软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2014,25(3):457−471 [doi: 10.13328/j.cnki.jos.004421]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于事件结构的并发程序可视化调试方法*

伍晓泉[1,2,3], 魏 峻[1,2]

[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 软件研究所 软件工程技术研究开发中心,北京 100190)

[3](中国科学院大学,北京 100049)

通讯作者: 伍晓泉, E-mail: wuxiaoquan07@otcaix.iscas.ac.cn

摘 要: 在多核和并发技术得到广泛应用的今天,如何有效地调试并发程序,成为一个重要且亟待解决的研究课题.并发程序的不确定性及其行为的复杂性,使得传统的调试技术难以得到有效的应用;而软件维护场景中错误发现与错误调试过程的分离使得错误重现难以实现,面向缺陷报告的调试需求使得自动的错误定位技术难以应用,加剧了调试的困难.针对软件维护阶段由缺陷报告导向的程序调试场景,提出了可视化的并发程序调试方法.该方法能够根据缺陷报告中的信息对程序进行切片,缩小需要分析的代码范围;通过静态分析构造出程序行为的全局视图,帮助程序员发现隐含的程序执行路径;根据事件结构的语义简化程序行为视图,使得行为模型规模可控;根据图形中的分支,引导用户关注路径中的关键操作,从而更快地发现程序中的缺陷.与动态调试方法相比,该方法能够避免错误重现的代价.借助缺陷报告中的信息以及事件结构模型的特点,该方法能够尽量减少状态爆炸的发生.已开发出的交互式并发程序调试工具原型 JESVis Debugger,初步实现了所提出的方法.

关键词: 可视化调试;并发程序;事件结构;配置结构;标记迁移系统

中图法分类号: TP311 **文献标识码**: A

中文引用格式: 伍晓泉,魏峻.基于事件结构的并发程序可视化调试方法.软件学报,2014,25(3):457−471. http://www.jos.org.cn/1000-9825/4421.htm

英文引用格式: Wu XQ, Wei J. Visual debugging concurrent programs with event structure. Ruan Jian Xue Bao/Journal of Software, 2014,25(3):457−471 (in Chinese). http://www.jos.org.cn/1000-9825/4421.htm

Visual Debugging Concurrent Programs with Event Structure

WU Xiao-Quan[1,2,3], WEI Jun[1,2]

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

[2](Technology Center of Software Engineering, Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

[3](University of Chinese Academy of Sciences, Beijing 100049, China)

Corresponding author: WU Xiao-Quan, E-mail: wuxiaoquan07@otcaix.iscas.ac.cn

Abstract: The multi-core and concurrency technology are widely used today. How to debug concurrent programs effectively becomes an important topic. Due to nondeterministic and complex behavior of the concurrent program, traditional debugging techniques are inappropriate to apply. In bug-reports-oriented debugging scenarios, the separation of error detection and error debugging processes makes the failures more difficult to reproduce, which exacerbates the difficulty of debugging. For bug-report-oriented debugging scenarios, this paper proposes a method to visualize program behaviors to assist programmers to debug manually. The method reasons concurrent program behavior through static analysis, giving a global view to programmers to help them observe the behavior of the program and guiding their attention on suspicious operation. This approach can avoid the cost of reproducing concurrent failures. With the information in the bug report as well as the feature of the event structure model, it eliminates the state explosion problem that may arise

* 基金项目: 国家高技术研究发展计划(863)(2012AA011204);国家自然科学基金(61173005, 61003029)
收稿时间: 2013-01-30; 修改时间: 2013-03-11; 定稿时间: 2013-05-03

during static analysis of concurrent programs. A prototype of interactive debugging tool for Java called JESVis Debugger, is developed as an initial realization of the proposed method.

Key words: visual debugging, concurrent program, event structure, configuration structure, labeled transition system

随着多核技术的发展,并发程序重新受到人们的关注,并已得到了广泛的应用.然而,开发出正确的并发程序并不是一件容易的事情,并发程序中往往存在缺陷.程序调试是一种从程序中发现和减少缺陷的系统化方法,是软件工程领域的重要研究问题.目前,针对并发程序的调试技术尚未发展成熟.有研究表明,在实际开发过程中,尚缺乏有效的调试工具辅助程序员完成并发程序调试.同时,调试过程中常常会引入新的程序缺陷[1].因此,如何有效地调试并发程序成为了一个重要而亟待解决的研究课题.

程序调试过程起始于一个可观察到的程序失效(failure).程序失效是指程序运行时可观察到的与预期不符的程序行为,它是程序中静态存在的缺陷(defect/bug)引起的.程序缺陷是指实际程序与正确程序之间的差异,例如一条写错了的程序语句,它通常是由程序员的错误(mistake)引起的.程序调试的目的就是根据程序失效的现象找出导致失效的程序缺陷.

对于顺序程序,常用的高度方式是基于断点的,它以程序的状态转换模型为基础:在这种模型下,每条程序语句的执行,会将程序从一个状态迁移到另一个状态.测试人员可以设定断点,运行程序使其停止在指定的位置,通过获取内存单元的内容,可以了解程序运行到断点处的状态,检查其是否与预期相符,以期发现程序中的缺陷.通过不断地更换断点的位置,反复迭代上述过程,可以多次执行程序,逐步缩小探测范围,直到找出程序中的缺陷.

然而,基于断点的调试方式并不适用于并发程序:一方面,这种方式要求程序行为是确定的,程序失效是可重现的,这样,每次执行都会使程序依次经历相同的状态.而对于并发程序,并发操作的调度顺序是不确定的,因此多次执行程序到达的状态可能不同,这会导致并发错误无法重现;另一方面,并发程序中的资源竞争、死锁等问题与进程间的相互作用有关,通过观察某一程序状态难以发现[2].同时,这种观察行为本身会影响并发操作的时序,从而影响程序的执行,这种现象被称为探测效应(probe effect) [3].

除此之外,并发程序调试中的另一个挑战是:程序行为复杂导致程序行为难以理解.线程之间相互作用复杂,开发人员难以通过阅读程序代码理解程序行为.将程序特征可视化,是辅助程序理解的重要手段,研究人员采用了各种方法对程序的静态和动态特征进行展示.程序的静态特征包括程序数据结构和程序依赖图[4],它刻画程序的组成、数据结构等内容;程序的动态特征包括程序运行到某一个状态下的程序变量值、堆栈状态以及程序行为.其中,程序行为通常用 UML 顺序图、状态图和协作图等描述[5−7].

程序调试的典型场景发生在程序测试阶段:在软件发布前,测试人员通过测试发现程序失效.确定性重放和自动的错误定位,是测试阶段常用的两种自动化调试技术.在发现程序失效后,通过重现错误可以找到导致错误的程序执行路径.为了克服并发程序不确定性给调试带来的困难,研究者提出了各种方法试图实现并发程序的确定性重放[8−10].测试阶段还可以利用大量的测试用例,观察、跟踪、记录程序的运行结果,利用自动的错误定位技术找出程序缺陷[11].而程序调试的另一场景发生在软件维护阶段,在这种场景下的程序调试是缺陷报告驱动的:用户在使用软件的过程中发现了程序失效、异常等,向软件提供方提交缺陷报告,并记录在缺陷跟踪库中.用户在报告中记录导致错误的输入和执行步骤、错误发生时所产生的错误日志及异常堆栈等信息,供软件提供方调试.此场景区别于测试阶段错误调试的特点,使得确定性重放技术和自动错误定位技术难以应用,具体包括以下两个方面:

- 错误发现与错误调试过程相互独立.这两个过程通常由不同的人执行:错误发现通常由用户完成,错误调试由软件提供方完成.双方的软件运行环境也不尽相同.这些特点会加剧错误重现的困难.一方面,缺陷报告中能用于错误重现的信息可能并不完整,甚至存在错误.据统计,缺陷报告中存在错误或缺陷报告不完整而导致程序错误无法重现,是著名开源软件 Eclipse 的缺陷库中最严重的问题[12];另一方面,出错的环境难以复制,错误难以跟踪记录.在用户环境中进行完整的错误跟踪和记录通常是不易被接受的,因为它可能带来程序性能的损失以及用户数据泄露的风险;

- 缺陷报告驱动的调试更具有针对性.它只针对缺陷报告中提及的程序失效行为,寻找对应的程序缺陷.而测试过程通常并不针对某一个错误或某一个类型的错误.因此,测试过程中可以使用自动化的测试工具获得大量测试结果,从而根据程序语句或线程通信与程序失效之间的关联关系进行自动的错误定位,减轻人工调试的负担.但它却难以应用于缺陷报告导向的程序调试场景.

