Concurrency Bugs 检测技术研究

刘宁 李菁菁 夏亦谦 高策 张坚鑫

摘 要 在多核时代，如何在多线程软件发布之前对其有效地测试并检测到 concurrency bug 很重要。Concurrency bug 有很多种，例如数据竞争和原子性冲突。但 Concurrency bug 是最难检测和调试的一类 Bug。当前对 concurrency bug 的检测和验证已经有不少成果。本文对几个不同的 concurrency bug 检测技术进行了描述和比较。

关键词 并发，Bug 检测，MUVI，ConSeq

Researches on Concurrent Bug Detection

LIU Ning LI Jing-jing XIA Yi-qian GAO Ce ZHANG Jian-xin

Abstract In the multi-core era, it is critical to efficiently test multi-threaded software and expose concurrency bugs before software release. There are many types of Concurrency bugs, such as data race and atomicity violation. But concurrency bugs are among the most difficult to detect and diagnose of all software bugs. Previous work has made significant progress in detection and validation. In this paper, we describe and compare several different technologies to detect concurrency bugs.

Keywords Cocurrency, Bug detection, MUVI, ConSeq

1. 引言

随着硬件技术的发展，计算机步入了多核时代，软件并发执行以更充分地利用资源，提高运行效率。

但是并发执行的不确定性可能造成一些并发 Bug。例如即使有相同的输入，如果线程没有适当地同步，会导致输出结果不同。还有不同线程对共享内存的交叉存取导致内存状态错误。

因此产生了基于对程序的静态分析和动态分析的并发 Bug 检测技术，。

本文总结了近年来学术界为检测并发 Bug 的尝试，包括（）。本文列举了这些技术对应的系统，分析比较了不同系统的特点与缺陷。

2. 并发 Bug

2.1. 并发 Bug 分类

常见的并发 Bug 包括以下几种类型。

2.1.1. 数据竞争(Data race)

数据竞争是指冲突的内存访问，即多个线程同时访问相同的地址，且至少有一个为写操作。如果没有同步机制，内存访问可能存在任意地顺序导致内存状态不同。

2.1.2. 违反原子性操作(Atomicity violation)

如图 2-1 所示，当 thd->proc_info 没有被同步机制保护时，可能在 1.1 和 1.2 之间，2.1 将它的值设置为 NULL，从而导致错误。所以需要对 1.1-1.2 以及 2.1 用临界区保护起来，保证原子执行。

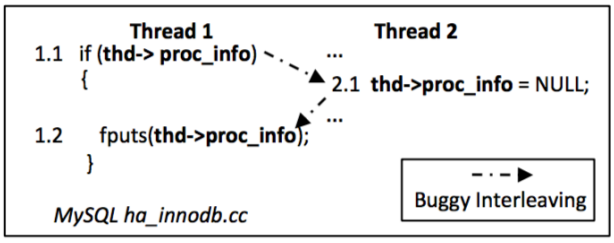

图 2-1 MySQL 中的 Atomicity violation bug

2.2. 并发 Bug 特征

对现实世界 conurrency bug 特征的研究发现：

- 非死锁 concurrency bugs 里的三分之一 bugs 是由于程序员所安排的时序被破坏而引发的，这些 concurrency bugs 不能通过简单的同步语义，例如锁和事务表达出来。
- 被检测的非死锁 concurrency bugs 里的

34%的 bugs，均涉及到不能被已有 bug detection 工具发现的多变量问题。MUVI 重点解决了多变量导致的 concurrency bug[5]。

- 被检测的 concurrency bugs 里的大约 92%的 bugs，都可以以不多于 4 次的内存读写在确定的顺序下触发。在 ConSeq 里面也有用到这个特征[6]。
- 被检测的非死锁 concurrency bugs 里的 73%的 bugs,都不能通过简单地添加或者改变锁来进行修复，而且许多修复第一次都是失败的。

3. 并发 BUG 检测技术

3.1. MUVI

之前已有不少检测并发错误的方案(加上引用)，但它们都无法处理多变量的问题。这是因为它们没有检测变量间的一致性关系。MUVI(全称是什么)通过代码分析和数据挖掘技术检测多个变量间的相关关系，并检测其在交互过程中产生的并发错误。多变量主要因为两个原因引起并发错误：

