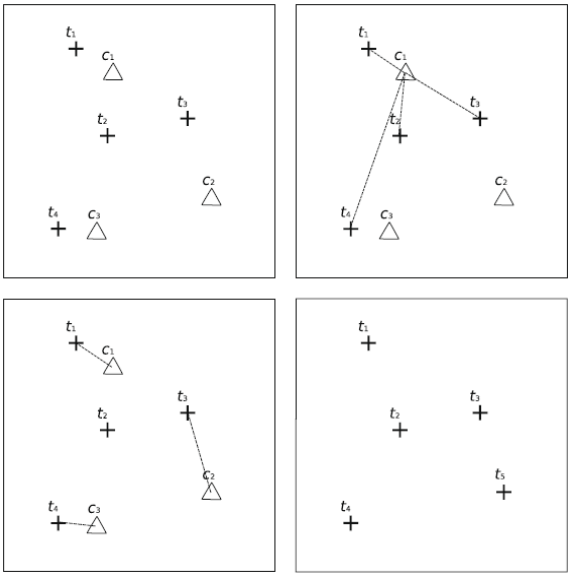

图 1　FSCS-ART 算法[1]

### 2.2 并发测试

测试多线程程序并能够发现并发错误是一项艰巨的任务, 因为线程间可能的交错行为随线程数量及大小而发生“组合爆炸”. 而且, 在并发错误与输入域的关系方面, 还缺乏必要的研究.

并发测试中较为常见的方法是压力测试(stress testing), 即反复执行同一组测试用例. 这种方法往往会执行很多无谓的测试, 并难以覆盖各种线程交错执行的可能性.

另一种常见的并发测试方法是系统性测试(systematic testing), 即针对给定的多线程程序和输入, 通过控制线程调度, 遍历所有可能的线程交错执行序列. 虽然系统性测试不会漏掉并发错误, 但并不适合运行时间较长的并发程序. 为了改进效率, 可以使用偏序归约[8]来减少冗余的线程交错执行序列, 或直接限制线程上下文的切换次数[9,10].

本文采用的并发测试方法是主动测试[11-13](active testing). 这类方法针对给定的多线程程序和输入, 先预测可能发生并发错误的线程交错执行序列, 再通过控制线程调度对预测到的线程交错执行序列进行测试. 这种方法相对随机测试和系统性测试而言耗时少, 但可能会漏掉一些并发错误.

Maple[3]是覆盖度驱动的并发主动测试工具. Maple 所提出的覆盖度准则以线程间可能的交错模式(interleaving idiom)为基础. 线程交错模式表示线程间

可能的交互关系. 针对双线程并发程序即可发生的并发错误, Jie Yu 等人提出了 6 种线程交错模式, 如图 2 所示. 每一种线程交错模式在并发程序中的具体实例称为线程交错实例(iRoot). 一个测试用例的线程交错实例覆盖度, 就是该测试用例在运行时可能发生的线程交错实例数量. 我们的研究以 Maple 为工具来预测测试用例的覆盖度, 并执行并发主动测试.

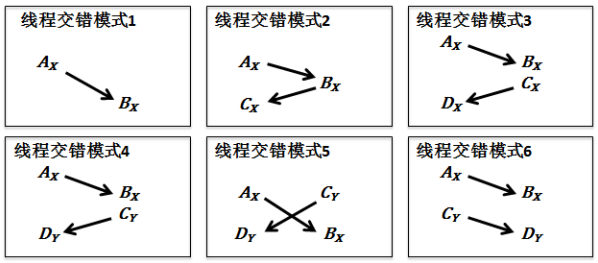


图 2 线程交错模式

### 2.3 并发测试的输入集

过去的几年里, 针对给定输入的并发错误检测与验证研究已经取得了重大的进步, 但一个程序的输入集往往很大. 针对给定一个输入集的并发错误检测, Deng DD 等人[14]提出了新的覆盖度准则 CFP(Concurrent Function Pairs)来估算不同输入的线程交错序列的重叠, 以减少并发错误检测的冗余. 他们提出方法的粒度在函数上, 而我们提出方法的粒度在指令上.