综上所述,软件维护阶段错误调试的场景,给并发程序的调试带来了更大的挑战.但同时,用户提交的缺陷报告中也包含了对程序错误的描述等有用信息.本文针对缺陷报告导向的程序调试场景,研究并发程序的调试方法.由于并发错误重现困难且自动错误定位方法难以实施,错误调试主要依赖于程序员人工进行.因此,对程序行为的理解和推断成为程序调试的关键步骤.将程序行为可视化,可以有效地帮助程序员理解并发程序行为.为了避免动态调试方法中需要重现并发错误的难题,我们采用静态分析的方法,推理出并发程序可能的行为路径.此项任务主要面临两方面的挑战:一是如何构造并发程序行为模型;二是如何克服静态分析并发程序可能出现的状态空间爆炸问题.对于前者,我们采用事件结构作为描述并发程序行为的模型,在现有开源工具的基础上,实现了从程序源代码中自动抽取出配置结构图(configuration structure)的过程;对于后者,我们利用用户提交的缺陷报告缩小分析范围,只需从源代码中抽取部分程序片段进行分析,避免了生成整个程序的行为视图.此外,我们根据事件结构的语义对程序行为视图进行了简化,删去了大量对调试无用的路径,并对分支路径中的操作进行标记,从而引导用户关注影响程序行为的关键步骤,辅助其定位程序缺陷.该方法具有以下特点:(1) 不需要多次执行程序以触发程序失效,避免了错误重现的代价;(2) 能够根据缺陷报告中的信息,借助程序切片技术缩小可疑代码的范围,辅助错误定位;(3) 能够自动地从程序源代码中抽取程序行为模型,利用配置结构图,将与错误状态相关的程序行为展示出来,简单、直观;(4) 能够自动检测出活锁死锁等并发错误,并引导程序员关注影响程序行为的关键操作,发现数据竞争等并发错误.

1 问题分析

在缺陷报告驱动的调试场景下,程序员通常按照如下步骤完成调试工作:(1) 理解缺陷报告:从缺陷报告中分析程序出错点、可疑的程序变量和方法、导致程序出错的输入等信息;(2) 尝试错误重现:重新执行程序,从而获得到达程序出错点的执行路径;(3) 程序理解:根据错误路径上的程序语句,理解相关的程序行为;(4) 错误定位:借助可视化调试工具,在错误路径上逐步缩小可疑语句的范围,定位程序中导致程序失效的缺陷.

在这一过程中,缺陷报告的理解主要依靠人工完成.并发程序自身固有的不确定性导致错误重现困难.而在缺陷报告导向的错误调试场景中,错误发现与错误调试过程的分离,使得现有的并发错误重现方法难以被用户接受.此场景下的并发程序调试过程主要依靠于程序员手工完成.人工调试的主要目标并非找到错误的执行路径,而是寻找导致失效的程序缺陷.错误路径中包含了过多的语句信息,冗长的错误执行轨迹有时多达几千甚至几万个操作,无法提供给程序员错误定位所需的信息[11].这要求调试工具不但应有辅助程序理解的功能,还应具有辅助程序员进行错误定位的能力.

由于无法使用大量的测试用例,基于谱的方法(spectrum-based method)和基于统计的方法(statistics-based method)都无法应用在此场景中用于自动的错误定位[13].因此,我们采用基于切片的错误定位方法(slice-based method),根据缺陷报告中的信息,对程序进行切片.缺陷报告中包含两类信息:一类为结构化信息,它们通常有固定的格式,比如:程序错误日志、Java 程序的异常堆栈信息等;另一类为非结构化信息,如文字描述.根据缺陷报告,通常可以发现程序的“出错点”,例如:程序的哪一行抛出了异常,或哪一行的输出是错误的.根据“出错点”可以找出值得怀疑的程序变量,它的值可能与程序失效相关.另外,缺陷报告中的日志与异常堆栈信息记录了错误路径上调用的方法,它参与了程序的错误执行,也可能与程序失效相关.我们将这些可疑变量与方法从缺陷报告中识别出来作为方法的输入,对切片后的程序行为进行可视化.程序行为可视化方法的有效性主要受限于它所能处理的程序规模,程序切片能够提高可视化方法的有效性:一方面,当可视化图形中的节点数目超过 100 个时已难以通过人工进行分析,因此,展示程序的全状态空间并不是一个好主意——切片能够引导程序员关注与缺陷报告相关的程序行为,帮助其进行错误定位;另一方面,程序规模增大之后,程序的状态空间也会随之增大,在

构造程序行为模型之前进行切片,可以极大地减少发生状态空间爆炸的可能.

程序行为是最重要的程序特征.通过静态分析展示程序行为,可以有效地帮助程序员理解程序行为.而采用什么样的模型来展示并发程序行为,是可视化方法中的关键.程序组成、数据结构等信息不能刻画并发程序的行为;控制流图也不适用,因为并发程序中频繁的线程交互会使控制流图变得庞大而杂乱.UML 顺序图、协作图等可以描述线程之间的通信,但它通常只用于描述程序某一次执行中的线程交互关系,用于辅助检测死锁等安全性问题,而对非死锁问题无效.在模型检测领域,通常用一个有限状态自动机来刻画并发程序行为,标记迁移系统(labeled transition system)是其中的典型代表.它能够描述系统所能达到的全局状态以及状态之间的迁移关系.标记迁移系统中每条从初始状态到终结状态的路径代表程序的一次执行,它刻画了系统所有可能的执行路径,图形复杂,因此并不适合直接用于可视化程序行为.此外,在该模型中无法根据终结状态区分正确和错误的执行路径,因为到达同一终结状态的路径可能既包含正确的执行,也包含错误的执行.例如:图 1 中包括两个线程,一个是 Java 虚拟机为 main 函数自动创建的主线程(线程 1);另一个是类 MyThread 的实例 *t*(线程 2),在 main 函数中创建并触发.两个线程共享变量 *x*,可以各自执行对 *x* 的读写操作,最后在主线程中合并.在这个例子中,程序的状态迁移示意图如图 2 所示.两线程中的操作无论以怎样的次序交替执行,都会终止在相同的状态上(*x*=1,*y*=1).然而这个程序中存在一条含有“原子性违背(atomicity voilation)”错误的路径:在执行过程中,线程 2 中对 *x* 的赋值操作(第 33 行)可能会在线程 1 对 *x* 的两次读操作(第 7 行和第 9 行)之间执行,导致线程 1 这两次读到的 *x* 值不一致而引发程序抛出异常.

```
1   public class Example {
2       protected static int x=1;
3       protected static int y=−1;
4       public static void main(String[] args){
5           Thread t=new MyThread();
6           t.start();
7           if (x>=0){
8               try {
9                   y=f(x);
10              } catch (Exception e) {
11                  e.printStackTrace();
12          }
13      }
14          try {
15              t.join();
16          } catch (InterruptedException e) {
17              e.printStackTrace();
18          }
19          x=x*x;
20          y=x*x;
21      }
22      private static int f(int x)
23          throws Exception{
24          if (x>0){
25              return x;
26          } else {
27              throw new Exception();
28          }
29      }
30  }
31  class MyThread extends Thread{
32      public void run(){
33          Example.x=−1;
34      }
35  }
```