- 相关变量没有被同时更新(inconsistent updates)
- 相关变量在存取操作时，没有被同时保护(multi-variable concurrency)

3.1.1. MUVI 实现

3.2. ConSeq

ConSeq 取自 Consequence，是一个面向结果的反向分析框架。该系统分析 Bug 的生命周期三阶段（分别为产生，传播和造成系统崩溃），利用并发 Bug 的特性，反向找到可能存在的并发 Bug。

3.2.1. Bug 生命周期

如图 3.2-1 所示，Bug 的生命周期包含三个过程：

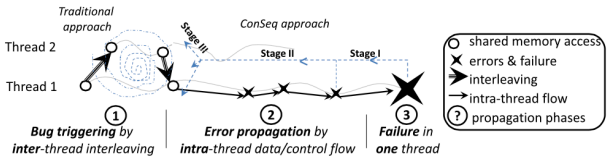

图 3.2-1 Bug 生命周期

- 触发：不同线程交叉访问共享内存，在某种执行顺序下产生错误数据；
- 传播：错误数据被读指令获取，程序执行时继续传播；
- 失败：某个操作使用错误数据，导致可见错误，例如程序崩溃、陷入死循环或产生错误输出。

例如在图 3.2-2 中，S3 执行后 S1 执行，导致内存中被写入错误的 runningURL 值，这是触发阶段；然后 S4 中读入了 runningURL 错误数据，这是传播阶段；最后不满足断言，这是失败阶段。

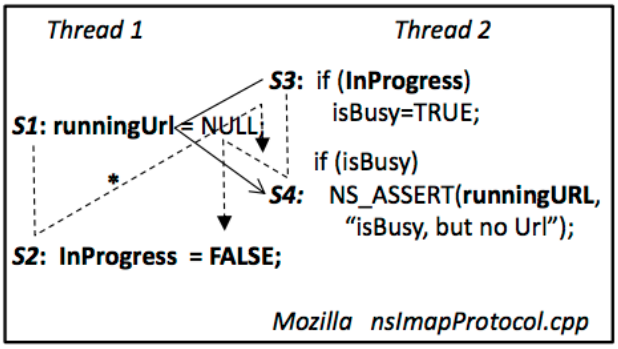

图 3.2-2 Bug 传播过程

3.2.2. 并发 Bug 的特性

通过对已有的 70 个非死锁并发错误的分析，发现以下 3 个并发 Bug 的特性：

- 并发错误通常发生在一个线程里。即错误的产生是由于多线程，但结果通常仅有一个线程产生了可见错误。说明可以将分析阶段分为并发分析和顺序分析两个阶段。
- 并发 Bug 的 Failure pattern 和顺序 Bug 相似。说明可以根据顺序 Bug 的 failure pattern 找到可能由并发 Bug 导致 failure 的地方。
- 传播路径通常很短。即读入错误数据之后，很快会产生 Failure。说明很容易找到产生 Bug 的代码。

3.2.3. ConSeq 实现

ConSeq 结合静态分析与动态分析技术，使用图 3.2-3 所示模型，沿着 bug 传播链反向分析可能存在的 Bug。

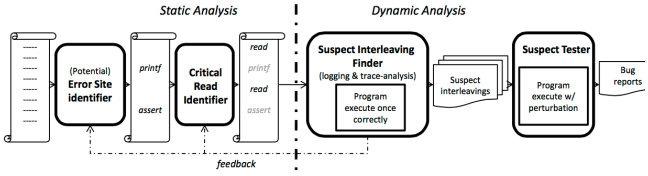

图 3.2-3 ConSeq 结构

Error-site identifier，通过对 binary 的静态分析，找出五类可能的 failure 的位置，通过一些特征值来发现，比如 assert，printf，fprintf，stderr 等，还有一些工具，SCC，Daikon（用于发现 implicit errors）；如图 1 的代码示例，通过静态分析，可以找出 S4 是可能的 failure 的位置。