## 3 并发程序适应性随机测试方法

令|S|表示集合 S 的元素个数, *iRoot*(*t*)表示测试用例 *t* 可能覆盖的线程交错实例集, *d*(*t*,*T*)表示测试用例 *t* 与已执行测试用例集 *T* 之间的距离.

这里 *d*(*t*,*T*)是根据测试用例之间的差异来度量的. 测试用例之间的差异越大, 距离也就越远. 为了衡量测试用例之间的差异, 我们需要先通过 Maple 工具来预测每个测试用例的覆盖度, 并计算测试用例之间的差异. 之后再选择测试用例执行并发主动测试.

以下给出两种最优测试用例的选择策略.

### 3.1 最优测试用例选择策略 1

在第一种策略中, 候选测试用例 $t\in C$ 与已执行测试用例集的距离 $d_1(t,T)$用候选测试用例 *t* 所覆盖但未被执行到的线程交错实例个数来计算, 即

$$d_1(t,T)=|\ iRoots(t)\setminus AR(T)|,\ t\in C$$

其中, *AR*(*T*)表示测试用例集 *T* 已执行到的线程交错实例集, 且:

$$AR(T)\subseteq\cup_{t\in T}iRoots(t)$$

在每次选择时, 因为 *T* 和 *AR*(*T*)发生变化, 每个在候选测试用例集 *C* 中的测试用例都需要重新计算距离. 这种选择策略的整体测试流程如下:

1) 初始化已执行测试用例集 *T* 为空集; 初始化 *T* 已执行到的线程交错实例集 *AR*(*T*)为空集;

2) 随机生成候选测试用例集 *C*;

3) 计算 *C* 中每个候选测试用例 $t\in C$ 与已执行测试用例集 *T* 之间的距离 $d_1(t,T)$;

4) 设 $t^*\in C$ 为最优测试用例, 即 $t^*$满足对任意 $t\in C$, $d_1(t,T)\le d_1(t^*,T)$. 若 $d_1(t^*,T)=0$, 即表明已执行所有可能的线程交错实例, 测试结束; 否则执行测试用例 $t^*$, 并在主动测试 $t^*$后, 捕捉记录这次测试已执行到的线程交错实例集 $N^*$;

5) 把 $t^*$加入到 *T* 中, 并将其从 *C* 中移除; 将 $N^*$并入 *AR*(*T*)中;

6) 随机生成一个新的候选测试用例 *t*, 将其加入 *C* 中;

7) 返回步骤 3), 继续测试用例选择和执行.

### 3.2 最优测试用例选择策略 2

定义. 如果集合 $M\subseteq C$ 满足以下两个条件, 则称 *M* 是 *NR*(*C*)的最小覆盖:

1) $NR(C)\subseteq\cup_{t\in M}iRoots(t)$;

2) 对任意满足条件 1)的 *C* 的子集 *S*, $|M|\le|S|$.

其中, *NR*(*C*)表示候选测试用例集 *C* 所覆盖的但仍未被执行到的交错序列实例集, 即

$$NR(C)=(\cup_{t\in C}iRoots(t))\setminus AR(T)$$

在第二种策略中, 要计算出对所有已知仍未执行到的线程交错实例构成最小覆盖的测试用例集 *M*. 候选测试用例 $t\in M$ 与已执行测试用例集的距离 $d_2(t,T)$用 *t* 所覆盖但未被执行到的线程交错实例个数来计算, 即

$$d_2(t,T)=|\ iRoots(t)\setminus AR(T)|,\ t\in M$$

这种选择策略的整体测试流程如下:

1) 初始化已执行测试用例集 *T* 为空集; 初始化 *T* 已执行到的线程交错实例集 *AR*(*T*)为空集; 初始化 *M* 为空集;

2) 随机生成候选测试用例集 *C*;

3) 计算出对 *NR*(*C*)构成最小覆盖的测试用例集 *M*(算法如图 3 所示);

4) 计算 $M$ 中每个候选测试用例 $t \in M$ 与已执行测试用例集 $T$ 之间的距离 $d_2(t,T)$;

