诚　信　声　明

我声明，所呈交的课程论文是本人在老师指导下进行的研究工作及取得的研究成果。据我查证，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得其他教育机构的学位或证书而使用过的材料。我承诺，论文中的所有内容均真实、可信。

课程论文作者签名：　　签名日期：　　年　月　日

可视化的 Java 多线程程序错误定位工具

摘要

随着多核 CPU 的出现，多线程程序在软件开发领域得到广泛的应用。与此同时，多线程程序当中的并发错误也给软件开发者带来的很多困难。2009 年，Sangmin Park 等人提出了名为 Falcon 的新方法用来进行多线程程序中并发错误的错误定位。该方法可以按照可疑度的大小报告程序中的数据访问模式。但是，Falcon 纯文本的展示方式仍然不便于程序员快速定位错误。所以在本篇论文中，作者首先介绍 Falcon 的实现原理，然后以 Falcon 为基础，开发一个可视化的 Java 多线程程序错误定位工具。

关键词：错误定位；软件调试；Java；多线程

Visual Fault Localization Tool for Java Multi-threaded Program

Abstract

With the widespread deployment of multi-core processors, multi-thread programming becomes popular in software industry. Meanwhile, concurrent faults in multi-threaded programs cause troubles to software developers. In 2009, Sangmin Park *et al.* presented a new dynamic fault-localization technique to locate concurrent faults in multi-thread programs. It can report data access patterns with suspiciousness scores. However, the results are presented in plain text format which are still hard to read and analyze for programmers. This paper introduces Falcon's algorithm and then implements Falcon approach with a visualization tool.

Keywords：Fault localization; Software debuging; Java; Multi-threaded

目录

1 课题背景

近年来，随着多核处理器的出现和迅速发展，并行计算、多线程程序逐渐在软件开发领域得到了广泛的使用。与此同时，并行计算的广泛使用也给软件测试提出了新的挑战。

1.1 程序的并发错误

因为并行程序本身的特点和缺乏有效的并行程序设计方法与工具，使得正确编写并行程序，调试和优化并行程序都很困难。一方面，编写并行程序比编写串行程序更加困难。除了串行程序常见的错误[1]之外，并行程序还有诸如死锁、数据争用等独有的错误。所以一段并行程序中存在错误的可能性更大。另一方面，测试并行程序比测试串行程序更加困难。这是因为：第一，并行程序的运行结果具有不确定性，难以重现错误；第二，程序的运行结果牵涉到多线程操作，使得错误不易定位 [1]。

微软公司在 2007 年发布的针对 684 名员工的调查报告显示 66% 的受访者需要经常处理并发程序中的问题（包括测试、调试、修复）。超过 72% 的受访者认为重现并行程序中的错误困难或非常困难，63.4% 的受访者需要花费数天才能解决并行程序中的错误。在微软公司的调查报告中，受访者把更好的错误定位工具放在了他们需求的首位 [2]。由此可见，在软件开发的过程中，并发程序中的错误制约了软件开发效率。软件开发领域需要一款能够准确地在并发程序中进行错误定位的工具。

1.2 Java 并发程序的错误定位技术

在软件调试过程中，错误定位是识别程序中错误准确位置的活动。软件调试是软件开发过程中最费时费力的活动，而错误定位又是调试过程中最费时费力的活动。所以，软件开发领域中需要错误定位技术来指导程序员找到程序中的错误 [3]。

为了解决 Java 并发程序错误定位这个问题，一些学者经过不断的努力，提出了多种方法。Ronsee 等人提出了一种方法可以用来检测数据争用，并且可以记录和重复并发程序 [4]。Shan Lu 等人尝试通过检识别非顺序的线程交错来检测原子性破坏（AVIO）[5]。

[1]在本篇论文中，漏洞、缺陷、错误可以互换地使用。

1.3 Java 程序的并发错误

1.3.1 标识说明

在具体描述这两种数据访问模式之前，首先约定本篇论文中关于不同线程对共享变量读写的表述方式。本文使用 $b_{t,S}$ 来表示一个线程访问了一个共享变量。其中 b 是访问的类型，分为读（R）或者写（W）两种；t 表示操作变量的线程，通常用线程号（一个整数）进行表示；S 表示含有被访问的共享变量的语句。例如，$R_{1,S2}$ 表示线程 1 在语句 $S2$ 中对一个共享变量进行了读操作。有的时候可以通过上下文得出含有共享变量的被访问语句，所以为了简化描述，后文中也会在不影响理解的情况下将 $R_{1,S2}$ 简写为 R_1。

1.3.2 错误类型的划分

如果想要定位并发程序中的错误，首先要能够发现并发错误出现的规律和找到并发错误的方法。近年来，学者们尝试建立了一些有错误倾向的数据访问模式（data access pattern），如果在并发程序运行的过程中，不同线程访问共享数据的方式符合这些数据访问模式，就有出现并发错误的可能性。这些数据访问模式包括两种：顺序破坏（order violation）和原子性破坏（atomicity violation）。

顺序破坏

顺序破坏就是访问一个共享变量的两个线程（其中一个线程进行写操作）因为访问顺序交错导致读、写错误的数据。顺序破坏一共有三种形式，如表 1-1所示。

表 1-1: 顺序破坏模式

	线程读写	描述
1	$R_1 - W_2$	写入意外的数值
2	$W_1 - R_2$	读出意外的数值
3	$W_1 - W_2$	线程 1 写入的数值丢失

原子性破坏

一个原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行倒结束，中间不会被其它线程打断，否则就可能会产生错误。这种不可被打断的特性被称作原子性。原子性破坏就破坏了原子操作

的原子性，即一个线程操作共享变量的过程中，另外一个线程操作被错误的插入。原子性破坏一共有 5 种形式，如表 1-2所示。

表 1-2: 原子性破坏模式

	线程读写	描述
1	$R_1 - W_2 - R_1$	不可重复读
2	$W_1 - W_2 - R_1$	线程 1 数据被线程 2 意外修改
3	$W_1 - R_2 - W_1$	线程 2 读入错误数据
4	$R_1 - W_2 - W_1$	丢失修改
5	$W_1 - W_2 - W_1$	丢失修改

1.4 Falcon 方法

2009 年，乔治亚理工学院（Georgia Institute of Technology）的 Sangmin Park 等人提出了一个名为 Falcon（Fault Localization in Concurrent Programs）的新方法。Falcon 可以定位多线程并发程序中错误的数据访问模式，并且可以根据被测程序的通过、失效数给出每个数据访问模式的可疑度，从而便于用户快速地定位错误 [6]。用户可以根据 Falcon 得到的可疑度，找出最有可能存在错误的数据访问模式，然后根据数据访问模式的信息可以定位到该错误所在的行号和变量。

Falcon 算法通过两个步骤来识别程序中的多线程错误。第一步，需要实时地监测程序不同线程对共享变量的访问，并识别、记录下程序运行过程中出现的访问模式。第二步，需要对第一步得到的数据进行统计和分析，结合测试用例的执行结果（通过或失效），得到每一个访问模式的可疑度，并按照可疑度的大小从高到低排序。

和以往的错误定位工具的工作原理相比，Falcon 有两点不同：

1. Falcon 多次执行同一个测试用例，而不是执行多个测试用例；
2. Falcon 并不追求覆盖更多的代码，而是尝试记录数据访问模式。

1.4.1 实例

在具体介绍 Falcon 算法之前，本节先给出一段伪代码程序。然后本文将在 1.4.2和 1.4.3两节再使用这个具体的实例来演示 Falcon 算法，依次介绍 Falcon 的两个主要步骤。

