南京财经大学 软件需求分析与设计演示报告

2018 —— 2019 第 一 学期

课程名称：软件需求分析与设计

任课教师：刘阳

学生姓名：方泽强

班　　级：软件 1601

学　　号：2120162016

演示报告题目：基于 JPF 的哲学家就餐模型检验

内容摘要：本文使用 Javapathfinder 模型检验工具，在 Eclipse 平台上，将被测系统抽象成 java 语言描述的模型，配置 jpf 形式化规约，以此来验证哲学家就餐模型在其经典规则下存在发生死锁的情况

关 键 词：JPF 哲学家就餐问题 模型检验

目　录

一、 模型检验原理

模型检验的原理朴素易懂，如图 1 所示就是将需求用形式化方法转述成规则在模型检验工具里进行配置，在将抽象出来的模型放入工具中检验其是否满足所配置的规则，模型检验工具会相应的输出结果如满足，不满足并举出反例或者提示内存不足。

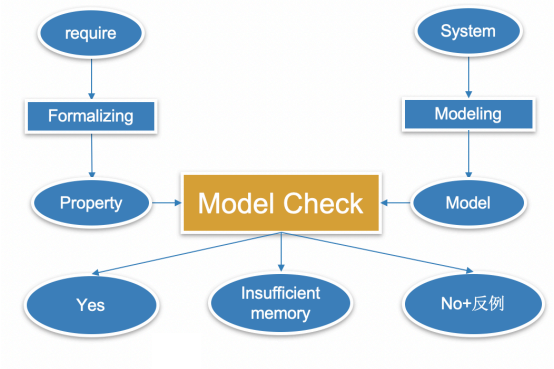

图 1 模型检验原理

作为形式方法的模型检查不依赖于基于经验的猜测测试。在理论上，如果存在违反给定规范的情况，模型检查将会找到它。模型检查是一种严格的方法，它可以详尽地探索所有可能的 SUT[1]行为。在检查所有数据组合或发现错误之前，模型检查不会停止,如图 2 所示。

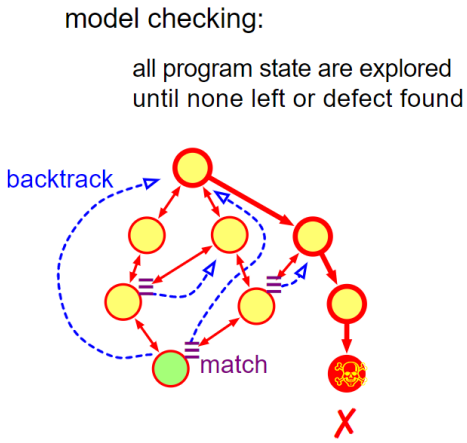

图 2 检验工作示意图

[1] SUT：这里指被测系统，即 system under test

本报告所使用的模型检验工具 Javapathfinder 正是运用典型的模型检验原理来进行验证工作，SUT 在这里以 java 的 class 类型文件为主，而规约描述方法为 jpf 文件，验证的结果以 jpf 专有的 report 形式输出，如图 3 所示。

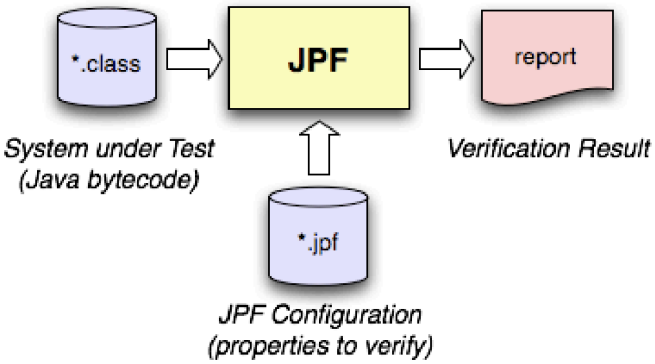

图 3 JPF 工作模型

以检测除零错误的例子来说明 javapathfinder 的模型检验原理，源代码如下：

```
import java.util.Random;

public class Rand {
	public static void main (String[] args) {
		Random random = new Random(42);          // (1)

		int a = random.nextInt(2);                  // (2)
		System.out.println("a=" + a);

		//... lots of code here

		int b = random.nextInt(3);                  // (3)
		System.out.println("    b=" + b);

		int c = a/(b+a -2);                         // (4)
		System.out.println("       c=" + c);
	}
}
```