5) 设 $t^* \in M$ 为最优测试用例, 即 $t^*$满足对任意 $t \in M$, $d_2(t,T) \le d_2(t^*,T)$. 若 $d_2(t^*,T)=0$, 即表明已执行所有可能的线程交错实例, 测试结束; 否则执行测试用例 $t^*$, 并在主动测试 $t^*$后, 捕捉记录这次测试已执行到的线程交错实例集 $N^*$;

6) 把 $t^*$加入到 $T$ 中, 并将其从 $C$ 和 $M$ 中移除; 将 $N^*$并入 $AR(T)$中;

7) 随机生成一个新的候选测试用例 $t$, 将其加入 $C$ 及 $M$ 中;

8) 返回步骤 3), 继续测试用例选择和执行.

第二种策略的整体测试流程与第一种策略基本相同, 但要在选择的最优测试用例被执行后将其从 $M$ 中移除, 同时将新生成的测试用例加入 $M$(步骤 6、7). 因为新的候选测试用例集 $C$ 所覆盖但仍未被执行到的线程交错实例集 $NR(C)$, 一定会被旧的 $NR(C)$与新测试用例所覆盖线程交错实例集 $iRoot(t_{new})$的并集所覆盖.

计算 $M$ 时, 忽略掉元素个数不少于当前的$|M|$的测试用例组合情况, 会大幅减少算法的耗时(行 06).

第二种策略相比第一种策略在测试用例选择阶段会花费较多的时间, 但有可能会得到更优的测试用例选择策略, 以进一步避免执行不必要的测试用例. 而且考虑到测试程序的复杂程度, 第二种策略的测试用例选择花费的时间往往远少于一个测试用例的执行时间.

### 3.3 优化

在选定测试用例 t 进行检测时, Maple 总是检测 $iRoot(t)$中所包含的线程交错实例. 在我们提出的方法中, 可以只检测 $N(t)$中所包含的线程交错实例, 从而可以提高并发测试的效率. 这是因为 $iRoot(t) \backslash N(t)$这部分线程交错实例在之前已经被检测过了. 其中 $N(t)$表示候选测试用例 $t$ 所覆盖的但仍未被执行到的交错序列实例集, 即

$$N(t) = iRoots(t) \setminus AR(T)$$

## 4 实验结果及分析

我们采用线程交错模式这一覆盖度准则设计适应性随机测试方法, 并在 Maple 基础上开发了实验工具.

每一个候选测试用例的线程交错实例预测阶段, 在运行该测试用例时监测不同线程对同一内存位置的访问. 当两条指令(不妨设为 E0, E1)对同一内存位置进行了访问, 且它们属于不同线程时, 这两条指令构成了两个线程交错模式 1 的线程交错实例(E0->E1, E1->E0). 当完成针对线程交错模式 1 的预测后, 通过分析这些线程交错实例, 再组合出复杂线程交错模式的线程交错实例, 以此得出最终可能引发故障的线程交错实例集.

当所有候选测试用例都完成预测阶段后, 通过对比各候选测试用例所覆盖的线程交错实例集, 选取出最优的候选测试用例并执行.

针对线程交错模式 1, 我们进行了一系列的测试. 一次完整测试, 指的是完成了完整的测试流程的测试.

```
00  M={}; min=MAX_INT;  /*全局变量, min 表示 M 集非空时的元素个数*/
01  def calculateM(C, Chosen, NotTestediRoots):        /*递归计算 M, 初始输入为( C, {}, NR(C) )*/
02      if | NotTestediRoots |==0:
03          if |Chosen|<min:
04              min=|Chosen|;
05              M=Chosen;
06      elif |Chosen|<(min-1):
07          for t in C:
08              NotTestediRoots2 = NotTestediRoots - iRoot(t);
09              if | NotTestediRoots2 |<| NotTestediRoots |:
10                  C2={}∪C;
11                  C2.remove(t);
12                  Chosen2={}∪Chosen;
13                  Chosen2.add(t);
14              calculateM (C2, Chosen2, NotTestediRoots2);
```