下面的伪代码描述的是一段包含有三个线程，两个共享变量的程序。

代码 1: 示例伪代码

```
x=0; y=0;
Thread1
1: if(x==0) x=1;
2: if(y==0) y=1;
3: if(x==2 and y==2) assert(false);
Thread2
4: if(x==1) x=2;
5: if(y==1) y=2;
Thread3
6: if(x==1) x=3;
7: if(y==1) y=3;
```

由于多线程程序的运行结果并不确定，本文假设 4 种可能出现的执行语句序列及相应的对变量 x 和变量 y 的访问序列，如表格 1-3 所示。

表 1-3: 4 种可能的运行结果

	执行语句序列	对 x 的访问序列	对 y 的访问序列
1	1-6-4-2-7-5-3	$R_{1,1}-W_{1,1}-R_{3,6}-W_{3,6}-R_{2,4}-R_{1,3}$	$R_{1,2}-W_{1,2}-R_{3,7}-W_{3,7}-R_{2,5}-R_{1,3}$
2	1-6-4-2-5-7-3	$R_{1,1}-W_{1,1}-R_{3,6}-W_{3,6}-R_{2,4}-R_{1,3}$	$R_{1,2}-W_{1,2}-R_{2,5}-W_{2,5}-R_{3,7}-R_{1,3}$
3	1-4-6-2-7-5-3	$R_{1,1}-W_{1,1}-R_{2,4}-W_{2,4}-R_{3,6}-R_{1,3}$	$R_{1,2}-W_{1,2}-R_{3,7}-W_{3,7}-R_{2,5}-R_{1,3}$
4	1-4-6-2-5-7-3	$R_{1,1}-W_{1,1}-R_{2,4}-W_{2,4}-R_{3,6}-R_{1,3}$	$R_{1,2}-W_{1,2}-R_{2,5}-W_{2,5}-R_{3,7}-R_{1,3}$

1.4.2 访问模式识别

Falcon 的第一步就是要获取和识别程序执行过程中出现的数据访问模式，具体的算法如算法 1所示。当被测程序运行的时候，Falcon 使用一个固定大小的滑动窗口来识别模式，因而需要解决滑动窗口的更新和实时模式识别这两个问题。

滑动窗口的更新

初始时，窗口是空的，因此，窗口将会存入第一个线程访问变量的记录。如果新出现的记录和之前的记录属于不同的线程（即一个线程逃逸访问），新记录则被存入在窗口的空位中。然后继续读入记录，如果新出现的记录和之前的记录属于同一个线程（即一个线程本地访问），新的记录则替换原有的记录。需要注意的是：因为线程的写操作更容易产生错误（在顺序破坏和原子性破坏中都至少有一个写操作），所以若新的记录是读操作，则不替换原有写操作的记录。

对于 1.4.1给出的实例，取表 1-3中第一个运行情况进行讨论，对变量 x 的访问序列是 $R_{1,1}-W_{1,1}-R_{3,6}-W_{3,6}-R_{2,4}-R_{1,3}$。滑动窗口如图 1-1

算法 1: GatherPatterns

输入: m:shared memory location
b:memory access type
t:thread ID
s:meomory access location
P_t:current set of patterns(initially null)
输出: P_t:updated set of patterns

```
if m does not yet have any window then
    w←createWindow()
    w.insert(b, t, s)
    registerWindow(w, m)
else
    w←getWindow(m)
    (b2, t2, s2)← w.getLastAccess()
    if t = t2 then
        w.update(b, s)
    else
        if w is full then
            Pt ←getPatterns(w)
            w ←slideWindow(w)
        end
        w.insert(b, t, s)
    end
end
return Pt
```

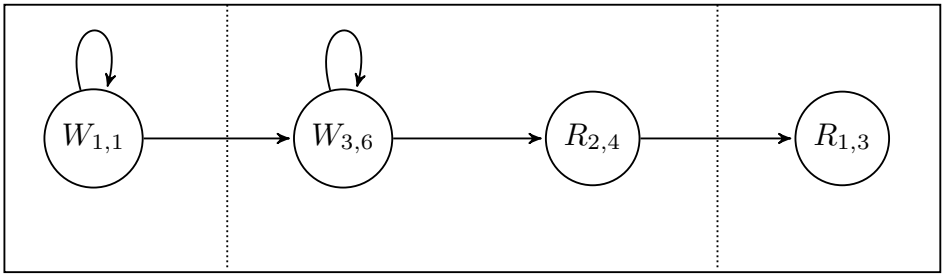

图 1-1: 滑动窗口示例

所示。开始是线程 1 连续两次对变量操作 $R_{1,1} - W_{1,1}$。滑动窗口首先读入线程 1 的读操作 R_1，接着又读入了线程 1 的写操作 W_1，因为两次访问都是同一个线程，所以滑动窗口向前滑动，只记录 W_1。同理，窗口内又读入了 W_3 和 R_2。当滑动窗口已满的时候，则移除最早的记录。对于图所示的滑动窗口，窗口大小是 3，当读入线程 2 的操作 $R_{2,4}$ 时，窗口已满，则窗口滑动，移除最早的记录 $W_{1,1}$。

滑动窗口的容量是有限的，如果窗口过小，可能在没有判断出模式的情况下就移除了早先的记录。如果窗口过大，又会占用过多的系统资源，从而造成 Falcon 的运行效率低下。所以，选择合适的滑动窗口大小也是一个复杂的问题。在 Falcon 论文中，作者通过试验发现当窗口大小为 9 的时

候就足以识别所有的模式 [6]。

实时模式识别

当滑动窗口的内容更新时，如果检测到滑动窗口已满，则扫描窗口里的记录来识别数据访问模式。识别访问模式时，首先检查是否有表格1-2中所列的模式。如果没有，再检查是否有表格 1-1中所列的模式。因此，Falcon 如果已经识别到了原子性破坏的访问模式，就不会重复识别顺序破坏的模式。

1.4.3 模式的可疑度排序

Falcon 的第二步是通过统计分析得到每个模式的可疑度。执行第二步时，需要第一步得到的数据访问模式和被测程序的运行结果。

在 Falcon 中，作者使用了 Jaccard 系数来衡量一个模式的可疑度。Jaccard 系数，也被操作 Jaccard 相似系数，在统计学中用来被衡量两个样本集的相似性。对于集合 A 和 B，Jaccard 相似系数为：

$$J(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|} \tag{1}$$

在 Falcon 工具中，可以通过衡量一个数据访问模式的通过测试集和失效测试集之间的相似性作为该模式的可疑度。对于一个模式 p，有通过的执行数 $pass(p)$，失效的执行数 $failed(p)$ 和该被测程序所有测试失效的个数 $totalfalied$，将公式 1展开可以得到 p 的可疑度：

$$suspiciousness(p) = \frac{failed(p)}{totalfailed + passed(p)} \tag{2}$$

对于 1.4.1中表格 1-3 给出的实例和四个数据访问模式，假设程序运行 4 次后的运行结果和每次运行出现的数据访问模式如表格 1-4所示。 对于

表 1-4: 示例伪代码的假设运行结果

简化的访问模式	运行 1	运行 2	运行 3	运行 4	可疑度
x 的访问模式:$W_1 - W_3 - R_1$	✓	✓			0
x 的访问模式:$W_1 - W_2 - R_1$			✓	✓	0.5
y 的访问模式:$W_1 - W_3 - R_1$	✓		✓		0
y 的访问模式:$W_1 - W_2 - R_1$		✓		✓	0.5
运行结果	Pass	Pass	Pass	Failed	

第一个模式，根据公式 2，可以得到它的可疑度是

$$suspiciousness(p) = \frac{failed(p)}{totalfailed + passed(p)} = \frac{0}{1+2} = 0$$

对于第二个模式，根据公式 2，可以得到它的可疑度是

$$suspiciousness(p) = \frac{failed(p)}{totalfailed + passed(p)} = \frac{1}{1+1} = 0.5$$