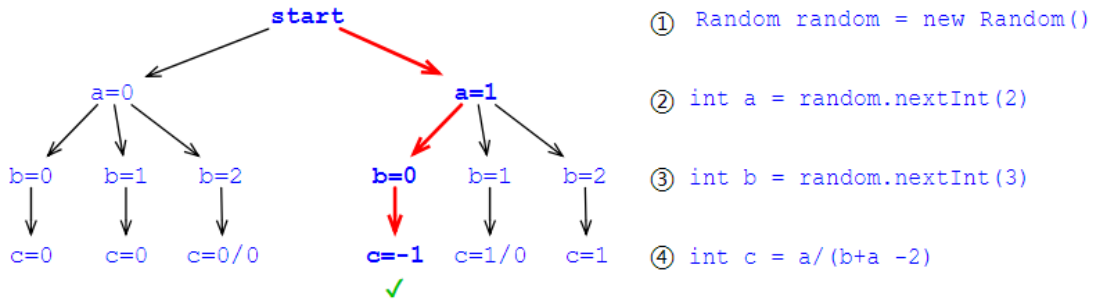

图 4 Java 应用程序运行模型

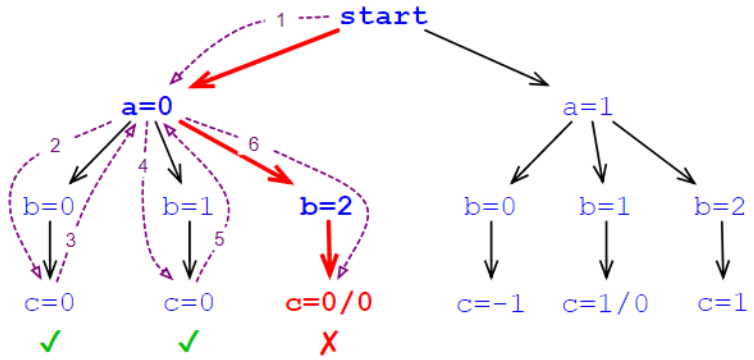

图 5 检测除零错误验证模型

无论验证系统发现什么错误，JPF 还保持完整的跟踪（执行路径）这个错误产生的路径（用红色箭头表示），这意味着我们不必调试程序来查找。

二、 模型检验工具

此次模型检验演示我选用的工具是 Java™Pathfinder（JPF）。JPF 是 Java™字节码程序的可扩展软件模型检查框架。该系统是在 2005 年开源的 NASA Ames 研究中心开发的，2017 年以前可以在 Apache-2.0 许可下在该服务器上免费获得，现在需要从其 github 主页上下载 jpf-core 核心文件。

JPF 不仅仅是一个模型检查器：它是一个可以用于模型检查的 JVM[2]。如果你熟悉形式化方法，可能需要了解其支持的模型检查方法和功能。以下是 JPF 的主要特点：

[2] JVM：这里指 Java 虚拟机

- **显式状态**模型检查是 JPF 的标准操作模式。这意味着 JPF 会跟踪局部变量，堆栈帧，堆对象和线程状态的具体值。除了有意不同的调度行为外，JPF 应该像普通的 JVM 一样输出相同的结果。当然它更慢（它是在 JVM 上运行的 JVM，执行大量额外工作）。

- **符号执行**意味着 JPF 可以使用从当前执行路径（例如 x>0 && x<43）中得知如何使用某个变量获得的符号值来执行程序。此外，JPF 甚至可以混合具体代码和符号的执行，或在它们之间切换。有关详细信息，请参阅 Symbolic Pathfinder 项目文档。