Fig.1 Sample program

图 1 示例程序

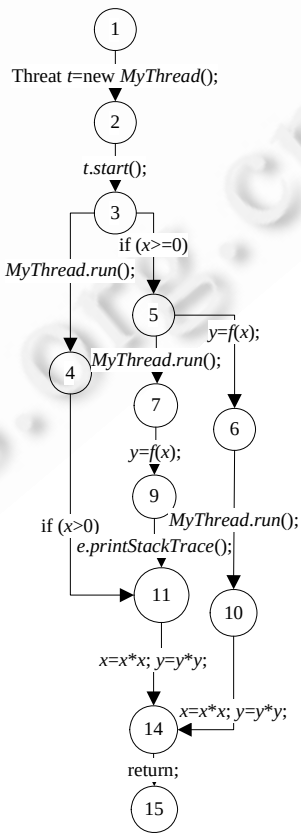


Fig.2 State transition diagram of the sample program

图 2 示例程序的状态迁移示意图

综上所述,现有的程序行为模型在用于并发程序调试时存在以下两方面的问题:

- 模型表达能力不足.控制流图、UML 顺序图和协作图不能刻画程序状态,标记迁移系统不能区分到达同一终结状态的程序路径上的不同事件;
- 模型规模不可控.当程序规模复杂时,控制流图、UML 顺序图和协作图会变得杂乱而无法进行人工分析,标记迁移系统的状态空间数目会随程序规模呈指数级增长而出现状态爆炸问题.

为克服以上问题,我们采用配置结构图对程序的行为进行可视化.它将不同上下文中执行的动作描述为事件,利用事件结构的语义,将所有可能的执行路径划分为等价类,只需要选取等价类中的一条路径进行展示,就可以刻画程序行为的全局视图.还可以利用图中的分支结构,引导程序员关注影响程序行为的关键操作,辅助其完成错误定位.

2 方 法

本文在之前工作[14]的基础上,完善了缺陷报告驱动的并发程序错误调试方法.该方法的输入为程序源代码,以及从缺陷报告中分析出的可疑变量和方法.输出为一张带有标记的程序行为视图.方法的主要步骤包括:

- 程序切片:根据输入,利用切片技术去除与程序失效无关的代码,缩小分析范围,辅助错误定位;
- 模型构造:从程序源代码中抽取出程序行为,并用配置结构模型表示;
- 模型分析与展示:从配置结构模型中分析出部分关键的程序行为,并通过用户交互界面将其展示成便于程序员理解的方式.

2.1 程序切片

程序切片是指通过分析程序语句之间的依赖关系,剔除部分程序语句以减少程序规模的方法.它最早由Weiser[15]提出,通常应用于程序调试、测试和程序理解等.在本文中,我们用程序切片来剔除与可疑变量和可疑方法无关的语句,缩小需要探查的程序范围.静态程序切片方法通过分析程序语句之间的静态依赖关系,生成程序的静态依赖图来执行切片的方法.在本文中,切片的准则由可疑变量和可疑方法两部分构成.切片的结果,是与可疑变量有依赖关系,同时包含了可疑方法的程序语句集合.并发给程序切片技术带来了新的挑战.共享内存单元和同步结构增加了线程之间的依赖.而由于推断所有线程之间的交替关系十分复杂,因此计算线程之间的依赖关系也变得复杂[16].在本文中,我们采用静态的后向程序切片算法[17]计算影响可疑变量值的那些程序语句的范围.

2.2 模型构造

从代码中提取程序的抽象行为模型是具有挑战性的工作,挑战主要来自于复杂的程序语义.构造出的模型既要准确地描述程序的行为,又要足够清晰简洁易被程序员接受.与程序代码、程序控制流图和程序语句依赖图相比,基于状态的程序行为模型能够更清晰地表达程序的可能行为轨迹,它既能表示一次执行时语句的执行顺序,又能表示不同的交替次序之间的关联关系,以及多次执行中程序所到达的状态是否相同.但表示程序全状态空间的模型并不适合用于可视化,也无法区分不同执行中发生的不同事件.为了克服这些问题,在模型构造时,我们从高级程序设计语言源代码中抽取出程序的行为模型,并表示为一种事件结构模型——配置结构(configuration structure)的形式.

2.2.1 基于配置结构的行为模型

配置结构(CS)[18]可以描述系统中的动作以及动作之间的因果关系,它由两个元素构成 $cs=(E,C)$.其中,

- E 代表一个事件集合;
- C 代表在 E 上的一个集合,它的成员 c 是 E 的一个有限的子集.

一个配置 c 可以看作程序到达的一个状态,这个状态是系统从初始状态开始连续执行了 c 中的事件之后达到的状态.一个空的配置{}表示系统的初始状态.一个良构的(well-formed)配置结构具有与文献[19]中定义的事件结构相同的表达能力.下面用一个例子来说明配置结构的语义.一个并发程序有两个线程,每个线程中只有一个操作,分别是 $x=1$ 和 $x=2$.那么这个并发程序所能到达的状态空间用状态迁移图表示,如图 3(a)所示,其对应的

配置结构为 $cs'=(E',C')$:

- $E'=\{e_1,e_2,e_3,e_4\}$,其中,e_1,e_4 对应操作 $x=1$,e_2,e_3 对应操作 $x=2$;
- $C'=\{\{\cdot\},\{e_1\},\{e_2\},\{e_1,e_3\},\{e_2,e_4\}\}$.

我们将只相差一个事件的两个配置用线连起来,并让箭头指向包含事件数目较多的那个配置,所得到的即为程序对应的配置结构图,如图 3(b)所示.它与状态迁移图的区别在于:配置结构图是无环的,随着事件的发生不断生长.当配置结构图出现分支时,分支不再汇合,因为分支的路径分别经历了不同的事件.其优点是,事件结构是一种真实并发模型(true concurrency model),组成模型的事件具有偏序关系.相比状态迁移图中使用操作所有可能的交替(interleaving)来刻画事件之间的偏序关系,更加简洁.

(a) 状态迁移图 (b) 配置结构图

Fig.3 Labeled transition diagram and configuration structure diagram

图 3 标记迁移图与配置结构图

2.2.2 模型构造方法

目前,已有一些模型检测方面的研究工作能够从源代码中抽取程序行为,表示为状态迁移图的形式.为了利用现有的研究成果,我们将模型的构造过程分为两步:先从程序中抽取状态迁移图,再将它转换为配置结构图.从程序中抽取抽象的状态模型,其难点主要在于如何将程序语言的复杂语义与状态迁移模型相匹配.构造方法有两种:一是基于追踪(tracing)的方法,通过执行系统生成程序的执行轨迹,从而暴露系统的行为,再将其转换成状态迁移模型[20];另一种方法叫抽象释义(abstract interpretation),通过明确定义程序变量之间的对应关系,生成抽象的系统行为模型[21].我们采用了后一种方法.得到状态迁移模型后,利用模型转换算法,将其展开为配置结构模型.