同理，可以依次计算得到表格 1-4中最后一列所示的可疑度。

1.5 本课题的目标

本文将对指导老师所提供的 Falcon 源码（部分实现的版本）进行分析、补充和重构，总结错误定位工具的实现技术。进而实现 Falcon 工具的可视化，从而可以更加直观地展现出程序中的错误位置。

在本篇论文的第 2 章将介绍 Falcon 的原理和算法实现，第 3 章将简要介绍可视化的实现，第 4 章总结本次毕业设计。

2 Falcon 工具的技术分析与重构

2.1 Falcon 框架

在图 2-1中显示的是 Falcon 文件组织框架。在 src/目录下是源代码，而 src-dummy/，目录下是需要插装的方法的签名[2]。具体到 src/目录里，

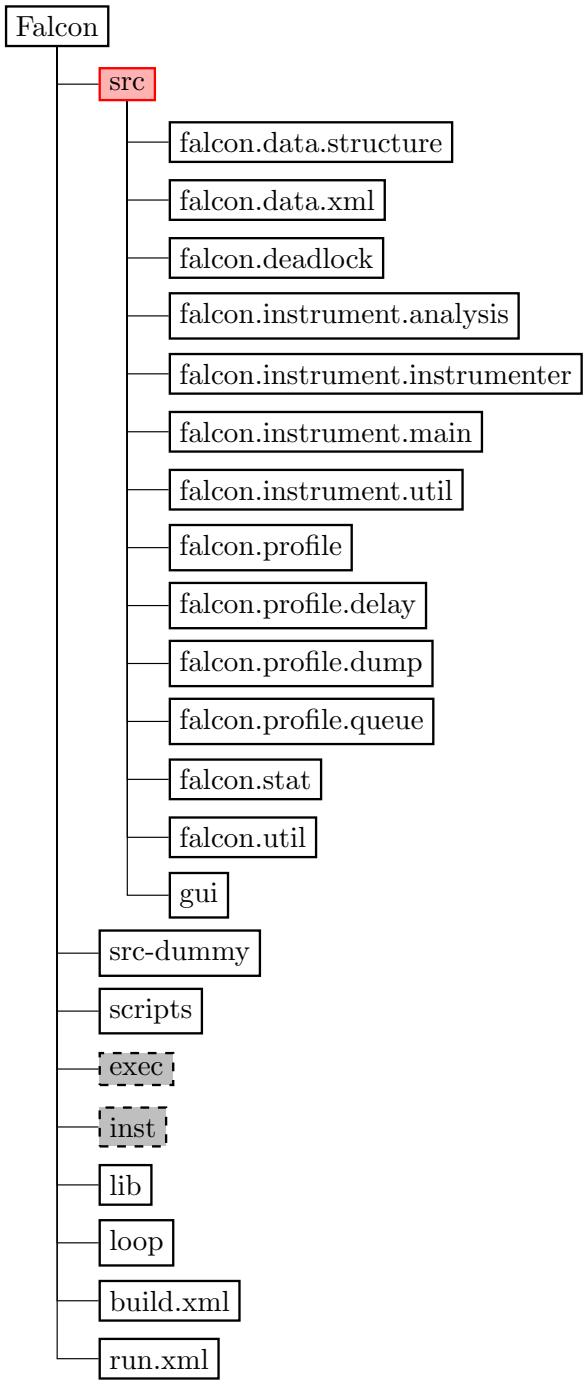

图 2-1: Falcon 文件组织框架

[2]方法的签名是方法声明的一部分，包括方法名和参数列表。

falcon.data.structure 里是 Falcon 中用到的数据结构。falcon.data.xml 用来实现序列化和反序列化。falcon.deadlock 里的代码用于检测被测程序运行时是否死锁，如果死锁，execInfo 里记录的运行结果将是 Deadlock。falcon.instrument 里的代码用于实现插装。falcon.profile 里的代码主要是探针，用来插入到被测程序当中。falcon.stat 里的代码用于统计结果，得到每一个模式的可疑度。falcon.util 里的代码是 Falcon 中常用的数据类型。gui 中的代码用于实现可视化。

scripts/下是 Python 脚本，用来自动化插装、运行多个被测程序。lib/下存放有 Falcon 中使用的类库。loop/有 1.loop，10.loop，100.loop 等文件，用来控制被测程序重复执行。build.xml 和 run.xml 分别是使用 ant 进行编译和运行被测程序时的脚本。除了代码之外，还有 inst 和 exec 两个文件夹是运行 Falcon 过程中生成的，用来保存被测程序的插装结果和运行结果。

2.2 被测程序的插装技术

程序插装是指使被测试程序在保持原有逻辑完整性基础上在程序中插入一些探针，通过探针的执行并抛出程序的运行特征数据。基于这些特征数据分析，可以获得程序的控制流及数据流信息，进而得到逻辑覆盖等动态信息 [7]。

实现插装时，Falcon 使用 Soot 对被测程序进行静态的线程逃逸分析，从而确定一个变量是否被多个线程共享访问。如果一个变量是被多个线程访问，则插装该变量的读写操作，这样在程序运行时就可以得到访问共享变量的记录。如代码 2所示，对于一个 Java 语句 “x = y + 1;”，在其前后分别进行插装探针，可以获得读变量 y 和写变量 x 的信息。

代码 2: 插装程序读写操作示意

```
...
accessObj(method, Stmt(x=y+1), Var(y), read);
x = y + 1;
accessObj(method, Stmt(x=y+1), Var(x), write);
...
```

2.2.1 线程逃逸分析

在多线程程序当中，线程本地数据（thread local data）是指被一个线程拥有的数据。这里的“拥有”的意思是这些数据只能被这个线程访问 [8]。与之相对的是可以被多个线程访问的线程共享数据（thread shared data）。多个线程访问同一个数据可能会带来数据争用或死锁等问题。线程逃逸分析（thread escape analysis）的作用是监视所有对线程本地数据的访问，当检测到有另外一个线程尝试访问这个线程本地数据，即标记这个数据为逃逸（escaping）[9]。在 Falcon 中，首先需要通过线程逃逸分析判断一个变量是否被多个线程访问。

2.2.2 Soot 简介

Soot 是加拿大 McGill 大学 Sable 研究组开发的 Java 字节码分析和优化开源工具。它为字节码的分析、优化、反编译、注解等提供一个可扩展的框架 [10]。Soot 作为字节码分析和转换工具，广泛地应用于编译器优化研究，程序分析等领域中。

Soot 在 Falcon 工具中主要有三个作用。一是对被测程序进行静态的线程逃逸分析，用来确定一个变量是否被多个线程访问。在 Soot 的 soot.jimple.toolkits.thread.ThreadLocalObjectsAnalysis 包中提供了相应的 API 可以用来进行线程逃逸分析 [11]。二是使用 Soot 分析 Java 语句来判断线程对变量的访问是读还是写。三是使用 Soot 提供的 API 将探针插装到被测程序当中。

2.2.3 插装的过程

Soot 的分析与变化是建立在其内部的中间表示法（Intermediate Representation，IR）上 [12]。在 Falcon 中，使用的 IR 是 Jimple—Soot 中最核心的 IR。在 Soot 里可以按照需要对 Jimple 进行分析、变换和优化，并生成相应.class 文件。在 Falcon 进行插装时，需要运行 falcon.instrument.main.Main.java。该类继承 SceneTransformer 类，重写了 `internalTransformer()`方法，可以将探针插装到 Jimple 中间表示当中，最后用插装后的 Jimple 生成完成插装的.class 文件。

插装主方法

为了便于 Falcon 判断进行插装的位置，首先需要对被测程序的

main()方法进行修改成如下代码 3所示。

代码 3: 未插装的被测程序主方法

```
public static void main(String[] args){
    runOneTest(args);
}
```