Critical-read identifier，用静态分析找到影响 error site 的 critical-read（short propagation distance）。Critical-read 指的就是导致程序失败的读操作。因为本文指出，能够导致程序崩溃的原因一定是由读操作引起的。如图 1 的代码示例，在这个代码片段中，虚线代表不会产生 bug 的执行路径，如 S1->S2->S3->S4 或者 S3->S4->S1->S2；实线代表会产生 bug 的执行路径：S3->S1->S4。在这段代码中，Critical-read 就是 S4，即在进行断言操作时读到了个空值，导致程序中断。

Suspicious-interleaving finder，动态分析，运行一次程序，得到运行路径以及 control/data 依赖图，通过 trace 和依赖图来找到可能的关联变量；再对它们进行分析，排除掉不可能的，剩下的就是可能性较大的 bug。当要判断 R 与 W'是否有关联（数据依赖）时，它们必须要满足三个条件，（一）它们必须都访问相同内存；（二）W'必须在 R 之前；（三）被 W'写入内存的值在被 R 读取之前不能被重写。图 3 就展示了三种不满足（三）的情况：在 W'操作之后，总会有一次 W 操作先于 R 被执行，其中 W、W'和 R 访问的都是同一块内存。图 3(a)是线程内的一种执行顺序，W 在 W'和 R 之间被执行；图 3(b)是由于一些栅栏同步机制强制规定了 W'必须在 W 之前被执行，同时 W 必须在 R 之前被执行；图 3(c)是互斥现象，如 W'和 W 与 R 互斥，或者 W'与 W 和 R 互斥。只要满足这三种情况的一种，R 原本该读到的由 W'写入的值就会被 W 覆盖掉，从而判断出关联变量为 W。以上是 Suspicious-interleaving identification 的核心分析方法。在图 3.2-4 的代码示例中，可以分析出来，可能导致程序中断的共享内存访问操作是 S1。

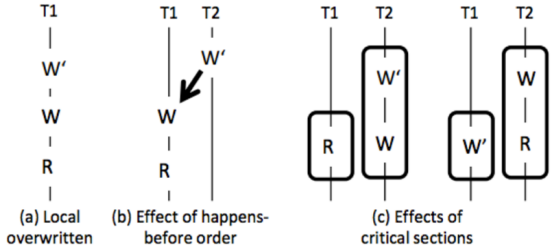

图 3.2-4 W'总会先于 R 的执行顺序

Suspicious-interleaving tester，动态分析，找到那些真正会导致的 failure 的 bug。在这最后一步中，ConSeq 会在 critical read 指令之前插入一些条件延迟（time-out）来改变程序的执行顺序。在图 1 的示例代码中，程序本该以 S3->S4->S1->S2 的顺序运行，ConSeq 在 S4 之前插入延迟，使得程序以 S3->S1->S4 的顺序执行，然后产生中断。这样就成功验证了这段代码是存在 Bug 的。

3.2.4. 缺点

由于 ConSeq 利用了并发 Bug 传播路径较短的特点，无法找到有较长传播路径的 Bug。

3.3. CFP

根据每个输入所覆盖的并行函数对(CFP)选择输入集合，从而减少对同一 Bug 的重复检测。

3.3.1. Bug 检测存在冗余

现有的技术通常包括以下几个步骤：（1）设计 Input, 产生测试输入来保证代码的覆盖率（2）检测 Bug， 使用动态 Bug 检

测工具找到可能的 Interleaving（3）验证 Bug，对每个输入执行多次，来排除掉一些疑似的 Bug。

针对上述步骤，已有很多研究对此进行改进和优化。对于第一步，无论是对但进行还是多进程程序，很多技术已经能够自动生成高代码覆盖率的输入。对于第二步，很多工具已经能够检测出不少类型的 interleaving bug，例如 data race, atomicity-violation 等。对于第三步，也有各种 schemes 能够高效的验证 suspicious buggy interleaving。

然而这些方法都没有改变第二步和第三步的基础实现，这导致原本就存在的性能损耗没有得到解决，通常会带来 10 倍到 100 倍的性能下降，因此需要新的方法来解决性能问题。