为了定义这个过程,我们首先定义一个确定性的标记迁移系统(definite labeled transition system,简称 DLTS)$L=(S,\Sigma,\Delta,\hat{s})$包含 4 个元素.其中,

- S 是一个状态的集合,表示系统可能达到的所有状态.系统的一个全局状态由两部分构成,分别是系统中全局变量的状态和各个线程计数器的状态;
- Σ是一个有限的操作集合,表示系统中所有的操作;
- Δ是一个偏序关系 $S\times\Sigma\to S$;
- $\hat{s}\in S$ 是系统的初始状态.

而与之对应的配置结构是一个四元组(E,C,lb,st),其中,

- (E,C)组成了一个配置结构;
- lb:是 E 到Σ的一个映射,配置结构中的每个事件都有一个迁移系统中的迁移与之对应;
- st:是 C 到 S 的一个映射,每个配置对应迁移系统中的一个状态.

我们使用展开算法将状态迁移图转换为配置结构图.配置结构图描述系统发生的事件而并非状态,因此配置结构图中没有环.若程序的状态迁移图中无环,那么展开算法是可终止的.在使用展开算法之前,我们首先检验程序的状态迁移图是否有环.有环的程序包含两种情况:一是程序中存在无限循环;二是程序中存在活锁.在这种情况下,我们直接在状态迁移图中标记出循环路径返回给用户.

对于无环的状态迁移图,展开算法如图 4 所示.算法的输入为一个标记迁移系统,输出为一个配置结构.算法基于对状态迁移图的宽度优先遍历.配置所在的层数等于该配置中包含的事件的个数,初始状态的层数为 0.算法从第 0 层开始,逐层生成相应的配置.方法 *unfolding* 描述了算法的主要框架.在算法中用到了 h 和 *cs_ts_map*

两个数据结构:*h* 用于记录可以继承到每个配置的事件的集合,*cs_ts_map* 记录了配置结构与标记迁移系统中的状态的对应关系.算法首先进行初始化:生成的一个空的配置 *initCF*={}(第 7 行),对应于标记迁移系统中的初始状态;同时,将其对应的继承事件集合置为空(第 10 行),将它和标记迁移系统中的初始状态的对应记录在 *cs_ts_map* 中(第 13 行),并将 *initCF* 加入配置结构图中(第 15 行).接下来,从 *initCF* 开始,对于处在第 *l* 层的每个配置,根据方法 *updateFrom* 中的算法生成其出边和出边所到达的配置节点(第 19 行~第 29 行).在探索完所有处于 *l* 层的配置后,更新每个新生成的配置所对应的继承事件集合 *h*.之后将 *l* 加 1,开始探索下一层的配置,直到不再生成新的配置,或到达人工设定的配置层数的最大值(第 30 行~第 34 行).

方法 *updateFrom* 以配置 *cf* 作为输入,通过 *cs_ts_map* 可以找出其在标记迁移系统中对应的状态 *state*(第 38 行),对于 *state* 的每一条出边(表示操作 *t*),我们对应生成 *cf* 的出边.边上所表示的事件根据 *h*(*cf*)生成.若 *h*(*cf*)中包含一个事件 *e* 与 *t* 对应,我们就用 *e* 标记 *cf* 的这条出边(第 46 行);否则,生成一个新的事件 *e′*用于表示 *cf* 的出边(第 48 行).出边到达的配置,由边上的事件决定.最后更新 *cs_ts_map*.

方法 *updateHset* 用于计算每个配置所对应的继承事件集合.*cf* 所能继承的事件的集合是它之前遇到的事件集合的一个子集,其中不包括 *cf* 中的事件.简单说来,如果一个事件与它之前所遇到的所有事件都相互独立,那么它可以被继承到下一个配置中去.算法的细节可见文献[18].此算法的特点是:它能够动态的判断两个操作之间的依赖关系,这样生成出的配置结构能够更准确的刻画并发系统的行为,避免静态方法对操作之间依赖关系的近似判断.

```
1   h: Configuration→Set⟨Event⟩
2   cs_ts_map: Configuration→State
3
4   EventStructure unfolding (TranstionSystem tranSys)
5   {
6       /*generate initial configuration initCF*/
7       Configuration initCF=new Cofiguration();
8
9       /*the inherit set of initCF is empty*/
10      h.put(initCF,new Set⟨Event⟩());
11
12      /*initCFG maps to initState*/
13      cs_ts_map.put(initCFG,tranSys.initState);
14
15      eventStructure.addnode(initCF);
16      int l=0;        /*the level of configuration*/
17
18      /*generate configuration structure by level*/
19      while (true) {
20          boolean exist=false;
21          /*generate outgoing edges and arrived
22              configurations for each
23              configuration at level l*/
24          For each cf in cs_ts_map.keySet() {
25              if (cf.size()==l) {
26                  exist=true;
27                  updateFromState(cf);
28              }
29          }
30          if (!exist)     {break;}
31          For each cf in cs_ts_map.keySet() {
32              if (cf.size()==(l+1)) {
33                  updataHset(cf);
34          }}}
35      return eventStru;
36  }
37  updateFromState(Configuration cf) {
38      State state=cs_ts_map.get(cf);
39      Collection⟨E⟩ edges=tranSys.getOutEdges(state);
40      if (edges==null||edges.isEmpty())       {return;}
41      For each element edge in edges {
42          String operation=tranSys.getLabelOf(edge);
43          Event event;
44          /*if there is an event in h(cf), then take it.*/
45          if (h.contains(cf,operation)) {
46              event=h.getEvent(cf,operation);}
47          /*otherwise, create a new event for operation.*/
48          else {event=createNewEvent(operation);}
49          Configuration nextCFG=cf+event;
50          eventStru.addnode(nextCFG);
51          eventStru.addedge(cf,nextCFG,event);
52          cs_ts_map.put(nextCFG,tranSys.getDest(edge));
53  }}
54  updataHset(Configuration cf) {
55      h.put(cf, new Set⟨Event⟩());
56      For each event e encounted by cf{
57          if (cf.contains(e)) {continue;}
58          boolean dependent=false;
59          For each predecessor of cf denoted by preCfg{
60              State preState=cs_ts_map.get(preCfg);
61              String operation=
62                  eventStru.getEvent(preCfg,
63                  cf).getOperation();
64              if (preCfg has encounted e) &&
65                  !(e is enalbed in preCfg) &&
66                  tranSys.independent(operation,
67                  e.getOperation(),preState))) {
68                  dependent=true;
69                  break;}}
70          if (!dependent) {
71              h.getHSet(cf).add(e);}}
72  }
```

Fig.4 Unfolding algorithm to generated the configuration structure

图 4 根据标记迁移系统生成配置结构的展开算法

在模型构造的过程中,需要对程序的整体进行编译,并将系统调用的所有代码加入分析过程,其中不仅包含

用户编辑的部分,还包括程序代码库、编程框架中一些隐式调用的代码.这样会造成程序行为图过于庞大,而无法使程序员关注到自己编写的或感兴趣的那部分代码上.因此,我们需要对生成的模型进行简化,删减用户不关注的程序库中的语句所对应的边,限制方法调用的深度,从而避免图形中生成与用户代码无关的边和节点,减小模型的规模.

2.3 图形分析与展示

为了便于程序员定位错误,我们需要对自动生成的配置结构图进行分析,并借助可视化交互界面对模型进行展示.其中,分析过程包括死锁检测和关键路径提取.