- **状态匹配**是避免不必要工作的关键机制。程序的执行状态主要由堆和线程堆栈快照组成。当 JPF 执行时，它检查每个新状态是否已经看到相同的状态，在这种情况下，没有用于继续当前执行路径，并且 JPF 可以回溯到最近的非探索的非确定性选择。可以由用户控制哪些变量对状态匹配有贡献，以及如何执行状态匹配。

- **回溯**意味着 JPF 可以检查以前的执行状态，以查看是否还有未经探索的选择点。例如，如果 JPF 达到程序结束状态，它可以向后行走以找到尚未执行的不同可能的调度序列。虽然理论上可以通过从头开始重新执行程序来实现，但如果从状态空间存储优化的角度看，回溯是一种更有效的机制。

- **局部顺序减少**是 JPF 用来最小化线程之间的上下文切换，这不会导致有趣的新程序状态。这是通过检查哪些指令可以具有线程间效果而即时完成的，即没有事先分析或注释程序。虽然这很快，但它无法处理“菱形”，因为它无法超越当前的执行。

JPF 的灵活性是通过提供大量扩展点的架构实现的，其中最重要的扩展点是：

- Search Policies - 在被搜索的状态空间顺序中进行控制
- Listeners - 监视 JPF 执行并与之交互（例如检查新属性）
- Peer Classes - 在主机 VM 级别建模库和执行代码
- Bytecode Factories - 提供不同的执行模型（如符号执行）
- Publishers - 生成特定报告

三、 验证对象

本报告验证对象为哲学家就餐模型，检验该模型执行时是否会发生死锁。

哲学家就餐问题是在计算机科学中的一个经典问题，如图 6 所示，是用来演示在并行计算中多线程同步(Synchronization)时产生的问题。在 1971 年，著名的计算机科学家艾兹格·迪科斯彻提出了一个同步问题，即假设有五台计算机都试图访问五份共享的磁带驱动器。随后，这个问题被托尼·霍尔重新表述为哲学家就餐问题。这个问题可以用来解释死锁和资源耗尽的现象。

图 6 哲学家就餐问题

问题陈述：

- 五位哲学家坐在圆桌旁，拿着一碗意大利面。叉子放在相邻的哲学家之间。
- 每位哲学家必须交替思考和吃饭。而哲学家只有拿左右叉子时才能吃意大利面。
- 每个叉子至多只能由一位哲学家持有，因此当且仅当另一位哲学家没有使用它时，该哲学家才能使用它。
- 在一位个体哲学家吃完之后，他们需要放下两把叉子，以便叉子可供他人使用。
- 一个哲学家可以获取他们右边或左边的叉子，在它们可用的前提下；但哲学家在获得两个叉子之前不能开始进食。
- 进食不受意大利面条或胃部空间的限制; 假设面条无限供应和哲学家有无限需求。

问题是如何设计一个行为方式（并发算法），这样任何哲学家都不会饿死; 也就是说，每个人都可以永远地继续在吃和思考之间交替，假设没有哲学家可以知道其他人何时可能想吃或想（即哲学家间不能进行交流）。

将上述规约抽象为模型，如图 7 所示:

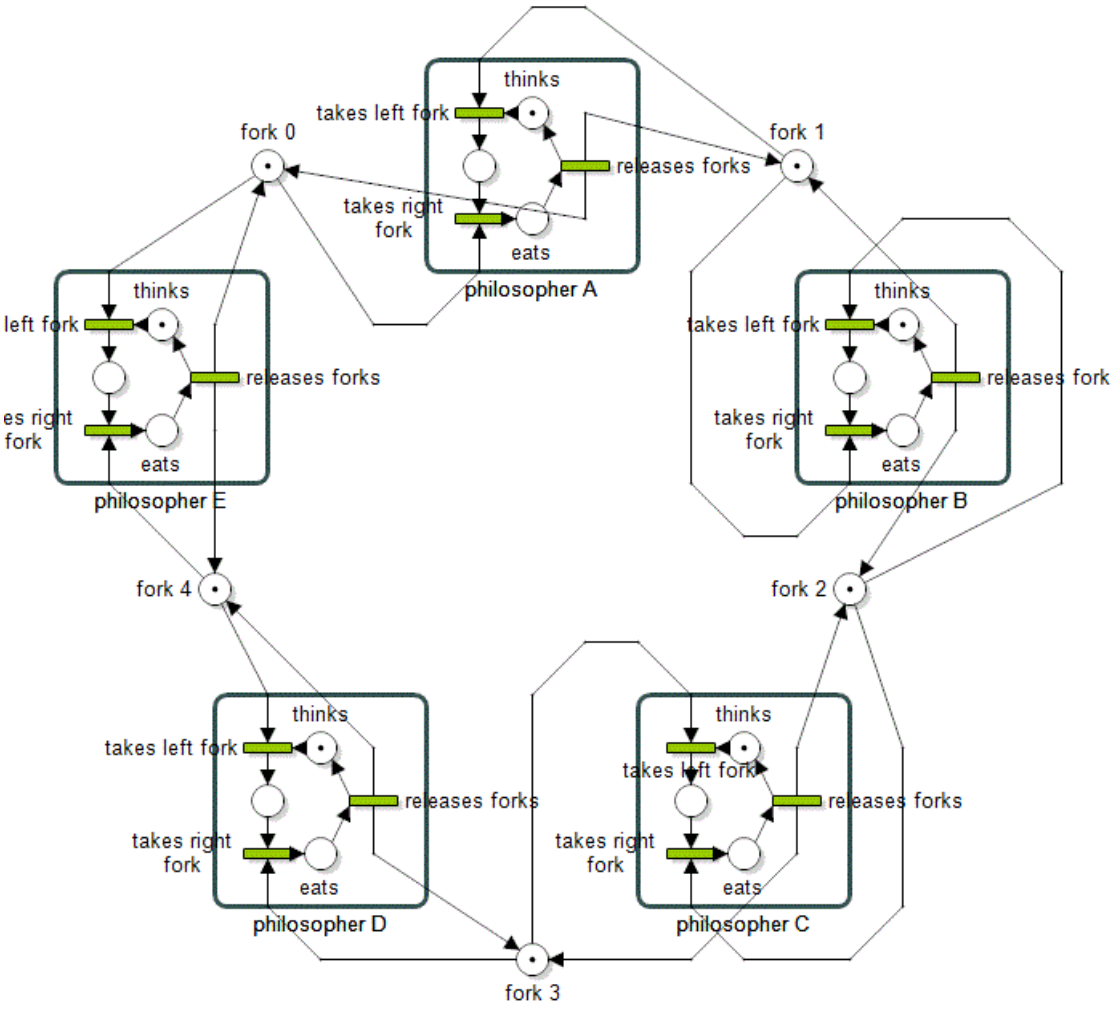

图 7 哲学家就餐行为模型

四、 验证过程

1. 配置完成jpf-core后，将其倒入eclipse
2. 新建一个工程命名为DiningPhil_Jpf
3. 新建类DiningPhil.java
4. 用java代码实现上述模型,如图8所示具体步骤如下