图 3　计算对 NR(C)构成最小覆盖的测试用例集 $M$ 的算法

表 1 各方法未执行优化的情况下平均每次完整测试

| 测试程序 | 线程交错实例个数 | ART1 | | | | Random | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 时间(秒) | 执行测试用例个数 | | | 时间(秒) | 执行测试用例个数 | | |
| | | | 最小值 | 平均值 | 最大值 | | 最小值 | 平均值 | 最大值 |
| P2_1v1 | 34 | 47.084 | 2 | 2.209 | 3 | 81.653 | 6 | 6.624 | 7 |
| P2_2v3 | 34 | 47.047 | 2 | 2.212 | 3 | 80.803 | 6 | 6.569 | 7 |
| P2_1v3 | 34 | 46.361 | 2 | 2.159 | 3 | 78.781 | 6 | 6.427 | 7 |
| P5_1v1 | 89 | 105.927 | 2 | 2.288 | 3 | 250.616 | 6 | 6.828 | 7 |

表 2 各方法在执行优化的情况下平均每次完整测试

| 测试程序 | ART1(opt) | | | | ART2(opt) | | | | Random(opt) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 时间(秒) | 执行测试用例个数 | | | 时间(秒) | 执行测试用例个数 | | | 时间(秒) | 执行测试用例个数 | | |
| | | 最小值 | 平均值 | 最大值 | | 最小值 | 平均值 | 最大值 | | 最小值 | 平均值 | 最大值 |
| P2_1v1 | 43.402 | 2 | 2.172 | 3 | 43.062 | 2 | 2.098 | 3 | 47.221 | 6 | 6.614 | 7 |
| P2_2v3 | 43.399 | 2 | 2.197 | 3 | 42.974 | 2 | 2.105 | 3 | 47.145 | 6 | 6.623 | 7 |
| P2_1v3 | 42.103 | 2 | 2.149 | 3 | 41.933 | 2 | 2.081 | 3 | 45.009 | 6 | 6.410 | 7 |
| P5_1v1 | 91.954 | 2 | 2.246 | 3 | 92.107 | 2 | 2.227 | 3 | 99.184 | 6 | 6.793 | 7 |

表 3 平均每次发现第一个并发错误的测试对比

| 测试程序 | ART1 | | Random | | ART1(opt) | | Random(opt) | |
|---|---|---|---|---|---|---|---|---|
| | 时间(秒) | 执行测试用例个数 | 时间(秒) | 执行测试用例个数 | 时间(秒) | 执行测试用例个数 | 时间(秒) | 执行测试用例个数 |
| P3_1v1 | 32.76 | 1.401 | 46.49 | 3.353 | 30.11 | 1.394 | 29.82 | 3.296 |
| P4_1v1 | 48.71 | 1.519 | 83.43 | 3.462 | 44.19 | 1.512 | 49.49 | 3.441 |

表 1 是 ART1 方法与 Random 方法未执行优化的情况下 1000 次完整测试得出的实验结果对比. 其中 ART1 表示使用最优测试用例选择策略 1 的方法, Random 表示随机测试方法(即随机选择测试用例). 表 2 是 ART1 方法、ART2 方法与 Random 方法在执行优化的情况下 1000 次完整测试得出的实验结果对比, 线程交错实例个数同表 1. 其中 ART2 表示使用最优测试用例选择策略 2 的方法. 在这些实验中, 每次完整测试随机生成的候选用例个数最少为 6 个, 最多为 7 个.

表 3 是表示 ART1 方法与 Random 方法在发现第一个并发错误的测试中消耗的时间与执行测试用例的个数的对比.

测试程序 P*a_bvc*(a=2,3,5,4)接收的输入为两个整数 *x*、*y*, 根据 *x* 与 *y* 的值(是否为正)来决定程序产生的子线程的执行内容, 生成的随机数为正的概率与非正的概率的比值为 *b*:*c*. P2 与 P5 的并发错误可能因输入的不同而不同, 所以对它们进行完整测试. P3 与 P4 只有在输入满足一定条件的情况下才有可能产生并发错误, 因此对它们进行发现第一个并发错误所需时间的测试. 测试程序 P2 的关键代码如图 4 所示, 可以看出, 根据测试用例的不同, 程序可能会对不同的共享变量存在数据争用. 限于篇幅, 这里不再列出其他测试程序的代码.

```
Thread1:                    Thread2:
{                           {
    v1++;                   if(x>0)
    v2++;                       v1++;
    v3++;                   else
    v4++;                       v2++;
}                               if(y>0)
                                    v3++;
                                else
                                    v4++;
                            }
```