2.3.1 死锁检测

死锁状态表现为配置结构图中的一个终结状态.判断方法为:若到达终结点路径的最后一个操作不是 return 操作,则该终结状态可能为死锁状态(deadlock).不考虑程序设计语言中的异常处理,过程式程序通常以一个方法为程序入口,顺序执行方法中的语句,直到最后一句,通过 return 语句返回.若未执行到 return 操作程序就到达了终结状态,则很可能是因为程序中出现了死锁.对于死锁状态,我们通过重新布局到达此终结状态的路径来显示线程之间的通信.

2.3.2 关键路径提取

在配置结构图中,对于从初始状态到达相同终结点的不同执行路径,他们在语义上是等价的.例如:到达相同终结点的 A,B 两条执行路径,若沿 A 路径执行到终结状态发生死锁,则沿着 B 路径执行到该终结点必然发生死锁.我们利用配置结构中到达同一终结点的路径之间的等价性简化系统行为视图,从而减少状态爆炸的发生.

基于程序的配置结构图,我们将关键路径定义如下:

- 从初始节点出发到达终节点的一条路径称为完整路径,P 为配置结构图中所有完整路径的集合;
- $T=\{t_1,t_2,\ldots,t_N\}$为配置结构图中 N 个终节点的集合;
- 对于 $p\in P,T(p)=t$ 表示路径 p 的终结点 t;
- $C_i=\{p\in P|T(p)=t_i\}$为一个等价类,$P=C_1\cup C_2\cup\ldots\cup C_N$;
- $KP\subseteq P$ 为该程序的关键路径,当且仅当$\forall 0\leqslant i\leqslant N,|KP\cap C_i|=1$.我们将从每个等价类中选取一条完整路径进行展示的过程称为关键路径提取.提取关键路径的方法为:从每个终结点开始逆向搜索到程序的初始节点终止,所得到的即为该程序的关键路径.

在关键路径图中,程序分支点对分析程序行为至关重要.从同一状态出发,若经过不同的操作使程序到达了不同的状态,那么这些操作之间存在冲突,例如图 3 中的 $x=1$ 和 $x=2$ 两个操作.这些冲突的操作可能是对共享变量的读写操作,它们在不同关键路径上的执行顺序不同,正是这些不同的执行次序导致了程序中顺序违背、原子性违背等常见的并发错误.因此,识别并标记这些操作有助于引导程序员发现程序中的问题.

3 方法实现

我们针对 Java 程序,利用开源框架 Indus(Indus:http://indus.projects.cis.ksu.edu/),Bandera(Bandera:http://bandera.projects.cis.ksu.edu/),GraphViz(GraphViz:http://www.graphviz.org/)实现了本文所提出的方法,完成了交互式 Java 并发程序调试工具 JESVis Debugger 的核心功能.工具的实现框架如图 5 所示.下面我们结合图 1 中的例子,讲解方法的主要步骤和实现结果.本节所涉及到的图片与文件,可从项目页面 http://code.google.com/p/jesvis-debugger/downloads/list 处下载.在这个例子中,程序失效的表现为异常退出,并打印异常堆栈信息.当用户观察到程序失效后,通常会将这一现象包括异常堆栈信息记录在缺陷报告中,供程序员调试.在调试中,程序员首先对程序缺陷报告进行理解,并尝试重现报告中描述的程序失效.在这个例子中,很容易判定程序失效与第 9 行程序有关,但程序失效却难以重现.因此,我们利用 JESVis 对程序进行调试.从程序源代码中可以看出,第 9 行语句涉及 x,y 两个变量,我们将这两个变量作为方法的输入.

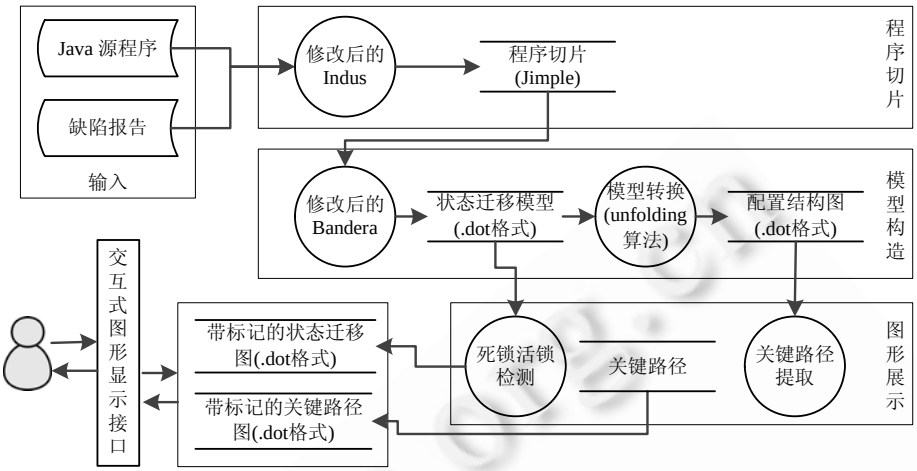


Fig.5 Framework of JESVis Debugger

图 5 JESVis Debugger 实现框架

3.1 利用Indus对程序进行切片

我们首先根据输入对程序进行切片,进行初步的错误定位.Indus 是一个针对 Java 的程序切片框架,它以著名的静态程序分析框架 Soot(Soot:http://www.sable.mcgill.ca/soot/)为基础,首先将 Java 程序转换成一种三地址代码格式的中间表示——Jimple,在此基础上进行分析.采用中间表示增加了系统的灵活性,同时降低了分析的复杂度.

Indus 定义了两个级别的程序切片准则,分别为程序语句级(statement level)和方法级别(method level)的准则.程序切片是根据这两级的切片准则,计算与准则中的程序语句和方法有依赖关系的其他程序语句,并将它们保留在切片结果中.本方法的输入是识别出可疑变量和可疑方法,其中,可疑方法与 Indus 中的方法级切片准则相对应.对于识别出的可疑变量,我们将用户代码中包含可疑变量的程序语句作为程序语句级的切片准则.我们采用 Indus 默认的静态后向切片方法对程序进行切片.将切片后的程序作为下一步模型构造的输入.

3.2 基于Bandera构造程序的配置结构模型

Bandera 是一个针对 Java 程序的模型检测工具集,具有程序分析、转换及模型检测的功能[22].它以用户自定义的主类中的 main 函数为入口构造程序的标记迁移系统,在此模型上进行明确状态的模型检测.在实现中,我们修改了 Bandera 工具集中用于执行模型检测的工具 BogorTool,从而将 Java 程序的标记迁移系统模型提取出来.之后,我们实现了将标记迁移系统转换为配置结构模型的算法.

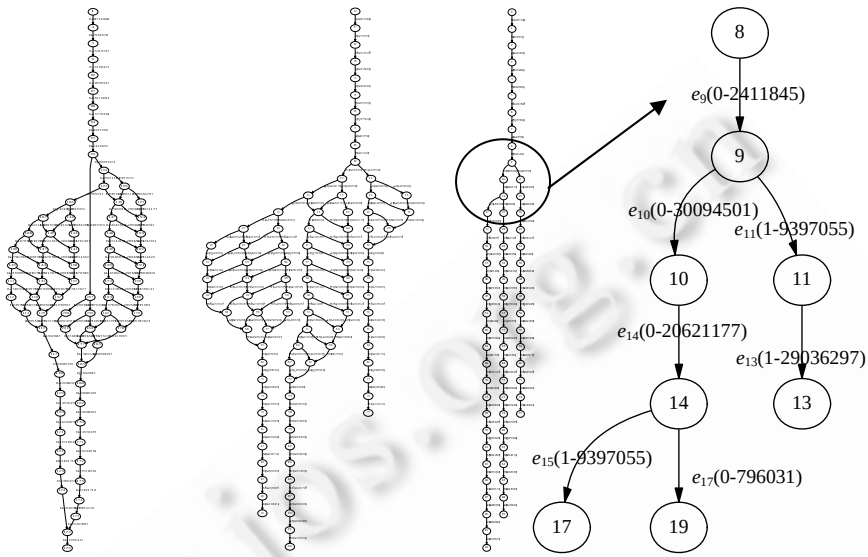
由于采用了 Jimple 格式作为中间代码,会导致模型构造过程中的两个的问题:

- 程序状态图繁杂,节点数目大量增加,不便于人工观察;
- 状态图中的边与源程序语句不对应,而是对应 Jimple 语句,增加了程序员理解程序的负担.