即将原来的 main()方法改名为 runOneTest()，然后重新写一个 main()方法，并调用 runOneTest()方法。经过修改后，Falcon 就可以按照下面代码 4所示完成对 main()方法的插装。其中 startTestCase()用来滑动窗口的初始化，endTestCase()用来返回被测程序的运行结果。

代码 4: 插装后的被测程序主方法

```
public static void main(String[] args){
    startTestCase();
    runOneTest(args);
    finishTestCase();
}
```

这里，还需要考虑的一个特殊情况就是带有静态类变量（带有 static 标识符）的主类。在 Java 语言中，静态变量在类创建的时候即在 Java 虚拟机中完成分配内存、赋值等工作。这个时候如果仍然按照代码 4所示的方法进行插装，就无法对静态变量进行分析。在 Jimple 当中，静态变量、静态方法的初始化是在 <clinit>()的方法内进行，因此需要将 startTestCase()插装到 <clinit>()之前，如代码 5所示。

代码 5: 插装后的带有静态变量的被测程序主方法

```
public static void main(String[] args){
    startTestCase();
    <clinit>()
    runOneTest(args);
    finishTestCase();
}
```

插装主方法的过程中另外一个难点是在插装的 endTestCase()中需要得到被测程序的运行结果作为参数传递给 Falcon，但是被测程序的结果需要在运行结束后才能获得。这样就导致在 Falcon 运行时无法得到被测程序的结果，生成的 execInfo.xml 也存在错误。本文作者的解决方法是统一在插装 endTestCase()时把被测程序的运行结果设置为通过，这样生成的

execInfo.xml 里记录的运行结果全部都是通过。然后使用 Python 脚本，读取被测程序运行日志文件里的结果来纠正 execInfo.xml 里的运行结果。

插装方法

对方法的插装在 Falcon 中是可选的操作。插装方法之后，可以获得被调用函数的信息。插装的过程是在每个方法的调用点和返回点插入探针，当程序运行时，每次调用一个方法，就会带出该方法的相关信息。

插装变量

在 Falcon 当中，对于变量的插装首先需要对 Java 中的语句进行分析，找出包含有对变量操作的语句。因此需要对 Java 字节码当中会出现所有的语句（Soot 中 Jimple 里的 Stmt 类型）进行分情况讨论。虽然 Jimple 里语句共有 15 种，但除去调用语句、return 语句、break 语句等大量不涉及变量读写操作的语句外，需要进行处理的语句只有 4 种，分别是 If 语句 IfStmt、Switch 语句 TableSwitchStmt(对应于 JVM 里的 tableswitch 指令) 和 LookupSwitchStmt(对应于 JVM 里的 lookupswitch 指令)、赋值语句 AssignStmt。代码片段如代码 6所示。

代码 6: 对不同的 Stmt 进行处理

```
...
if (StmtUtil.isReturnStmt(stmt)||stmt instanceof IdentityStmt||
    stmt instanceof EnterMonitorStmt||stmt instanceof NopStmt||
    stmt instanceof GotoStmt||stmt instanceof ExitMonitorStmt||
    stmt instanceof BreakpointStmt||stmt instanceof ThrowStmt){
    // do nothing
} else if (stmt instanceof IfStmt) {
    //instrument if-stmt
    Value value = ((IfStmt) stmt).getCondition();
    instValue(stmt, value, true);
} else if (stmt instanceof TableSwitchStmt){
    //instrument table switch
    Value value = ((TableSwitchStmt) stmt).getKey();
    instValue(stmt, value, true);
} else if (stmt instanceof LookupSwitchStmt){
    //instrument lookup switch
    Value value = ((LookupSwitchStmt) stmt).getKey();
    instValue(stmt, value, true);
} else if (stmt instanceof AssignStmt) {
    instAssignStmt((AssignStmt) stmt);
}
...
```

找出了含有变量读写的语句之后，还需要要分析变量的读写操作和类型。对变量的读写操作可以通过语句的语义来进行判断。例如对于赋值语句“x=y”，可以知道赋值语句的右部，即变量 y 是读操作，而赋值语句的左部，即变量 x 是写操作。又如 If 语句和 Switch 语句里对条件的判断就是读操作。对变量的类型（基本数据类型或字段），可以使用 Soot 提供的 API 进行判断，在 Falcon 中，`instValue()`方法（如代码 7所示）用来完成该任务。分析出变量的读写操作和类型之后，就可以将探针插装到程序中。

代码 7: instValue() 方法片段

```
private void instValue(Stmt stmt, Value value, boolean isRead){
  if (value instanceof StaticFieldRef) {
      instStaticFieldRef(stmt, value, isRead);
  } else if (value instanceof InstanceFieldRef) {
      instInstFieldRef(stmt, value, isRead);
  } else if (value instanceof InvokeExpr) {
      instInvokeExpr(stmt, value);
  } else if (value instanceof Local) {
      instLocal(stmt, value, isRead);
  } else if (value instanceof BinopExpr) {
      BinopExpr expr = (BinopExpr) value;
      instValue(stmt, expr.getOp1(), true);
      instValue(stmt, expr.getOp2(), true);
  } else if (value instanceof UnopExpr) {
      UnopExpr expr = (UnopExpr) value;
      instValue(stmt, expr.getOp(), true);
  } else {
      // print no-instrumented value
      ...
  }
}
```

关于插装被测程序，被分析的原有代码缺少了对 If 语句和 Switch 语句这两个情况的考虑。原有代码也没有处理带有静态类变量的主类。本文作者补充了代码从而解决了上述两个不足。

2.3 串行化为 XML 和反序列化为对象

在 Falcon 运行的过程中，需要将运行中间的插装信息、被测程序的运行结果和输出结果进行序列化成 XML 文件进行保存。在分析、统计的过

程中，也需要将 XML 文件里的结果反序列化成内存中的对象进行处理。

2.3.1 XStream 简介

XStream 是由 codehaus 项目组开发的 Java 类库，用来将对象序列化成 XML（JSON）或反序列化为对象 [13]。使用 XStream 不用任何映射就能实现多数 Java 对象的序列化。在生成的 XML 中对象名变成了元素名，类中的字符串组成了 XML 中的元素内容。使用 XStream 序列化的类不需要实现 Serializable 接口。XStream 是一种序列化工具而不是数据绑定工具，就是说不能从 XML 或者 XML Schema Definition (XSD) 文件生成类。和其他序列化工具相比，XStream 有三个突出的特点：

1. XStream 不关心序列化/逆序列化的类的字段的可见性。
2. 序列化/逆序列化类的字段不需要 getter 和 setter 方法。
3. 序列化/逆序列化的类不需要有默认构造函数。
4. 不需要修改类，使用 XStream 就能直接序列化/逆序列化任何第三方类 [14]。

2.3.2 串行化和反序列化的过程

在 falcon.data.structure 包中共有三个类用来保存程序运行的中间信息，分别是保存插装信息的 InstInfo，被测程序运行信息的 ExecInfo 和 Falcon 结果的 SummaryInfo。这三个类需要实现序列化和反序列化。在 falcon.data.xml 包中，UML 类图如图 2-2所示。首先创建 XMLHandler 接口，包含有 `Serialize(Info, path)`和 `deSerialize(path)`两个方法。再分别创建 XMLInstInfo、XMLExecInfo 和 XMLSummaryInfo 三个类来实现 XMLHandler 接口。

InstInfo 类包括被调用的方法列表和程序中出现的变量构成的列表。经过序列化之后生成的 instInfo.xml 的片段如代码 8所示，XML 元素的解释见表 2-5。