图 4 测试程序 P2 的关键代码

由表 1 可以看出, 在所进行过的实验中, 我们提出的两种最优测试用例选择策略相对于随机测试方法在需要执行的测试用例的个数方面有显著的进步(减

少了 66%左右), 且第二种策略更优. 而且在消耗的时间方面, 我们提出的两种最优测试用例选择策略相比随机测试方法都有一定幅度的减少(有检测优化时 8%左右, 无检测优化时可达 40%甚至更多).

根据表 3 的实验, 在发现第一个并发错误所需的时间方面, 当无检测优化时, 我们提出的策略比随机方法更快(时间缩短了 30%以上). 但有检测优化时, 我们提出的策略在测试程序非常简单时可能反而比随机测试方法略慢一点. 这是因为在初始阶段, 我们的策略需要对多个候选测试用例提前进行预测, 之后仅对新增候选测试用例进行预测; 而随机测试方法每次只对单个测试用例进行预测. 在发现第一个并发错误所需执行的测试用例个数方面, 无论是否执行检测优化, 我们提出的策略比随机方法依旧有显著的提升(减少了 50%至 60%).

## 5 总结

本文通过研究适应性随机测试在并发程序上的应用, 提出了两种合适的适应性随机测试方法. 我们通过对比线程交错实例来进行适应性随机测试中距离的计算, 这就使测试程序不再仅限于数值程序. 相对于随机测试方法, 我们提出的方法极大地减少了测试所需的测试用例个数, 并在一定程度上缩短了完整测试所需的时间, 而且没有引入额外的显著开销. 但因为在初始阶段需要对多个候选测试用例提前进行预测, 故有可能会推迟发现第一个错误.

## 参考文献


1 Chen TY, Leung H, Mak IK. Adaptive random testing. Lecture Notes in Computer Science, 2004, 3321. 320–329.

2 Chen TY, Kuo FC, Merkel RG, Tse TH. Adaptive random testing: the ART of test case diversity. Journal of Systems and Software, 2009, 83(1): 60–66.

3 Yu J, Narayanasamy S, Pereira C, Pokam G. Maple: a coverage-driven testing tool for multithreaded programs. Acm Sigplan Notices, 2012, 47(10): 485–502.

4 Chan KP, Chen TY, Towey DP. Restricted random testing: adaptive random testing by exclusion. International Journal of Software Engineering and Knowledge Engineering, 2006, 16(4): 553–584.

5 Chen TY, Merkel RG. Quasi-random testing. IEEE Trans. on Reliability, 2007, 56(3): 562–568.

6 Mayer J. Lattice-based adaptive random testing. Automated Software Engineering, 2005: 333–336.

7 Chen TY, Merkel RG. An upper bound on software testing effectiveness. ACM Trans. on Software Engineering and Methodology, 2008, 17(3): 372–382.

8 Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software. Acm Sigplan Notices, 2005, 40(1): 110–121.

9 Musuvathi M, Qadeer S. Iterative context bounding for systematic testing of multithreaded programs. ACM Sigplan Notices, 2007, 42(6): 446–455.

10 Musuvathi M, Qadeer S, Ball T, Basler G, Nainar PA, Neamtiu I. Finding and reproducing heisenbugs in concurrent programs. OSDI, 2008, 8. 267–280.

11 Park S, Lu S, Zhou Y. Ctrigger: exposing atomicity violation bugs from their hiding places. ASPLOS, 2009, 37(1): 25–36.

12 Sen K. Race directed random testing of concurrent programs. PLDI, 2008, 43(6): 11–21.

13 Zhang W, Sun C, Lu S. Conmem: detecting severe concurrency bugs through an effect-oriented approach. Acm Sigplan Notices, 2010, 45(3): 179–192.

14 Deng DD, Zhang W, Lu S. Efficient concurrency-bug detection across inputs. Acm Sigplan Notices, 2013, 48(10): 785–802.