例如,图 1 中程序共含有两个线程 8 个操作,自动生成的事件结构图中状态数目超过 150 个.因此,对于自动抽取出的模型,需要进行图形简化,在状态图中去除与程序理解不相关的边和节点,并将属于源代码中同一条语句的边进行合并,使其与源程序相对应.

根据图 1 中的程序自动生成的状态迁移如图 6(a)所示.从初始节点开始的每一条到达终点的路径都表示程序的一次执行,每次执行中并发操作的执行顺序不同,但他们最终都到达相同的终结状态(y=1,x=1).经过检测,状态迁移图中无环.将其转换成为的配置结构图如图 6(b)所示,程序的状态空间分成了 3 支,分别由 3 个不同的事件集表示.为了图形的简洁,我们为每个操作设置了一个 ID,ID 由操作所在的线程号和操作的标识符组成,中间用短线连接.操作标示符与操作内容的对应关系存放在 transition.xml 文件中.这样,我们就构造出了图 1 所示程

序中的配置结构图,以下对程序的分析都以此图为基础.



(a) 状态迁移图 (b) 配置结构图 (c) 关键路径图

ID	操作(Bandera 定义的 bir 格式)	所属方法	行号
1-9397055	when *BogorJava.isInitialized*(*isi*,"(|*Example*|)") do {/|*Example.x*|\:=−1;} goto loc2;	*MyThread.run*	33
0-30094501	do {[|$$i_0$|]:=/|*Example.x*|\;} goto loc6;	*Example.main*	7
0-796031	do {[|$$i_1$|]:=/|*Example.x*|\;} goto loc8;	*Example.main*	9

(d) 操作列表

Fig.6 Analysis of the sample program

图 6 示例程序分析过程

3.3 模型分析与展示

首先进行死锁检测.使用第 2.3.1 节中的方法对图 6(b)进行检测.图中所有路径分别到达 3 个不同的终结状态.到达终结状态前的最后一个操作的 ID 均为 0-11906412,从 transition.xml 中可以看出,此操作为 return 语句.由此判断此程序中不存在死锁.

由于配置结构图中路径纷杂,不利于程序员分析程序行为,因此,我们利用配置结构中路径的等价性根据配置结构图生成程序的关键路径.由于采用 Jimple 中间表示后,一条 Java 源程序对应多条 Jimple 语句.为了简化图形,将其与源程序对应,在关键路径提取时,需要尽量保持同一线程操作连续,以便将它们还原成源代码中的语句.同时,保持同一线程操作连续有助于减少线程上下文切换,利于程序员理解程序交互行为.基于这个需求,我们尽量选取线程上下文切换少的关键路径作为输出.从图 6(b)中抽取出程序的关键路径,如图 6(c)所示.从图可以观察到程序行为的全局视图:此程序一共包含 3 条关键路径,分别引导程序到达 3 个不同的状态.其余执行轨迹,都与这 3 条执行路径中的某一条路径具有等价性.因此,分析时只需要分析这 3 条路径上的程序行为.

在图中,我们记录下程序分支处的 3 个操作 1-9397055,0-30094501,0-796031,3 个操作代表的含义如图 6(d)所示.操作 1-9397055 对应于 MyThread.run 方法中对 x 的赋值操作(图 1 第 33 行);操作 0-30094501 对应于 main 方法中的 if 操作(图 1 第 7 行)的一个步骤,先将 x 的值取到临时变量 i_0 中;操作 0-796031 对应于 main 方法中语句 $y=f(x)$(图 1 第 9 行)的一个步骤,先将 x 的值取到临时变量 i_1 中.这 3 个操作是程序调试过程中应当被关注的关键操作,它们引导程序最终运行到不同的状态.为了便于程序员调试,我们将图 6(c)中的操作对应到程序源代码中.我们根据操作所对应的源程序中的行号,对图中一条路径上来自源代码同一行的操作进行合并.合并后的

图形如图 7 所示,我们将含有操作 1-9397055,0-30094501,0-796031 的语句块标记在图中.图中带有符号“+”的边是可展开的边,为便于查看,我们将属于同一方法的边折叠起来.虚线连接的表示起点和终点处两条边对应源程序中同一行语句.

图 7 是程序员看到的最终视图.为了便于描述,我们将图中的 3 条路径从左到右分别编号为 1,2,3.对于例 1 中的程序,出错点为程序第 9 行.调试时首先根据程序出错点找到包含程序第 9 行语句 $y=f(x)$的程序路径:路径 1 与路径 2,然后在这些路径中分辨出哪一条是导致程序出错的路径.由于程序出错时打印了异常堆栈信息,因此可以判断路径 2 为导致程序出错的路径,因为路径 2 中含有语句 *e.printStackTrace*(),而路径 1 中不包含这条语句.接下来,我们分析错误路径中的关键操作与其他路径中的关键操作有什么区别.路径 2 中的关键操作有 3 条,为 if ($x\geqslant 0$),*MyThread.run*()和 $y=f(x)$.在路径 1 和路径 3 中,这 3 个操作以不同的顺序执行.分别考察 3 条路径上的关键操作可以发现,方法 *MyThread.run*()夹在了 if ($x\geqslant 0$)和 $y=f(x)$两条语句之间执行.正是这条非预期的执行路径导致程序中发生了原子性违背的错误,引发了异常.这样就找到了程序中的缺陷.

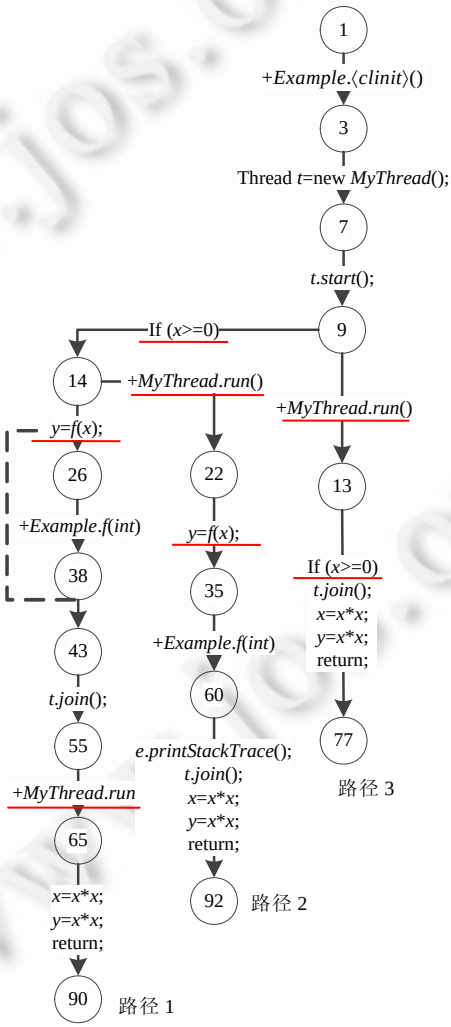


Fig.7 Analysis result

图 7 分析结果

在调试过程中,程序切片缩小了需要分析的程序语句范围,配置结构图提供了程序行为的全局视图,关键路

径的提取缩小了需要探查的路径,分支操作的标记能够引导用户关注到影响程序行为的关键操作,交互式的用户界面能够帮助用户应对更大规模的程序,从而辅助其进行程序调试.