- 声明一个静态Fork类
- 声明一个继承Thread父类的静态Philosopher类
- Philosopher构造方法中应有左右两个Fork类属性
- 在Philosopher类中重写父类thread的run方法
- 定义一个静态整型：哲学家数量，设置为5
- 编写主函数实现模型方法

```
public class DiningPhil {

  static class Fork {
  }

  static class Philosopher extends Thread {

    Fork left;
    Fork right;

    public Philosopher(Fork left, Fork right) {
      this.left = left;
      this.right = right;
      //start();
    }

    @Override
    public void run() {
      // think
      synchronized (left) {
        synchronized (right) {
          // eat
        }
      }
    }
  }

  static int nPhilosophers = 5;


  public static void main(String[] args) {
    if (args.length > 0) {
      nPhilosophers = Integer.parseInt(args[0]);
    }

    Fork[] forks = new Fork[nPhilosophers];
    for (int i = 0; i < nPhilosophers; i++) {
      forks[i] = new Fork();
    }
    for (int i = 0; i < nPhilosophers; i++) {
      Philosopher p = new Philosopher(forks[i], forks[(i + 1) % nPhilosophers]);
      p.start();
    }

  }
}
```

图 8 java 代码实现

5. 新建配置参数文件DiningPhil.jpf，如图9所示。

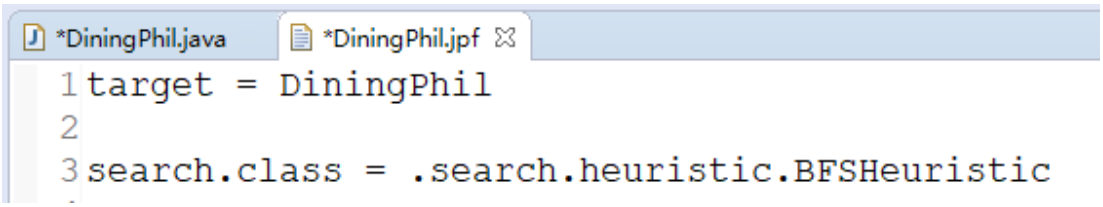

图 9 配置参数

6. 右键.jpf文件在Run Configuration中选中run-JPF运行,如图10所示。

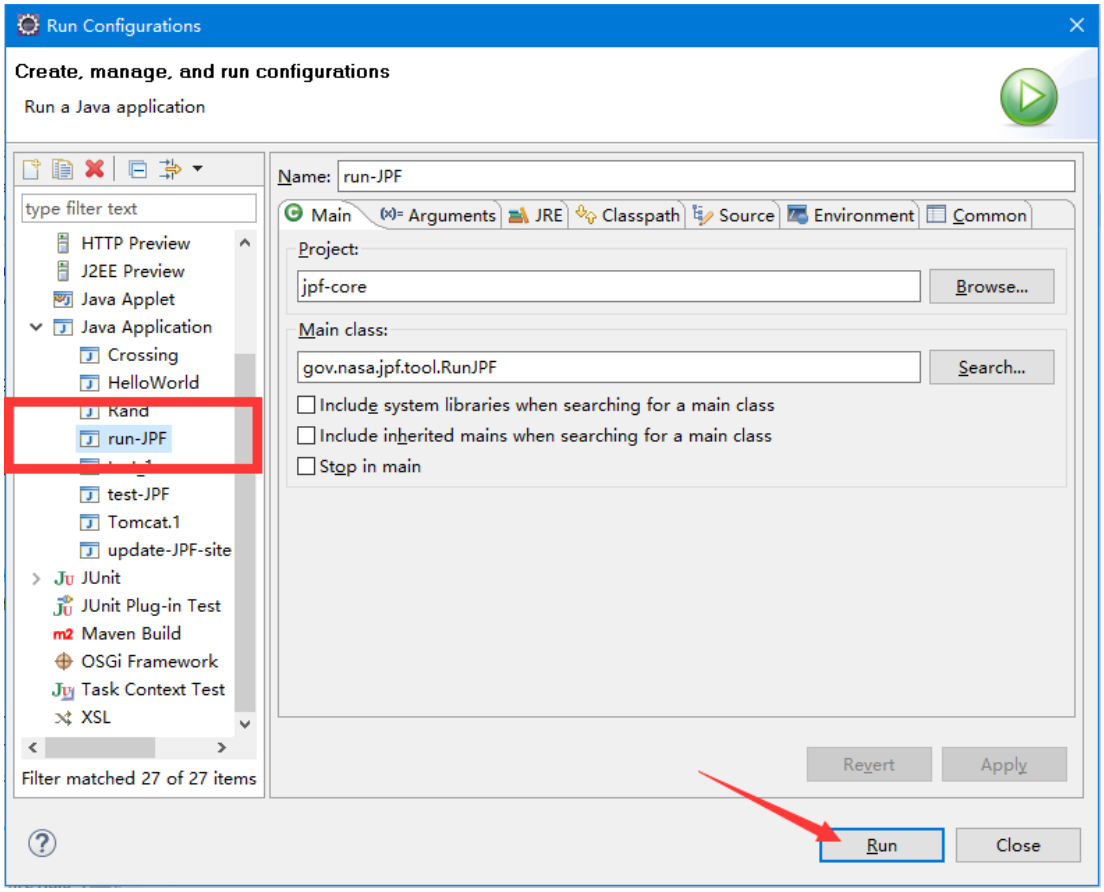

图 10 运行 JPF

7. 在控制台Console中查看验证结果，有出错点记录error,程序运行快照snapshot验证结果输出以及总体的数据统计，详细见下文五、验证结果分析。

五、 验证结果

1. 控制台开头中显示的信息如图 11，其注明了 SUT 和验证开始时间。

```
JavaPathfinder core system v8.0 - (C) 2005-2014 United States Government. All rights reserved.


====================================================== system under test
DiningPhil.main()

====================================================== search started: 18-12-13 上午11:10
```

图 11 输出结果一

2. 紧接着是错误报告 error，如图 12 所示，显示有一个 error；这个 error 类型为死锁，死锁的发生位置也很清晰地在报告里呈现；可以看到五个进程对象（即五个哲学家）从 Thread-1 到 Thread-5 的状态都为阻塞 BLOCKED，也都无法继续在后台运行。

```
====================================================== error 1
gov.nasa.jpf.vm.NotDeadlockedProperty
deadlock encountered:
  thread DiningPhil$Philosopher:{id:1,name:Thread-1,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  thread DiningPhil$Philosopher:{id:2,name:Thread-2,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  thread DiningPhil$Philosopher:{id:3,name:Thread-3,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  thread DiningPhil$Philosopher:{id:4,name:Thread-4,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  thread DiningPhil$Philosopher:{id:5,name:Thread-5,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