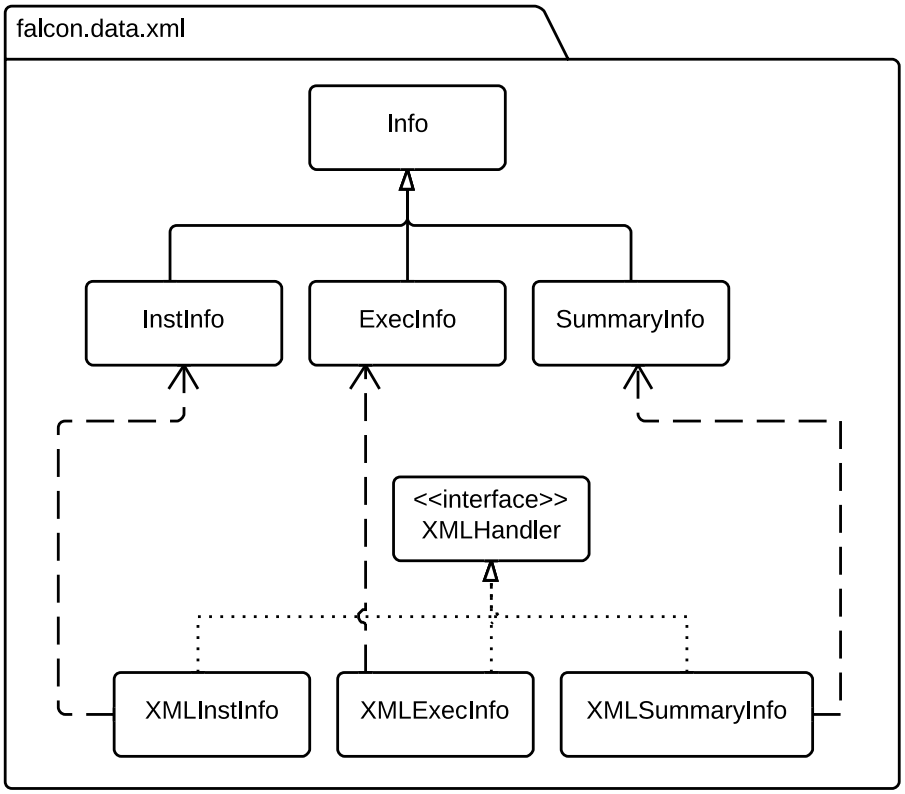

图 2-2: falcon.data.xml 包 UML 类图

代码 8: instInfo.xml 片段

```
<methodInfo>
 <mid>4</mid>
 <msig>&lt;contest.account.Account: void transfer(...)&gt;</msig>
</methodInfo>
<variableInfo>
 <vid>6</vid>
 <vtype>WF</vtype>
 <vsig>this.&lt;contest.account.Account:double amount&gt;</vsig>
 <vfile>Account.java</vfile>
 <vline>24</vline>
</variableInfo>
```

ExecInfo 类包含有该次程序运行的结果和出现的数据访问模式。经过序列化之后生成的 execInfo.xml。SummaryInfo 类包含有多次运行结果的统计信息和各个数据访问模式的信息、可疑度。其中，顺序破坏和原子性破坏这两种数据访问模式是最主要的信息。经过序列化后，这两部分信息会如代码 9所示保存成 XML 文档，其中 XML 元素的解释见表 2-6。

表 2-5: instInfo.xml 元素介绍

XML 文档元素	解释
<mid>4</mid>	方法的编号为 4
<msig>...</msig>	方法的签名
<vid>6</vid>	变量的编号是 6
<vtype>WF</vtype>	变量的操作类型是写类属性
<vsig>...</vsig>	变量的签名
<vfile>Account.java</vfile>	变量在 Account.java 文件中
<vline>24</vline>	变量出现在第 24 行

代码 9: execInfo.xml 片段

```
<order>
    <pass>9</pass>
    <fail>1</fail>
    <susp>10</susp>
    <stack1></stack1>
    <stack2></stack2>
    <var1>6</var1>
    <var2>8</var2>
</order>
<atomicity>
    <pass>0</pass>
    <fail>1</fail>
    <susp>100</susp>
    <stack1></stack1>
    <stack2></stack2>
    <var1>4</var1>
    <var2>2</var2>
    <stack3></stack3>
    <var3>27</var3>
</atomicity>
```

表 2-6: 数据访问模式序列化为 XML

XML 文档元素	解释
<pass>9</pass>	被测程序执行了 10 次，其中 9 次得到正确结果
<fail>1</fail>	被测程序有一次运行结果错误
<susp>10</susp>	可疑度是 10%
<var1>6</var1>	模式的第一个操作的变量编号是 6
<stack1></stack1>	方法调用序列

2.4 改进

由于被分析的源码并不完整，需要对代码进行补充才能够运行。另外，本文以 Martin Fowler 的《重构——改善既有代码的设计》一书为指导，对源代码的部分实现过程进行了重构。虽然重构的操作比较琐碎，但是提高了代码的可读性，一些代码做到了复用。

2.4.1 对源代码的补充

被分析的源代码由于不完整，因此不能够对 `startTestCase()` 和 `endTestCase()`进行插装。缺少的这一部分代码涉及到了滑动窗口的初始化和返回被测程序运行结果等关键步骤，所以必须补充完整才能使 Falcon 运行。本文作者补充了这一部分的代码，使 Falcon 可以运行。此外，本文作者对 Falcon 原有算法的完善主要包括两点：

1. 增加了对 If 语句和 Switch 语句的分析；
2. 增加了对含静态变量的类的处理。

关于以上两点的具体实现，本文作者所做的工作在 2.2.3中已做了具体介绍和解释。

2.4.2 对 Falcon 的重构

本次毕业设计中对原有代码的重构包括以下几点：

1. 封装字段

falcon/data/structure 中部分类中 public 字段改为 private，并且加入相应的访问函数。

2. 重构部分方法

- 提炼函数，将过长的代码段放入独立的函数中，如添加 ExecInfo 类, SummaryInfo 类中 `print()`方法等；
- 以直接访问类属性取代方法调用，修改 InstMethod 类和 InstVariable 类的 toString 方法，减少方法调用次数，提高效率。

3. 使用 Enum 类型

- 添加 ExecuteResult 枚举类，替换之前使用 int 类型表示运行结果；

- 重写 EventType 枚举类，添加 `toString()`方法，删去 `getTypeString()`方法。

4. 提炼接口
将用于序列化和反序列化的 `Serialize()`和 `deSerialize()`提炼到一个独立的接口中。
5. 提炼超类
将 ExecInfo、InstInfo 和 SummaryInfo 三个类中相同的属性和方法移至超类 Info 中。
6. 修改部分代码，使满足编程规范。

2.5 运行环境与配置

2.5.1 运行环境

硬件

- CPU：Intel Core 2 Duo T6500, 2.1GHz
- 内存：2G

软件

- Java 运行环境：Java SE 1.7
- Python 运行环境：Python3.3
- 开发环境：Eclipse 4.2
- 构建工具：Apache Ant 1.8.4
- 基准程序: The ConTest Benchmark Suite
- Soot 2.4.0[3]

Apache Ant 是一个将 Java 编译、测试、部署等步骤联系在一起加以自动化的一个工具，脚本格式为 XML。Ant 和 C 语言使用的 make 脚本一样，可以用于自动化调用程序完成项目的编译，打包，测试等。Python 是一门面向对象的解释型编程语言。在 Falcon 当中，使用 Python 脚本用来调用 Ant 的 build 文件，便于自动化运行多个被测程序。The ConTest Benchmark Suite 由一组包含有并发错误的 Java 多线程小程序（代码行均

[3]目前 Soot 的最新版本是 2.5.0，但是使用该版本有兼容性问题。

小于 1000 行）组成的基准程序，可以用来检验多线程测试工具的有效性[15]。ConTest Benchmark 里的程序都给出了具体的场景，即预期的输出结果，这样便于判断程序的实际输出结果是否正确。程序中的错误在文档里都进行了详细的分析和说明，这样便于和错误定位工具得到的结果进行比对，从而可以检测工具的有效性。