4 相关工作

程序调试技术与程序测试、错误定位技术密切相关.在传统的程序调试场景中,自动化的测试并发程序需要在测试中尽可能地提高测试用例的代码覆盖率,尽可能多地触发错误,使得程序员观察到程序失效,进而修复程序,改进代码质量.为了克服并发程序执行的不确定性,测试过程中需要尽量多地生成线程的不同交替执行次序,以提高程序的覆盖率.这通常是通过压力测试[23]、随机增加线程停顿[24]或控制线程执行顺序[25]等方法来实现的.但测试无法确保所有的并发错误都能够被发现.并发测试工具 CHESS[26]将对线程调度的控制与搜索相结合,能够系统化地生成并发程序所有可能的线程交替,从而有效地发现并发错误.错误被触发后,进入程序调试阶段.根据调试策略可将调试技术分为以下两类,详述如下.

4.1 基于重放的人工调试

为了找到程序失效原因,通常首先需要知道怎样的执行导致了这个错误,哪些程序语句与这个错误相关.最直接的方式是通过重现错误获得错误执行路径.对于顺序程序,通过输入相同的测试用例就可以将错误重现(在相同的执行环境下).但是由于并发程序执行的不确定性,相同的输入也可能导致不同的执行结果,给重现带来了困难.为了克服这个问题,一些研究者将并发程序的调试与测试相结合,通过记录和控制并发程序的执行轨迹实现确定性的错误重放.借助此技术,程序员可以沿用传统的断点调试方法来调试并发程序.为了实现重放,首先需要记录程序的错误执行轨迹,可以通过程序运行环境和修改程序代码两种方式实现.例如,DejaVu[8]采用了基于全局时钟的方式,通过给系统中的关键事件(如同步操作和对共享内存的访问)分配一个全局时序来实现多处理器系统上的确定性重放;同时,在多线程之间给变量访问增加全局时序,使得多处理器的运行开销大大增加了.RecPlay[9]放弃了全局时钟而采用了 Lamport 时钟,通过监控线程的进入和退出事件来记录部分线程的执行时序,从而使记录和重放过程轻量化.而 Leap[10]跟踪的是每个共享变量可见的线程访问操作,从操作上比使用 Lamport 时钟的方法更简单.不同的重放方法都在追求用更少的性能代价来换取尽可能高的重放保真度.

上述方法通常发生在程序测试阶段,并不针对任何程序问题,因此并不适用于缺陷报告驱动的程序调试场景.同时,错误路径并不足以给程序员提供足够的信息进行错误定位,通过重放的方法获得了导致程序失效的执行轨迹后,程序员仍然需要人工的检测所有相关的执行语句,根据对程序的理解和并发程序的知识不断缩小搜索的范围,最终定位到导致程序失效的语句.本文的方法与之相比更具有针对性,调试的对象为与缺陷报告相关的程序代码片段.而且,与上述动态调试方法不同,本文通过静态分析展示程序行为,避免了触发并发错误的代价,并能够帮助程序员进行错误定位.

4.2 基于可视化的人工调试

为帮助程序员理解复杂的程序,可视化的内容包括程序静态结构、程序行为等.在本文中,我们关注于对并发程序行为的可视化.程序动态行为的可视化是针对某一次或多次程序执行的结果分析程序行为的方法,其过程中包括错误执行的发现、收集以及执行可视化 3 个步骤.错误执行可以通过压力测试、模型检测等方法发现,错误收集与重放方法中类似,可以通过修改程序运行环境或程序本身实现.相关工作中,执行可视化所用的程序行为视图包括 UML 时序图(sequence diagram)、消息序列图(message sequence chart)和协作图(collaboration diagram)等.很多学者通过改进 UML 时序图来使其支持对并发程序线程间交互行为的描述[6,27−29],克服了标准时序图中缺乏对线程运行状态、上下文何时切换等信息的描述.JAVAVIS[7]使用了对象图(object diagram)来描述程序状态,而 JaVis[6]还使用协作图来表示对象之间的交互.它通常用于辅助检测程序中的死锁等线程安全性问题,而对非死锁问题无效.当程序规模较大或程序交互复杂时,图形通常会变得过于复杂而不利于人工调试.与上述工作不同,Atropos[4,30]采用动态依赖图(dynamic dependency graph)来可视化程序行为,其优点是便于程序员通过后向追踪因果链的方式,从出错点找出与程序出错点相关的操作,进而找到程序缺陷.程序动态行为的

可视化方法无法提供程序行为的全局视图,只能刻画那些已被触发了的行为轨迹.

本文与上述工作的区别在于:通过静态分析的方式为程序员提供了程序行为的全局视图,避免了触发并发错误的代价.我们所采用的配置结构图既能描述线程的执行轨迹、不同线程交替之间的关系,又能反映线程到达的状态,有助于程序员对程序行为的全面理解;同时,利用事件结构视图,能够去除那些等价的执行轨迹,为程序员提供一个简单的程序视.利用配置结构图与源程序的对应关系,可以层次化地分析程序行为.而利用程序分支处的操作,可以引导用户关注影响程序行为的关键操作,尽快发现导致程序失效的缺陷,从而辅助用户进行错误调试.

5 讨 论

目前,我们对小规模的程序进行了实验,初步验证了方法的可行性.

对于大规模程序,可能面临的两个问题是:

- 程序规模较大,静态分析可能导致内存溢出;
- 程序行为视图过于复杂导致难以人工识别.

对于前者,我们可以利用偏序规约算法直接计算程序的关键路径,避免生成整个配置结构图,以减少状态爆炸的发生;对于后者,主要依赖于程序切片技术将用户关心的程序行为抽取出来,并依靠交互式图形展示方法展示复杂的图形.由于静态分析固有的特点,本文采用的切片方法存在一定的近似.而缺陷报告中的日志、堆栈信息等记录了程序运行时的部分信息,利用这些动态信息,可以精化程序切片的分析结果.另外,可以将本文的方法与断点调试技术相结合,根据用户设置的断点,生成程序在两断点之间的程序行为,并生成整个程序的行为视图,以提高方法的可用性.

6 结束语

本文针对缺陷报告导向的程序调试场景,提出了通过静态分析并发程序行为辅助程序调试的方法.此方法能够避免重现并发错误的代价,根据缺陷报告对程序进行切片,辅助错误定位,并将程序的行为展示成为简洁的图形表示.方法根据事件结构的语义,用关键路径图表示程序的全局行为,帮助用户识别潜在的线程执行顺序,并通过标记分支处的操作引导用户关注影响程序行为的关键操作,帮助其发现程序缺陷.我们已开发出针对 Java 程序的交互式并发程序调试工具 JESVis Debugger 的原型系统.如何继续改进本方法将其应用在实际开发过程中,是我们进一步研究的内容.

