软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(2):254−268 [doi: 10.13328/j.cnki.jos.004779]

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

基于数据链的软件故障定位方法*

杨 波[1,2], 吴 际[2], 刘 超[2]

[1](北方工业大学 计算机学院,北京 100144)

[2](北京航空航天大学 计算机学院,北京 100191)

通讯作者: 杨波, E-mail: yangbo@sei.buaa.edu.cn

摘 要: 软件中存在的故障很多与数据流紧密相关,对数据流故障定位是一个具有相当难度的研究问题.通过分析变量的定义-使用关系和变量间的依赖关系,并跟踪程序运行时各种操作对变量值的影响,即,变量操作状态的变化等基本信息,提出了一种综合考虑变量操作状态变化以及变量操作状态间依赖关系的数据链模型,利用该模型对程序中数据流故障进行定位.经过实验验证,所提出的基于数据链的故障定位方法的定位结果与基于定义-使用对、基于程序切片、基于概率依赖图和基于语句覆盖这 4 种典型的故障定位方法进行了对比,取得了更好的定位效果.

关键词: 故障定位;数据流;变量;数据链;软件测试

中图法分类号: TP311

中文引用格式: 杨波,吴际,刘超.基于数据链的软件故障定位方法.软件学报,2015,26(2):254−268. http://www.jos.org.cn/1000-9825/4779.htm

英文引用格式: Yang B, Wu J, Liu C. Software fault localization based on data chain. Ruan Jian Xue Bao/Journal of Software, 2015,26(2):254−268 (in Chinese). http://www.jos.org.cn/1000-9825/4779.htm

Software Fault Localization Based on Data Chain

YANG Bo[1,2], WU Ji[2], LIU Chao[2]

[1](College of Computer Science, North China University of Technology, Beijing 100144, China)

[2](School of Computer Science and Engineering, BeiHang University, Beijing 100191, China)

Abstract: Data flow related faults are very common in program and difficult to locate. This paper proposes a data chain model that takes account of variable change and dependencies information, and puts forward a fault location approach based on the data chain model. The main research of this paper is verified by experiments. The experimental results show that the proposed approach achieves better results comparing with other fault location approaches such as DU-pair, program slice, PPDG, and statement coverage.

Key words: fault localization; data flow; variable; data chain; software testing

对于安全关键系统而言,故障有可能带来严重的,甚至致命的安全失效.其中,程序的数据流导致的故障非常难以定位.值得注意的是,现有的故障定位研究未能充分地利用程序中变量的取值变化信息、变量间的依赖关系和程序逻辑信息.从对人工程序调试的过程进行分析可知:程序中的变量变化、变量间的依赖关系和程序的执行逻辑正是人工程序调试中重点关注的,并且这些信息与程序中的数据流故障联系紧密,而程序中的故障大多与程序中的数据流密切相关.

本文提出了一种针对 Java 程序运行时变量操作状态变化及其依赖关系的数据链模型.在此基础上,提出一种基于数据链的故障定位方法,该方法通过分析变量操作状态的转移情况和不同变量的操作状态之间的依赖情况来发现引发故障的可能位置(如程序语句).基于数据链的故障定位方法利用变量操作状态以及变量操作状

* 基金项目: 国家自然科学基金(61370051)

收稿时间: 2014-07-01; 修改时间: 2014-10-31; 定稿时间: 2014-11-26

态间的依赖关系,计算出在执行失败和执行成功场景中变量操作状态之间的转移关系以及变量操作状态出现的概率值,据此对这些变量操作状态的故障疑似度进行评级.

本文第 1 节介绍数据流故障及典型的故障定位方法,其中重点介绍利用变量信息的故障定位方法.第 2 节给出一个启发性的实例,根据该实例引出数据链,并给出数据链形式化的定义和分析.第 3 节详细阐述一种利用数据链模型进行故障定位的方法.第 4 节给出对数据链故障定位方法的评价准则,并进行实验和对实验结果进行详细的分析.最后,在第 5 节进行总结,并对后续工作进行展望.