主要原因是输入之间存在大量的重合，即同一个 bug 可以被多个输入检测到。如表 3.3-1 所示，每个检测出的 data race 或 atomicity violation 都平均被 2.7-4.5 个输入检测到。真实 bug 的 report 也是相似的，对 data race 平均有 4-7 个输入检测到，对 atomicity violation 平均有 3.5-7 个输入检测到。

表 3.3-1 检测重复率

	# All Race Pairs			# All Atom. Vio.			# Buggy Race Pairs			# Buggy Atom. Vio.		
App.	Total	Unique	**DupRate**	Total	Unique	**DupRate**	Total	Unique	**DupRate**	Total	Unique	**DupRa**
Click	3114	848	**3.6**	6145	2298	**2.7**	6	1	**6.0**	6	1	**6**
FFT	300	66	**4.5**	1423	369	**3.8**	28	4	**7.0**	35	5	**7**
LU	238	58	**4.1**	874	163	**5.4**	28	4	**7.0**	21	3	**7**
Mozilla	1991	481	**4.1**	2459	723	**3.4**	42	6	**7.0**	42	7	**6**
PBZIP2	293	65	**4.5**	499	143	**3.5**	32	8	**4.0**	39	11	**3**

3.3.2. CFP

CFP 是 Concurrent function pair 的缩写，是能够并行执行的函数对。如图 3.3-1 所示，由于锁的存在，foo1()和 bar()可以并行执行，而 foo2()和 bar()不可以。

```
/*Thread 1*/
foo1(){
  lock(L);
  foo2();
  unlock(L);
  ...
  ...
}
```

```
/*Thread 2*/
lock(L);
bar();
unlock(L);
```

图 3.3-1 并行函数对

可以使用同步信息来判断函数是否能够并行执行。如图 3.3-2 伪代码所示。

```
/* .ent, .exi: function entrance, exit;
 .ent_exi.lockset: lockset protecting the critical
     section that holds both entrance and exit;
 .vec_time: vector timestamps calculated using
     order-enforcing synchronization. */
Bool concurrentFunction(f1, f2)
{
  if(f1.thread_id == f2.thread_id) return false;

  /*Can f1 start between f2's entrance and exit?*/
  if ((f1.ent.lockset ∩ f2.ent_exi.lockset) != ∅) return false;
  if (f1.ent.vec_time < f2.ent.vec_time) return false;
  if (f1.ent.vec_time > f2.exi.vec_time) return false;

  /*Can f2 start between f1's entrance and exit?*/
  if ((f2.ent.lockset ∩ f1.ent_exi.lockset) != ∅) return false;
  if (f2.ent.vec_time < f1.ent.vec_time) return false;
  if (f2.ent.vec_time > f1.exi.vec_time) return false;

  return true; /*f1 and f2 are concurrent*/
}
```

图 3.3-2 判断并行函数对伪代码

3.3.3. 基于 CFP 的检测技术实现

第一步对每个输入，分析其能覆盖的并行函数对，并生成并行函数对集合(aggregated CFP)。第二步，选择最小输入集合，使其能够覆盖并行函数对集合。但这是一个 NP 问题，所以采用贪心算法来得到近似最优解。如图 3.3-3 所示，在 3 个输入中，Input1 可以覆盖的 CFP 最多，因此先选择 Input1；此时只有两个 CFP 没有被覆盖，在 Input2 和 Input3 中选择 Input2，因为它可以覆盖这两个函数对；最后发现没有被覆盖的 CFP 集合为空，算法结束。第三步，使用已有的 data race 和 atomicity violation 检测器对选择的输入进行检测。

Input1: $CFP_1=\{\{f_1,f_2\},\{f_2,f_3\},\{f_2,f_4\},\{f_4,f_5\}\}$
Input2: $CFP_2=\{\{f_1,f_2\},\{f_3,f_4\},\{f_3,f_5\}\}$
Input3: $CFP_3=\{\{f_2,f_3\},\{f_3,f_4\}\}$
$CFP_{Aggregated}=\{\{f_1,f_2\},\{f_2,f_3\},\{f_2,f_4\},\{f_4,f_5\},\{f_3,f_4\},\{f_3,f_5\}\}$

Step 1:
Selected input —— Input1
Selected functions —— $\{f_1, f_2, f_3, f_4, f_5\}$
$CFP_{Uncovered}=\{\{f_3,f_4\},\{f_3,f_5\}\}$