2.5.2 配置过程

1. 安装 Eclipse 插件
   - Python 插件 PyDev[4]
2. 创建工作目录（Falcon_Root）
3. 创建子目录
   - Falcon_Root/falcon，用来存放 Falcon 代码
   - Falcon_Root/subjects，用来存放基准程序
   - Falcon_Root/library，用来存放类库
4. 添加需要的类库至 Falcon_Root/library
   - sootclasses.jar
   - jasminclasses.jar
   - java_cup.jar
   - polyglot.jar[5]
   - xpp3_min.jar[6]
   - xstream.jar[7]
5. 导入项目代码，配置类库路径

2.5.3 运行

Falcon 运行的流程如图 2-3所示。

[4]http://pydev.org/download.html
[5]以上四个 jar 包均可以在 http://www.sable.mcgill.ca/soot/soot_download.html 下载。
[6]http://mvnrepository.com/artifact/xpp3/xpp3_min
[7]http://xstream.codehaus.org/download.html

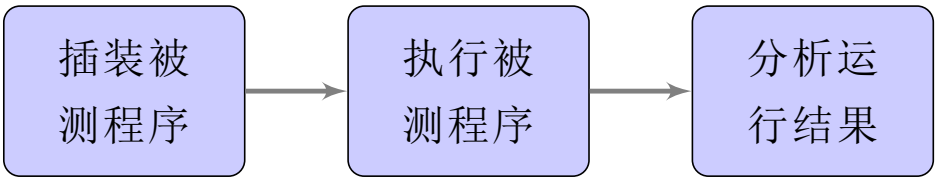

图 2-3: Falcon 运行流程图

在 Eclipse 上运行，具体包括一下几个步骤：

1. 编译

编译 Falcon 和 Benchmark。在 Eclipse 中，可以直接右键 build.xml 选择执行 ant。

2. 插装

运行 inst.py，调用 build.xml 中的 inst 任务进行插装。在安装过 PyDev 的 Eclipse 中，可以直接运行 python 脚本。

3. 运行

运行 exec.py，调用 run.xml 运行被测程序。

4. 纠正运行结果

运行 result.py，读取日志文件纠正 execInfo.xml 中错误的运行结果。

5. 输出排序结果

运行 output.py，调用 run.xml 得到统计结果。

2.6 运行结果

插装结果

第一步插装之后，Falcon 会生成记录静态分析被测程序过程的日志文件和保存有插装结果的 instInfo.xml，同时，还会得到经过插装的.class 文件。

运行结果

第二部运行经过插装的被测程序（第一步生成的经过插装的.class 文件），每运行一次被测程序，会生成带有运行结果的日志文件（.log）和保存有每一个数据访问模式的 execInfo.xml。

输出结果

最后一步之后，Falcon 会生成两个文件来保存结果，分别是序列化后的 summaryInfo.xml 和显示结果的 rank.txt。其中 rank.txt 的输出结果如下所示。

```
Result
Total Pass: 3
Total Fail: 7
Total Deadlock: 0
printing order violation
| rank | pass | fail | deadlock | access1 | access2 | score |
| 1 | 3 | 7 | 0 | WF @Account.java:25 | WF @Account.java:26 | 70.0% |
| 2 | 0 | 3 | 0 | RF @Account.java:31 | WF @Account.java:21 | 42.0% |
| 3 | 0 | 3 | 0 | WF @Account.java:26 | RF @Main.java:73 | 42.0% |
...
printing atomicity violation
| rank | pass | fail | deadlock | access1 | access2 | access3 | score |
| 1 | 3 | 7 | 0 | RF @Account.java:31 | WF @Account.java:25 | RF @Main.java:73 | 70.0% |
| 2 | 3 | 7 | 0 | RF @Account.java:31 | WF @Account.java:26 | RF @Main.java:73 | 70.0% |
| 3 | 0 | 4 | 0 | RF @Account.java:26 | WF @Account.java:25 | WF @Account.java:26 | 57.0% |
| 4 | 1 | 4 | 0 | WF @Account.java:21 | WF @Account.java:26 | WF @Account.java:25 | 50.0% |
| 5 | 0 | 3 | 0 | RF @Account.java:31 | WF @Account.java:25 | RF @Main.java:71 | 42.0% |
| 6 | 0 | 3 | 0 | RF @Account.java:31 | WF @Account.java:26 | RF @Main.java:71 | 42.0% |
...
```

示例的程序是 The ConTest Benchmark Suite 中的一个小程序。经过插装后共运行了 10 遍。首先 Result 显示了这 10 次运行，一共有 3 次通过，剩下 7 次得到的结果都与预期不符，没有出现死锁。接着显示的是按照可疑度由高到低排列的顺序破坏数据访问模式。其中，排名第 1 的模式是 $W_1 - W_2$ 型的顺序破坏。变量所在的位置分别是 Account.java 的第 25 行和 Account.java 的第 26 行。在 10 次运行中，有 3 次结果正确，7 次结果错误。按照公式 2可以计算出可疑度是 70%。顺序破坏数据访问模式过后，显示的是按照可疑度由高到低排列的原子性破坏数据访问模式，输出信息的具体含义与顺序破坏的相同。

3 Falcon 工具的可视化

3.1 可视化的目标

原始版的 Falcon 工具虽然可以具体定位到每一个数据访问模式的变量名和所在的语句，但由于结果只能以纯文字的格式显示，仍然不便于程序员迅速发现和定位错误。所以本文作者希望能够通过高亮显示的方式来突出包含错误模式的语句。同时希望将选择被测程序、插装、运行被测程序等一系列操作统一到一个图形化界面的工具里。

3.2 可视化的实现

3.2.1 MVC 设计模式

MVC 模式（Model-View-Controller）是软件工程中的一种软件架构模式。MVC 是以模型（Model）、视图（View）和控制器（Controller）三个单词的首字母缩写命名。使用 MVC 模式可以减少代码之间的耦合。MVC 模式的三个模块相互独立，改变其中一个不会影响其他两个，从而提高了应用程序的灵活性和可配置性。

模型用来封装应用程序的业务逻辑和基础数据。模型对外提供接口，可以被控制器和视图调用。视图是应用程序与用户的接口，作用是负责显示，即表达逻辑的内容。视图是模型的外观，可以访问模型的数据，但不能改变这些数据。视图只需要知道模型提供的接口，而不需要了解模型的内部逻辑。控制器是模型和视图之间的桥梁。控制器的作用是接受视图请求，并做出反应，执行相应的控制流，或者把响应结果返回到视图 [16]。

在 Falcon 可视化工具的实现过程中使用了 MVC 设计模式，这一部分的 UML 图如图 3-1所示。

3.2.2 SWT 框架

SWT(Standard Widget Toolkit，标准部件工具包) 是在 Java 平台下的一个图形化部件工具包。SWT 最初由 IBM 公司主导开发，现在作为 Eclipse 集成开发环境的一部分由 Eclipse 基金会维护 [17]。