1 数据流故障及典型故障定位方法

程序中的变量(数据对象)有可能发生状态的改变,这些状态改变序列的抽象表示称为数据流[1].变量的状态指的是变量的值,变量的状态是由程序员在其上施加的操作而改变的,对变量的操作分为定义与使用,数据流可以记录变量状态变化的轨迹.数据流故障指的是隐藏在程序中不正确的变量定义与变量使用,这些不正确的变量定义与变量使用使得变量在其数据流上存在与预期不符的状态,可能导致程序在执行时不能产生符合预期的结果.

Liu 等人认为,程序中存在两类故障.一类是导致程序崩溃的故障,这类故障会导致程序运行提前终止.通过分析程序调用栈,可以快速定位此类故障.另一类是程序设计逻辑故障,一般不会引起程序执行崩溃,只是引起程序出现逻辑错误,即,在特定的输入条件下,会导致产生错误的输出结果.逻辑故障的隐蔽性强,使得定位这类故障变得相当困难[2].数据流故障是逻辑故障中的一种,虞凯等人指出,程序中的故障包含与数据流故障和控制流故障,其中,与数据流故障包括变量的错误定义、变量的错误使用等;控制流故障则包括谓词条件的错误定义、语句执行顺序的改变等[3].本文所研究的故障主要是针对与数据流相关的逻辑性故障.实际上,这类故障也与控制流相关,尽管在特定情形下,有时其表现与控制流无关.比如对于不同的输入,程序执行的控制流完全相同,但输出结果却有的正确、有的不正确.这种情况更凸显了程序的输入结果依赖于数据流,即,输入数据及其计算逻辑.

对数据流故障进行定位,应该考虑变量的信息.目前也存在一些利用变量信息来进行故障定位的方法,Santelices 等人提出了基于定义-使用对覆盖率的故障定位方法[4],该方法通过计算程序执行时每一个定义-使用对的疑似度得分来判断可能包含故障的定义-使用对.Zeller 提出了 Delta 调试的方法[5],该方法首先通过得到程序中所有变量以及它们的值构成的集合,产生程序状态图.接着使用计算公共子图来识别执行成功与执行失败的程序状态之间的差异,从而找到导致故障的变量,通过跟踪这些变量的取值进行查找,直到找到相应的语句.

以变量为分析单元的故障定位方法没有考虑在运行时的变量操作状态变化和变量间的依赖信息.此外,上述基于程序行为特征分析的故障定位方法一般都会给出按怀疑分值降序排列的故障定位报告,这份报告可以提示程序员按照指定顺序检查可疑的程序单元.但是这样存在一个假设:程序员看到与故障相关的程序单元时,会立即发现故障.然而,该假设没有考虑程序上下文对程序员的影响.对于程序的实际开发者而言,在缺少程序上下文的提示下,也很难在短时间内根据故障定位报告找到程序中的故障.因此,这类方法对实际调试的辅助效果受到了质疑[6],并有研究专门做了理论分析[7].对于程序员来说,如果能够提供一些程序上下文的信息,则将对理解故障和帮助修复故障起到很重要的作用.

基于程序切片的故障定位方法也利用了程序中的变量信息及依赖关系.切片技术能够从整个程序中提取某个变量直接或者间接依赖的语句.在故障定位研究中,切片技术常用来减少需要分析的程序代码或者失效执行轨迹的规模.有研究表明:利用程序切片技术,可以使所分析的程序的平均规模缩减 1/3[8].虽然利用切片技术能够缩减分析的代码数量,但是当遇到大规模的软件系统时,1/3 的代码量分析起来依然非常费时.为了让切片技术更好地应用于故障定位研究中,研究人员提出了很多衍生的切片技术.此外,基于程序切片的故障定位方法在进行故障定位时,故障有可能不存在于所获得的疑似故障语句集合中,并且当故障包含在所获得的疑似故障语句集合中时,有待进一步检查的语句集合可能依然会很大[9].本文作者在此前的研究工作[10]中提出了基于数据链图挖掘的故障定位方法,这与本文所提出的故障定位方法有些类似,主要的区别在于:前者从数据链图的结

构出发寻找其中的频繁子图,没有更深层次地考虑变量操作状态间依赖关系的概率情况,且从数据链图寻找出的频繁子图会存在一些冗余.