Step 2:
Selected input —— Input2
Selected functions —— $\{f_3, f_4, f_5\}$
$CFP_{Uncovered}=\emptyset$

图 3.3-3 选择输入

3.3.4. 缺点

采用这种方法选择部分输入，可能会导致 False positive 比较高。这是因为，某些并行函数对在特定输入下执行时，才会产生 Bug。而本文使用 CFP 选择了部分输入，可能并不包括这个特定输入，所以 Bug 并未被检测到。

还有目前只针对 data-race 和 atomicity-violation 两种类型的 bug 进行检测。

3.4. Goldilocks

Java 是目前而言最受欢迎的编程语言之一，在业界中，有越来越多的公司选择 Java 作为开发语言。Java 是以共享内存作为其内存模型的语言[2]，因此也会遇到多线程环境下的数据竞争问题。Goldilocks[1]是在 Java 虚拟机上实现的，用于数据竞争动态检测的算法，它引入了一种新的运行时错误，被称为 DataRaceException。每当程序中出现数据竞争的情况时，虚拟机就会抛出 DataRaceException 的异常，可以及时处理，以免问题继续扩散。而如果在执行过程中，没有任何的 DataRaceException 抛出，则可以认为在程序的执行流中没有数据竞争。

3.4.1. 相关工作

Goldilocks 是一个基于 Eraser[3]提出的 Lockset 算法扩展而来的算法，Lockset 算法的思想就是，在多线程程序中，一般程序员都会使用锁对临界区进行保护，临界区中一般都是共享变量的访问操作，如果一个共享变量在程序多线程执行过程中能够始终被一个或多个锁保护的话，那么在该共享变量上肯定不会发生数据竞争。反之，则有可能发生数据竞争。基于这样的观察，提出了 Lockset 的算法：

```
Let locks held(t) be the set of locks held by thread t.
For each v, initialize C(v) to the set of all locks.
On each access to v by thread t,
    set C(v) := C(v) ∧ locks_held(t);
    if C(v) = { }, then issue a warning.
```

算法 3.4-1 Lockset

其思想非常简练，一共有两个阶段，第一个阶段为初始化阶段，对于每个线程 t，都会维护一个 locks_held(t)表明当前获得的锁集；对于每一个共享变量 v，这个变量在初始化的时候获得程序执行过程中的所有可能锁集 C(v)。之后是更新阶段，对于当前线程的每一次读写操作，都会更新当前被访问变量的候选集合 C(v)。

Lockset 算法是一种动态检测的算法，需要动态地对程序代码进行改写，用以统计读写、加锁释放锁、内存声明还有线程创建等操作。基于对这些操作的 Trace，可以用 Lockset 得到会出现数据竞争的变量。

3.4.2. Goldilocks 实现

3.4.2.1. 形式化描述

Goldilocks 提出了一种形式化的描述方法，来对读写、加锁释放锁操作等等进行更好的描述与刻画。它将所有需要关注的操作抽象成了三种 kind，分别是 SyncKind，DataKind 和 AllocKind。对于每一种 Kind，都是一些抽象操作的集合，这些集合在 Goldilocks 中被定义为：

- $SyncKind = \{acq(o), rel(o) \mid o \in Addr\} \cup \{read(o,v), write(o,v) \mid o \in Addr \wedge v \in Volatile\} \cup \{fork(u), join(u) \mid u \in Tid\} \cup \{commit(R,W) \mid R,W \subseteq Addr \times Data\}$
- $DataKind = \{read(o,d) \mid o \in Addr \wedge d \in Data\} \cup \{write(o,d) \mid o \in Addr \wedge d \in Data\}$
- $AllocKind = \{alloc(o) \mid o \in Addr\}$
- $Kind = SyncKind \cup DataKind \cup AllocKind$