```

图 12 输出结果二

3. 在 error 总览报告的下方是程序运行过程的快照 snapshot，如图 14 所示；以线程 Threat-1 为例如图 13 所示，大括号内表示的是该线程对象的 id,name 等各个属性值；它所占用的资源是内存@162 上的 Fork 对象，其在请求内存@163 上的 Fork 对象时不得而导致状态阻塞；而 call stack[3]显示其在运行时被调用方法是 Philosopher 中的 run( )方法，具体可见 java 源代码中的第 39 行。

```
====================================================== snapshot #1
thread DiningPhil$Philosopher:{id:1,name:Thread-1,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  owned locks:DiningPhil$Fork@162
  blocked on: DiningPhil$Fork@163
  call stack:
        at DiningPhil$Philosopher.run(DiningPhil.java:39)
```

图 13 输出结果三（部分）

[3] Call stack 即方法调用堆栈，这里用来追溯发生错误的方法位置

```
==================================================== snapshot #1
thread DiningPhil$Philosopher:{id:1,name:Thread-1,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  owned locks:DiningPhil$Fork@162
  blocked on: DiningPhil$Fork@163
  call stack:
        at DiningPhil$Philosopher.run(DiningPhil.java:39)

thread DiningPhil$Philosopher:{id:2,name:Thread-2,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  owned locks:DiningPhil$Fork@163
  blocked on: DiningPhil$Fork@164
  call stack:
        at DiningPhil$Philosopher.run(DiningPhil.java:39)

thread DiningPhil$Philosopher:{id:3,name:Thread-3,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  owned locks:DiningPhil$Fork@164
  blocked on: DiningPhil$Fork@165
  call stack:
        at DiningPhil$Philosopher.run(DiningPhil.java:39)

thread DiningPhil$Philosopher:{id:4,name:Thread-4,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  owned locks:DiningPhil$Fork@165
  blocked on: DiningPhil$Fork@166
  call stack:
        at DiningPhil$Philosopher.run(DiningPhil.java:39)

thread DiningPhil$Philosopher:{id:5,name:Thread-5,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  owned locks:DiningPhil$Fork@166
  blocked on: DiningPhil$Fork@162
  call stack:
        at DiningPhil$Philosopher.run(DiningPhil.java:39)
```