针对程序中不同的故障进行特定的分析与定位,能够有效地提高定位的效率.Santelices 等人基于程序频谱分析了程序当中由不同元素导致的故障,最好由对应元素的程序频谱来进行分析和故障定位[4].张云乾等人对预测故障类型的可行性进行了研究,并采用马尔可夫过程对故障类型进行预测,从故障定位技术中选择出适合的技术来对不一样的故障类型进行定位[11].Chen 等人利用可以鉴别的图挖掘方法去挖掘程序的行为图,并且指出利用图挖掘的故障定位研究忽略了程序执行中的数据流信息[12].有时候,一个包含故障的控制流图和没有故障的控制流图是一样的,而不同的是这个控制流图里面数据流信息的变化.因此,如果能够考虑数据流的信息,应该能够获得更好的定位效果.

从这些研究可以看出:对特定类型的故障,采用专门的故障定位方法往往能够起到更佳的效果.程序中的故障大多与程序中的数据流密切相关,这些故障与由其他程序元素(比如谓词、函数或方法等)引起的故障所产生的行为特点是不一样的.

事实上,程序中数据流故障呈现出一些独特的特征:

(1) 有些故障不会导致错误的执行路径,但其计算结果是不正确的,即,没有导致控制流错误,但程序执行过程中某个或某些变量的值出现异常,甚至程序可能直到最终输出结果时,才表现出输出结果与预期的不符.

(2) 有些故障会导致程序中的控制流路径出现错误,但如果仅利用程序中控制流信息来进行故障定位,定位的效果并不太好[2,3,13−15].而在已有的一些利用程序中数据流或者依赖关系的故障定位研究中,定位的效果也有待改进[10,16−21].

(3) 有些故障需要经过多次计算,变量的值才会触发,比如,只有在某个循环体经过多次迭代计算之后才有可能导致某个变量值出现溢出、越界、误差过大等问题.这类现象可称为累积效应或迭代效应.这类故障往往涉及复杂类型的变量,如数组、链表以及对象类等.对这类故障进行定位时,追踪变量的操作状态变化轨迹就相当重要了.

(4) 有些故障时常会呈现出传递性,即,一个变量的错误取值会导致另一个变量的取值也出现错误,这称为变量的错误传递(或错误传播)特性.这种情况往往是变量之间的依赖关系在起作用,因此有必要在故障定位时考虑变量之间的依赖关系.

从上述的分析可知:软件的各种故障,特别是与计算结果或其控制逻辑相关的故障,大多数都与数据流相关,即,受到变量的取值变化及其相互之间依赖关系的影响.这些影响时常是彼此交织和错综复杂的,使得程序包含着各种不确定的和难以精确简单辨识的特征.因此,很有必要针对数据流故障提出相应的故障定位方法.

2 数据链模型

2.1 启发性实例

本文使用快速排序程序[22]作为启发性的实例,针对这个排序程序,设计了 5 个测试用例,分别是 t1,t2,t3,t4,t5.这 5 个测试用例的输入是数组 $a[\cdot]$的值,分别是“{5,7,8,2,1}”,“{6,8,9,4,3}”,“{1,4,9,3,5}”,“{9,1,3,2,5}”,“{5,7,8,9,4}”.此外,需要输入的数组 N 的长度为 5.

测试用例 t1,t2,t3,t4,t5 执行之后的测试判定分别为 pass,pass,pass,pass 和 fail.程序及其执行覆盖情况见表 1.故障语句位于第 7 行,其中,“$R2=R1+2$”应该为“$R2=R1+1$”.从表中可以看出:这 5 个测试用例在执行的过程中,程序中的每一条语句都被执行到了.

表 1 展示 Tarantula 方法的定位结果,其中,“●”表示程序语句被测试用例执行过.可以看出:Tarantula 方法得到的疑似故障语句集合为程序中所有语句,所有语句存在故障的可疑度和评级都是一样的,显然未能有效减小人工程序调试的工作量.