图 3.4-1 Goldilocks 形式化定义

而关于 Lockset 的更新规则，也进行了相应的改变：

1. $\sigma(t,n) \in \{read(o,d), write(o,d)\}$:
   if $LS(o,d) \neq \emptyset$ and $t \notin LS(o,d)$
   report data race on (o,d)
   $LS(o,d) := \{t\}$
2. $\sigma(t,n) = read(o,v)$:
   foreach (o',d):
   if $(o,v) \in LS(o',d)$ add t to $LS(o',d)$
3. $\sigma(t,n) = write(o,v)$:
   foreach (o',d):
   if $t \in LS(o',d)$ add (o,v) to $LS(o',d)$
4. $\sigma(t,n) = acq(o)$:
   foreach (o',d):
   if $(o,l) \in LS(o',d)$ add t to $LS(o',d)$
5. $\sigma(t,n) = rel(o)$:
   foreach (o',d):
   if $t \in LS(o',d)$ add (o,l) to $LS(o',d)$
6. $\sigma(t,n) = fork(u)$:
   foreach (o',d):
   if $t \in LS(o',d)$ add u to $LS(o',d)$
7. $\sigma(t,n) = join(u)$:
   foreach (o',d):
   if $u \in LS(o',d)$ add t to $LS(o',d)$
8. $\sigma(t,n) = alloc(x)$:
   foreach $d \in Data$: $LS(x,d) := \emptyset$
9. $\sigma(t,n) = commit(R,W)$:
   foreach (o',d):
   if $LS(o',d) \cap (R \cup W) \neq \emptyset$
   add t to $LS(o',d)$
   if $(o',d) \in R \cup W$:
   if $LS(o',d) \neq \emptyset$ and $\{t, TL\} \cap LS(o',d) = \emptyset$
   report data race on (o',d)
   $LS(o',d) := \{t, TL\}$
   if $t \in LS(o',d)$
   add $R \cup W$ to $LS(o',d)$

图 3.4-2 更新规则

在这样的抽象下，使得 Goldilocks 对于事务，以及原子变量等等其他的同步操作有更好的支持。

3.4.2.2. 例证

```
Class IntBox {  int data; }
IntBox a, b; // Global variables

Thread 1:              Thread 2:       Thread 3:
-------------------    --------        -------------
tmp1 = new IntBox();   acq(ma);        acq(mb);
tmp1.data = 0;         tmp2 = a;       b.data = 2;
acq(ma)                rel(ma);        tmp3 = b;
a = tmp1;              acq(mb);        rel(mb);
rel(ma);               b = tmp2;       tmp3.data = 3;
                       rel(mb);
```

图 3.4-3 示例代码

对于示例中的代码，线程 1 中的代码先执行，之后再执行线程 2 中的代码，最后运行线程 3。Goldilocks 算法在其上的运行情况如图所示：

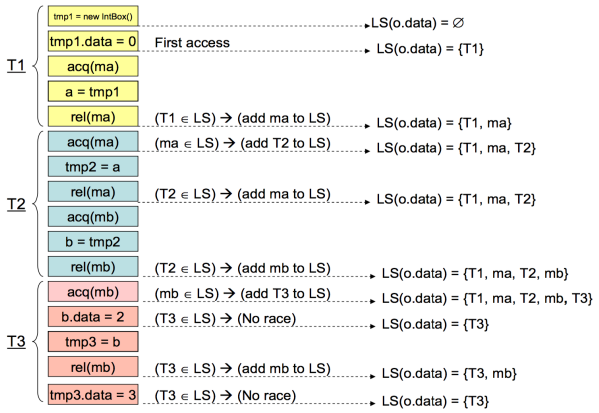

图 3.4-4 更新规则

在 Goldilocks 的运行情况中可以得出，在这样的执行顺序中，不会出现数据竞争的情况。

3.4.3. 不同实现方式比较

因为 Goldilocks 是基于 Lockset 的，所以也是一种动态的检测方法。Goldilocks 会监控 Java 字节码的运行，在这样的等级上，每一次变量的读写访问，或者是同步操作的指令，都是一条字节码的指令。同时每一个字节码的指令都会对应一个变量或者源码中的一行。

在已经发表的论文中，Goldilocks 的实现有两个。在较早时间发表的论文中，Goldilocks 是实现在 Kaffe 的运行时引擎的解释器模式下，Kaffe 是一个使用 C 语言实现的 Java 虚拟机。这样实现的优势在于能够得到源代码在虚拟机内部，内存的布局，以及可以使用虚拟机内部实现的诸多算法。