和 Java AWT、Java Swing 等其它相比，SWT 主要有以下优势：

1. 界面与本地操作系统对应；

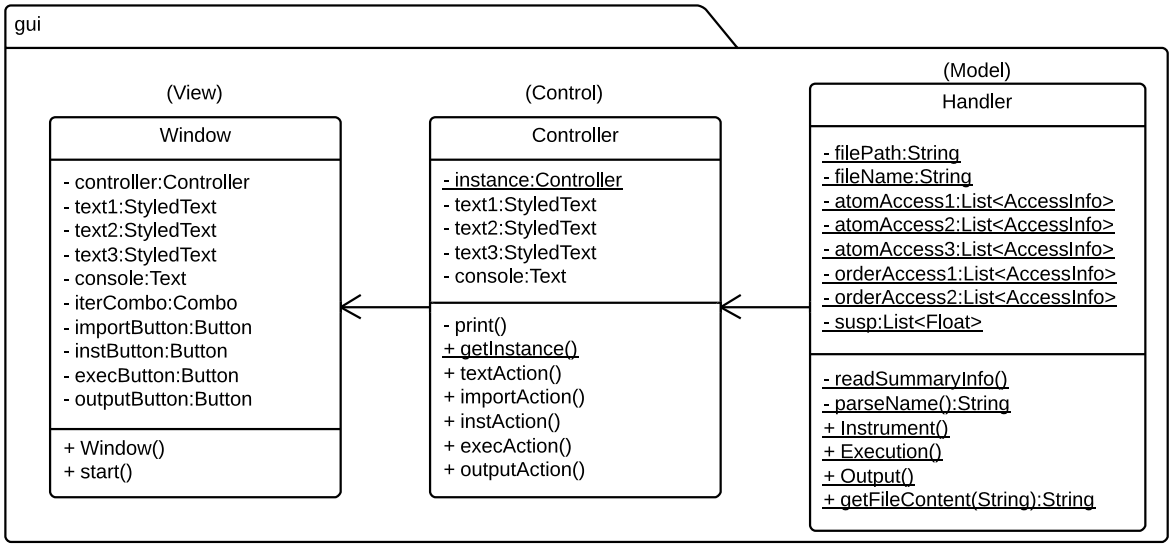

图 3-1: MVC 设计模式的 UML 类图

2. 简单实用的 API 可以是开发人员快速上手；

3. SWT 应用程序运行速度快 [18]。

在 Falcon 的可视化工具中，如何使含有错误的代码高亮显示是一个难点。在 SWT 中，提供了 StyledText 部件可以用来显示带有格式的文本，实现诸如加粗、背景色等显示效果。在窗口中添加 StyledText 对象，并调用 `addLineBackgroundListener()`方法，实现 LineBackgroundListener 接口中的 `lineGetBackground()`方法，即设定需要高亮显示的行数 *line*和背景的样式。当 StyledText 对象中加入了文字之后，就可以触发 LineBackgroundEvent，LineBackgroundListener 监听到事件后，就把第 *line* 行按照之前的设置高亮显示出来。

3.3 可视化的效果

Falcon 的可视化工具效果如截图 3-2所示。在使用时，首先需要导入被测程序。点击菜单栏上的 Import 按钮，选择被测程序中包含有主方法的类。如图 3-3所示。点击 Instrument 按钮可以进行插装。在 Run Times 里可以选择被测程序执行的次数，然后点击 Execution 按钮可以运行被测程序。点击 Output 按钮之后，文本框中会出现被测程序的代码。包含有错误的访问模式所在的代码行被黄色高亮显示。在右下角的显示框内，会显示这个访问模式的具体信息，包括可疑度，模式类型，变量名等。点击 Next 按钮则可以显示后一个错误的模式，点击 Previous 按钮可以显示前

图 3-2: Falcon 的可视化工具

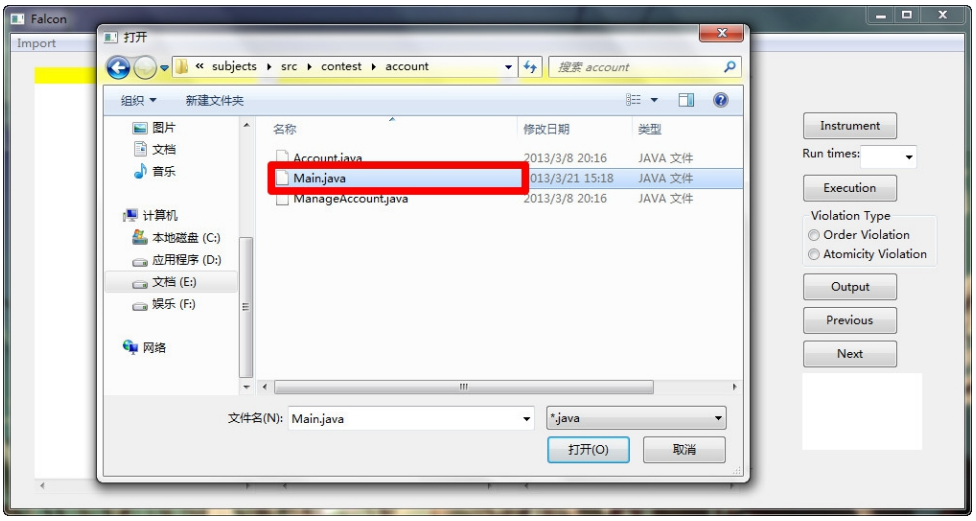

图 3-3: 选择被测程序中包含有主方法的类

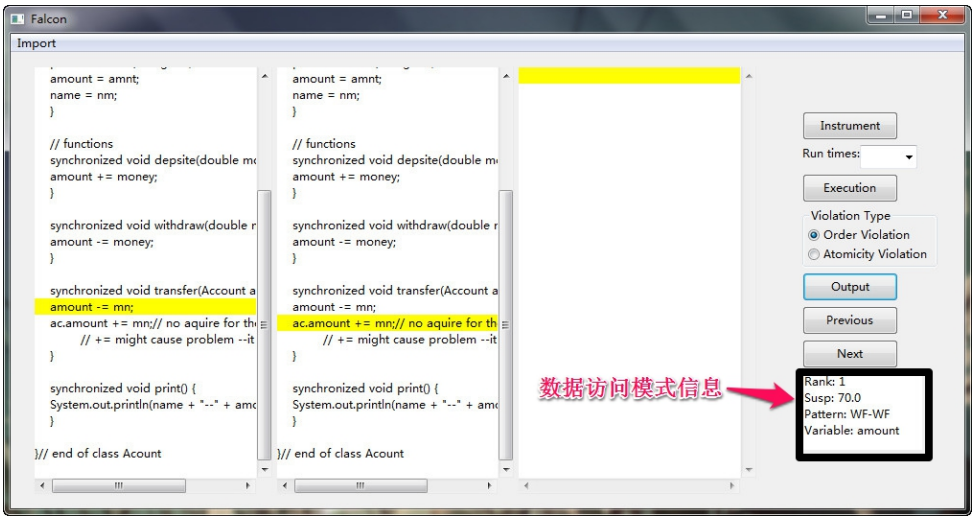

图 3-4: 顺序性破坏显示效果

一个错误的模式。

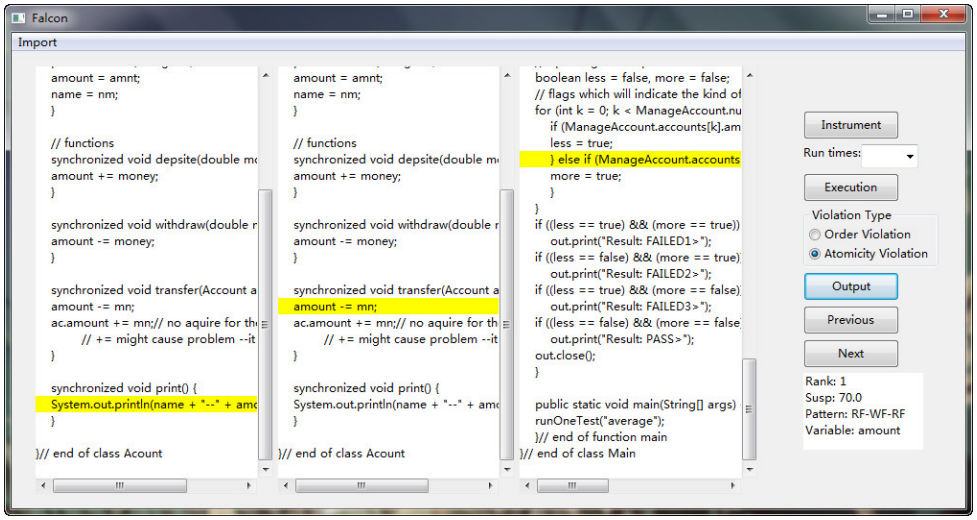

图 3-5: 原子性破坏显示效果

3.4 与 CORE 的对比

在 Sangmin Park 提出了 Falcon 的方法之后，乔治亚理工学院的两个研究生，Deepal Jayasinghe 和 Pengcheng Xiong，也尝试将 Falcon 扩展成为可视化工具。他们最终开发出了名为 CORE 的可视化并发错误定位工具 [19]。