图 14 输出结果三（完整）

4. 而最终报告呈现的总体结果 result 就是“有错误”有一个错误类型为死锁，发生在继承了线程类的对象 Philosop 里；result 下方还有一个对验证过程的统计报告 statistics 其中包含了时间统计 elapsed time 等一系列统计类型名目。

```
====================================================== results
error #1: gov.nasa.jpf.vm.NotDeadlockedProperty "deadlock encountered:   thread DiningPhil$Philosop..."

====================================================== statistics
elapsed time:       00:00:02
states:             new=15311,visited=34935,backtracked=50245,end=1
search:             maxDepth=23,constraints=0
choice generators:  thread=11210 (signal=0,lock=6885,sharedRef=2476,threadApi=14,reschedule=1835), data=0
heap:               new=11730,released=39019,maxLive=387,gcCycles=44387
instructions:       538250
max memory:         462MB
loaded code:        classes=64,methods=1479

====================================================== search finished: 18-12-13 上午11:10
```

图 15 输出结果四

综合 jpf 验证工具的输出结果，很明显在这个以 java 语言描述的哲学家就餐问题模型中按照其问题规约进行验证时必定存在一种情况导致被测系统陷入死锁状态；可以得出结论：哲学家不能永远的在吃和思考行为之间交替，当哲学家们左手（或右手）都同时握有一根叉子，同时又在等待另一根叉子时，哲学家们都陷入了等待彼此释放叉子的状态，最终导致哲学家们无法进行下一个行为因资源饥饿而消亡。

总结

以上难点之一在于 Java pathdinder 的安装与配置，运行需要 jdk1.8 环境，且倒入 jpf-core 后须手动更改编译路径；第二难在对哲学家就餐问题模型的 java 实现；第三难在配置该程序相应验证参数。

选用哲学家就餐问题作为此次验证对象，一是因为大二的操作系统原理接触过这个经典问题，其内容为大部分读者所熟知；二是由于兴趣使然，大二就曾了解过生产消费者，读者写者及银行家算法等问题，其中哲学家就餐问题较容易实现。

如果时间与能力允许，希望在该模型中引入第三方仲裁者（可比喻为服务生），让每个哲学家在拿筷子前都询问这个服务生，在保证哲学家相互不交谈的前提下，解决哲学家就餐的死锁问题，用 java 代码实现并用 javapathfinder 再次进行验证。

参考文献

[1] Willem Visser, Corina S. Pasareanu and Sarfraz Khurshid. Test Input Generation with Java PathFinder. In NASA Ames Research Center, California, USA.

[2] JavaPathFinder 官方网站：http://javapathfinder.sourceforge.net/

[3] jpf-core 工具下载地址：https://sourceforge.net/projects/javapathfinder/files/latest/download

[4] 运行 jpf 的集成开发平台 Eclipse 下载地址：https://www.eclipse.org/

[5] JAVA 软件开发工具包 JDK1.8 下载地址：https://www.oracle.com/technetwork/java/javase/downloads/index.html

[6] 在 Eclipse 平台上运行 jpf 的方法：http://javapathfinder.sourceforge.net/Running_JPF.html

[7] 哲学家问题维基百科详解：https://en.wikipedia.org/wiki/Dining_philosophers_problem