除了上述方法，Flanagan 和 Freund 在 ROADRUNNER 上也实现了 Goldilocks 算法，并发表了论文[4]。ROADRUNNER 是一个动态分析工具，在这样的实现中，Goldilocks 算法是通过在加载的时候，动态地插入 Java 字节码指令来实现的。

3.4.4. 与 Eraser 比较

之前的 Lockset 算法，是一种比较保守的算法，因此常常会在一些并没有数据竞争的

情况下也会报出异常。举例来说明，在下面的实现中，Thread1 中的对于 data 变量的初始化就会报出数据竞争的异常，因为没有任何一个锁保护该变量。

而在 Goldilocks 的实现中，对于数据竞争的检测是更加准确的。除此之外，Goldilocks 的形式化抽象，使得它可以支持更多的同步方法。比如对于内存事务的方法进行同步的操作，以及 volatile 变量的读写等等。这些同步方法在传统的算法中都是没有支持的，而在 Goldilocks 就可以得以支持。

4. 比较

Goldilocks 仅使用动态分析技术，而 MUVI、CFP 和 ConSeq 系统将静态分析和动态分析技术相结合，体现了 Hybrid 原则。

CFP 系统在选择输入的过程中，并没有选取最优解，而是使用贪心算法得到近似最优解。它牺牲结果的准确性来简化复杂的 NP 问题。这体现了 KISS(Keep It Simple and Stupid) 原则。

每个系统试图解决某一类或某几类的 Concurrency bug，而不是一次性解决所有的问题。这体现了避免过度通用(Avoid Excessive generality)原则。表 4-1 总结了不同系统检测的 concurrency bug 种类。

表 4-1 检测 Bug 种类

system	atomicity violation	multi-variable synchronization	infinite loop
MUVI	√	√	
ConSeq	√	√	√
Goldilocks	√		
CFP	√		

最后，很多系统都复用了已有的开源工具，而不是再次实现一遍。这体现了 open design 原则。

5. 总结与展望

本文提到的几个 concurrency bug 检测系统，构思精巧，能够有效针对不同的方面解决。

参考文献

[1] Elmas T, Qadeer S, Tasiran S. Goldilocks: a race and transaction-aware java runtime[C]// Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2007, 42(6): 245-255.

[2] Manson J, Pugh W, Adve S V. The Java memory model[M]. ACM, 2005.

[3] Savage S, Burrows M, Nelson G, et al. Eraser: A dynamic data race detector for multithreaded programs[J]. ACM Transactions on Computer Systems (TOCS), 1997, 15(4): 391-411.

[4] Flanagan C, Freund S N. The RoadRunner dynamic analysis framework for concurrent programs[C]//Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. ACM, 2010: 1-8.

[5] Lu S, Park S, Hu C, et al. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs[C]//Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. ACM, 2007, 41(6): 103-116.

[6] Zhang W, Lim J, Olichandran R, et al. ConSeq: detecting concurrency bugs through sequential errors[C]// Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems. ACM, 2011, 46(3): 251-264.

[7] Deng D, Zhang W, Lu S. Efficient concurrency-bug detection across inputs[C]. Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, 2013, 48(10): 785-802.

[8] Lu S, Park S, Seo E, et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics[C]// Proceedings of the 13th international conference on Architectural support for programming languages and operating systems. ACM, 2008, 43(3): 329-339.

[9] Huang R, Halberg E, Suh G E. Non-race concurrency bug detection through order-sensitive critical sections[C]// Proceedings of the 40th Annual International Symposium on Computer Architecture. ACM, 2013, 41(3): 655-666.

[10] Lu S, Park S, Zhou Y. Detecting concurrency bugs from the perspectives of synchronization intentions[J]. Parallel and Distributed Systems, IEEE Transactions on, 2012, 23(6): 1060-1072.

[11] Lu S, Tucek J, Qin F, et al. AVIO: detecting atomicity violations via access interleaving invariants[C]// Proceedings of the 12th international conference on Architectural support for programming languages and operating systems. ACM, 2006, 40(5): 37-48.