CORE 可以提供两个层次的视图来显示结果。如图 3-6所示，低层次的视图用来显示数据访问模式中变量和方法的详细信息。高层次的视图用来显示模式的的可疑度等统计信息。此外，CORE 提供了更为丰富的信息，包括插装信息、方法调用序列等。与 CORE 相比，本文中的可视化工具有如下特点：

1. 界面更加简洁；
2. 操作更加简单；
3. 将插装、运行被测程序这两个步骤集中到可视化工具当中。

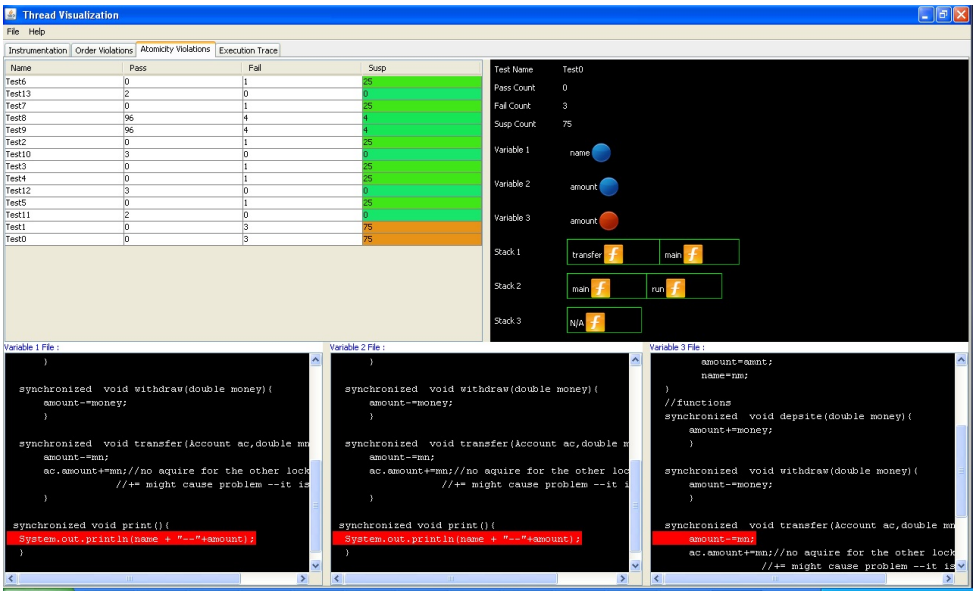

图 3-6: CORE 的显示效果

4 总结与展望

本论文主要完成的工作包括以下几个方面：

1. 分析了多线程程序错误定位工具的价值；
2. 通过实例叙述了 Falcon 算法的过程；
3. 详细描述了 Falcon 原始版本的实现和本文作者重构过程；
4. Falcon 的配置和运行；
5. 详细描述了 Falcon 可视化工具实现的过程和效果。

本文通过上述工作，详细地描述了 Falcon 算法的原理，并且通过对实例的说明，直观地展示了 Falcon 运行过程中重要的步骤。对 Falcon 实现过程中的几个重要部分和可视化的实现过程进行了细致的介绍。此外，在论文里还详细地列出了如何配置、运行 Falcon。

最后，虽然已经做出了一个具有可视化功能的原型，但是仍然有可以改进的地方。例如，界面还不美观；只能显示单个访问模式的信息，不能像 CORE 一样显示一个被测程序总体的运行结果。此外，Falcon 实现之后只在 The ConTest Benchmark 上进行了测试并取得了很好的效果，今后还需要使用更多的程序来验证 Falcon 的健壮性和有效性。这些不足需要在今后的学习中加以完善。

致谢

在此，我要感谢本篇论文的指导教师孙玉霞老师。感谢她对我的悉心指导，并且为我毕业论文的撰写指明方向。同时也要感谢大学四年里教过我的所有任课老师，我能够完成毕业论文离不开他们的辛勤栽培。

感谢 McGill 大学的 Sable 实验室、Codehaus 项目组、PyDev 项目组以及 Eclipse 基金会，正是他们的开源项目为本次毕业论文的实验部分无偿提供了必需的工具。还要感谢 TUG 和 CTeX 社区，他们为本次毕业论文的撰写和排版提供了优秀的工具。

……

谨把本文献给我最敬爱的父亲、母亲以及所有关心我、帮助我的人！

参考文献

[1] MCDOWELL C E, HELMBOLD D P. Debugging concurrent programs[J]. ACM Computing Surveys (CSUR), 1989, 21(4):593–622.

[2] GODEFROID P, NAGAPPAN N. Concurrency at Microsoft–An exploratory survey[C]//CAV Workshop on Exploiting Concurrency Efficiently and Correctly. .[S.l.]: [s.n.] , 2008.

[3] WONG W E, DEBROY V. A survey of software fault localization[J]. University of Texas at Dallas, Tech. Rep. UTDCS-45-09, 2009.

[4] RONSSE M, DE BOSSCHERE K. RecPlay: a fully integrated practical record/replay system[J]. ACM Trans. Comput. Syst., 1999, 17(2):133–152.

[5] LU S, TUCEK J, QIN F, et al. AVIO: detecting atomicity violations via access interleaving invariants[J]. SIGPLAN Not., 2006, 41(11):37–48.

[6] PARK S, VUDUC R W, HARROLD M J. Falcon: fault localization in concurrent programs[C]//Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. .[S.l.]: [s.n.] , 2010:245–254.

[7] 孙昌爱, 金茂忠. 基于程序插装的动态测试技术实现 [J]. 自动化与仪器仪表, 2001, 22(12).

[8] LI XIAOFENG What is thread local data?[OL]. `http://xiao-feng.blogspot.com`.

[9] LI XIAOFENG What is thread escape analysis?[OL]. `http://xiao-feng.blogspot.com`.

[10] 杨航. Soot 工具及其程序分析实例研究 [D]. 广州: 暨南大学, 2011.

[11] HALPERT R L. Static lock allocation[D]. Montreal: McGill University, 2008.

[12] 苟长弘. Soot 框架及 Java 变量分析 [D]. 广州: 暨南大学, 2011.

[13] WIKIPEDIA. XStream[OL]. `http://zh.wikipedia.org/wiki/XStream`.

[14] BANGALORE R. 使用 XStream 把 Java 对象序列化为 XML[OL]. http://www.ibm.com/developerworks/cn/xml/x-xstream/.

[15] EYTANI Y, HAVELUND K, STOLLER S D, et al. Towards a framework and a benchmark for testing tools for multi-threaded programs[J]. Concurrency and Computation: Practice and Experience, 2007, 19(3):267–279.

[16] 李海峰. MVC 模式架构的应用研究 [J]. 小型微型计算机系统, 2013, 1.

[17] WIKIPEDIA. XStream[OL]. http://zh.wikipedia.org/zh-cn/SWT.

[18] 那静. Eclipse SWT/JFace 核心应用 [M]. 北京: 清华大学出版社, 2007.

[19] JAYASINGHE D, PENGCHENG X. 2010 (Jan.).