References:

[1] Lu S, Park S, Seo E, Zhou YY. Learning from mistakes——A comprehensive study on real world concurrency bug characteristics. In: Proc. of the 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Seattle: ACM Press, 2008. 329−339. [doi: 10.1145/1346281.1346323]

[2] Xiong JX, Wang DX, Zheng WM, Shen MM. Event-Based visualization techniques on parallel debugging. Ruan Jian Xue Bao/Journal of Software, 1996,7(5):292−299 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7/292.htm

[3] McDowell CE, Helmbold DP. Debugging concurrent programs. ACM Computing Surveys, 1989,21(4):593−622. [doi: 10.1145/76894.76897]

[4] Lönnberg J. Understanding and debugging concurrent programs through visualization [Ph.D. Thesis]. Espoo: Aalto University, 2012.

[5] Fleming SD, Kraemer E, Stirewalt REK, Dillon LK. Debugging concurrent software: A study using multithreaded sequence diagrams. In: Proc. of the 2010 IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC). Washington: IEEE Computer Society, 2010. 33−40. [doi: 10.1109/VLHCC.2010.14]

[6] Mehner K. Trace-Based debugging and visualization of concurrent Java programs with UML [Ph.D. Thesis]. Paderborn: University of Paderborn, 2005.

[7] Oechsle R, Schmitt T. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java debug interface (JDI). In: Proc. of the Int'l Seminar on Revised Lectures on Software Visualization. Berlin, Heidelberg: Springer-Verlag, 2002. 176−190. [doi: 10.1007/3-540-45875-1_14]

[8] Choi JD, Srinivasan H. Deterministic replay of Java multithreaded applications. In: Proc. of the SIGMETRICS Symp. on Parallel and Distributed Tools. Welches: ACM Press, 1998. 48−59. [doi: 10.1145/281035.281041]

[9] Ronsse M, de Bosschere K. RecPlay: A fully integrated practical record/replay system. ACM Trans. on Computing System, 1999, 17(2):133−152. [doi: 10.1145/312203.312214]

[10] Huang J, Liu P, Zhang C. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In: Proc. of the 18th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Santa Fe: ACM Press, 2010. 207−216. [doi: 10.1145/1882291.1882323]

[11] Lucia B, Wood BP, Ceze L. Isolating and understanding concurrency errors using reconstructed execution fragments. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Jose: ACM Press, 2011. 378−388. [doi: 10.1145/1993498.1993543]

[12] Bettenburg N, Just S, Schröter A, Weiß C, Premraj R, Zimmermann T. Quality of bug reports in Eclipse. In: Proc. of the 2007 OOPSLA Workshop on Eclipse Technology Exchange. Montreal: ACM Press, 2007. 21−25. [doi: 10.1145/1328279.1328284]

[13] Wong WE, Debroy V. A survey of software fault localization. Technical Report, UTDCS-45-09, Richardson: Department of Computer Science, The University of Texas at Dallas, 2009.

[14] Wu XQ, Wei J, Wang X. Debug concurrent programs with visualization and inference of event structure. In: Proc. of the 2012 19th Asia Pacific on Software Engineering Conf. (APSEC). Hong Kong: IEEE, 2012. 683−692. [doi: 10.1109/APSEC.2012.134]

[15] Weiser M. Program slicing. In: Proc. of the 5th Int'l Conf. on Software Engineering. San Diego: IEEE, 1981. 439−449.

[16] Nanda MG, Ramesh S. Slicing concurrent programs. In: Proc. of the 2000 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Portland: ACM Press, 2000. 180−190. [doi: 10.1145/347324.349121]

[17] Ranganath VP, Hatcliff J. Slicing concurrent Java programs using Indus and Kaveri. Int'l Journal on Software Tools for Technology Transfer, 2007,9(5-6):489−504. [doi: 10.1007/s10009-007-0043-0]

[18] Hansen H, Wang X. On the origin of events: Branching cells as stubborn sets. In: Proc. of the 32nd Int'l Conf. on Applications and Theory of Petri Nets. Newcastle: Berlin, Heidelberg: Springer-Verlag, 2011. 248−267. [doi: 10.1007/978-3-642-21834-7_14]

[19] Winskel G. Event structures. Petri Nets: Applications and Relationships to Other Models of Concurrency, 1987,255:325−392.

[20] Duarte LM, Kramer J, Uchitel S. Towards faithful model extraction based on contexts. In: Fiadeiro J, ed. Proc. of the Fundamental Approaches to Software Engineering. Berlin, Heidelberg: Springer-Verlag, 2008. 101−115. [doi: 10.1007/978-3-540-78743-3_9]

[21] Corbett JC, Dwyer MB, Hatcliff J, Laubach S, Pasareanu CS, Robby, Zheng HJ. Bandera: Extracting finite-state models from Java source code. In: Proc. of the 2000 Int'l Conf. on Software Engineering. New York: ACM Press, 2000. 439−448. [doi: 10.1109/ICSE.2000.870434]

[22] Hatcliff J, Dwyer MB. Using the bandera tool set to model-check properties of concurrent Java software. In: Proc. of the 12th Int'l Conf. on Concurrency Theory. London: Springer-Verlag, 2001. 39−58. [doi: 10.1007/3-540-44685-0_5]

[23] Musuvathi M, Qadeer S. CHESS: Systematic stress testing of concurrent software. In: Puebla G, ed. Proc. of the Logic-Based Program Synthesis and Transformation, Vol.4407. Berlin, Heidelberg: Springer-Verlag, 2007. 15−16. [doi: 10.1007/978-3-540-71410-1_2]

[24] Stoller SD. Testing concurrent Java programs using randomized scheduling. Electronic Notes in Theoretical Computer Science, 2002,70(4):142−157. [doi: 10.1016/S1571-0661(04)80582-6]

[25] Wang C, Said M, Gupta A. Coverage guided systematic concurrency testing. In: Proc. of the 33rd Int'l Conf. on Software Engineering. Waikiki: ACM Press, 2011. 221−230. [doi: 10.1145/1985793.1985824]

[26] Musuvathi M, Qadeer S, Ball T, Basler G, Nainar PA, Neamtiu I. Finding and reproducing Heisenbugs in concurrent programs. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. San Diego: Springer-Verlag, 2008. 39−58.

[27] Artho C, Biere A. Applying static analysis to large-scale, multi-threaded Java programs. In: Proc. of the 13th Australian Conf. on Software Engineering. IEEE, 2001. 68−75.

[28] Artho C, Biere A, Honiden S. Exhaustive testing of exception handlers with enforcer. In: de Boer F, ed. Proc. of the Formal Methods for Components and Objects, Vol.4709. Berlin, Heidelberg: Springer-Verlag, 2007. 26–46. [doi: 10.1007/978-3-540-74792-5_2]

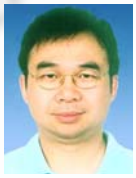
[29] Xie SH, Kraemer E, Stirewalt REK. Design and evaluation of a diagrammatic notation to aid in the understanding of concurrency concepts. In: Proc. of the 29th Int'l Conf. on Software Engineering. IEEE, 2007. 727−731. [doi: 10.1109/ICSE.2007.31]

[30] Lönnberg J, Ben-Ari M, Malmi L. Java replay for dependence-based debugging. In: Proc. of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. Toronto: ACM Press, 2011. 15–25. [doi: 10.1145/2002962.2002967]

附中文参考文献:

[2] 熊建新,王鼎兴,郑纬民,沈美明.基于事件模型的可视化并行调试技术.软件学报,1996,7(5):292–299. http://www.jos.org.cn/1000-9825/7/292.htm

伍晓泉(1984－),女,湖南石门人,博士生,主要研究领域为网络分布计算,软件工程,程序分析与调试.
E-mail: wuxiaoquan07@otcaix.iscas.ac.cn



魏峻(1970－),男,博士,研究员,博士生导师,主要研究领域为网络分布式计算,软件工程.
E-mail: wj@otcaix.iscas.ac.cn