Table 1 An example of software fault localization based on Tarantula

表 1 基于 Tarantula 方法的故障定位方法举例

程序	行号	t1	t2	t3	t4	t5	可疑度	评级
Public *int sort*(*int a*[·],*int N*){		{5,7,8,2,1}	{6,8,9,4,3}	{1,4,9,3,5}	{9,1,3,2,5}	{5,7,8,9,4}	$\frac{\%fail(s)}{\%fail(s)+\%pass(s)}$	
*int R*0, *R*1, *R*2, *R*3;		●	●	●	●	●	0.5	1
*R*0=0;	1	●	●	●	●	●	0.5	1
*R*1=0;	2	●	●	●	●	●	0.5	1
*R*2=0;	3	●	●	●	●	●	0.5	1
*R*3=0;	4	●	●	●	●	●	0.5	1
While (*R*1<*N*){	5	●	●	●	●	●	0.5	1
*R*0=*a*[*R*1];	6	●	●	●	●	●	0.5	1
***R*2=*R*1+2;**	7	●	●	●	●	●	0.5	1
*R*3=*R*1;	8	●	●	●	●	●	0.5	1
While (*R*2<*N*){	9	●	●	●	●	●	0.5	1
If (*a*[*R*2]>*R*0){	10	●	●	●	●	●	0.5	1
*R*0=*a*[*R*2];	11	●	●	●	●	●	0.5	1
*R*3=*R*2;}	12	●	●	●	●	●	0.5	1
*R*2=*R*2+1;}	13	●	●	●	●	●	0.5	1
*R*2=*a*[*R*1];	14	●	●	●	●	●	0.5	1
a[*R*1]=*R*0;	15	●	●	●	●	●	0.5	1
a[*R*3]=*R*2;	16	●	●	●	●	●	0.5	1
*R*1=*R*1+1;}	17	●	●	●	●	●	0.5	1
return 0;}	18	●	●	●	●	●	0.5	1
}	–	–	–	–	–	–	–	–
测试判定	–	pass	pass	pass	pass	fail	–	–

基于定义-使用对覆盖率方法的定位结果见表 2.可以看出:该方法得到的疑似故障的语句集合也几乎是程序中所有的语句,显然也未能有效减少人工调试工作量.

Table 2 An example of software fault localization based on DU-Pair

表 2 基于定义-使用对覆盖率的故障定位方法举例

定义-使用对	t1	t2	t3	t4	t5	评级
	{5,7,8,2,1}	{6,8,9,4,3}	{1,4,9,3,5}	{9,1,3,2,5}	{5,7,8,9,4}	
(2,5,*R*1)	●	●	●	●	●	1
(2,8,*R*1)	●	●	●	●	●	1
(7,9,*R*2)	●	●	●	●	●	1
(7,11,*R*2)	●	●	●	●	●	1
(7,13,*R*2)	●	●	●	●	●	1
(7,16,*R*2)	●	●	●	●	●	1
(13,16,*R*2)	●	●	●	●	●	1
…	●	●	●	●	●	1

图 1 展示了基于 delta 调试方法的结果,delta 调试利用了两个测试用例 t1 和 t5,测试数据分别是{5,7,8,2,1}和{5,7,8,9,4}.该方法得到的结果如图 1 所示.

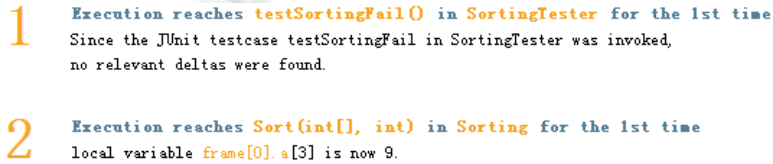

Fig.1 Delta debugging example

图 1 Delta 调试方法举例

从得到的结果很难得知程序中的故障位于语句 7,并且给出的 *a*[3]=9 也仅仅能够看出最终得到的排序后的

数组第 4 个值出现了问题,后续仍然需要花一定时间进行故障定位.

此外,利用切片的方法对该故障进行定位得出的结果也基本是全部的语句集合,难以显著地缩小需要分析的疑似语句的集合.针对目前定位数据流故障存在的问题,本文提出了一种数据链模型,并基于该模型,提出了专门针对数据流故障的定位方法.

2.2 数据链模型

2.2.1 变量和变量定义操作状态及其转移关系

(1) 变量

对于程序 S,其变量集合 V 可以表示如下:

$$V=\{v_1,\ldots,v_n\},n\in\mathbf{N}(\mathbf{N}\text{ 为自然数}).$$

每一个变量都有相应的类型:

$$V=V_o\cup V_p\cup V_r.$$

其中,V_o 为程序运行过程中所有方法或对象所创建的所有复杂类型变量,V_p 为程序运行过程中所有方法或对象所创建的简单类型变量,V_r 为程序运行过程中所有方法调用返回值对应的虚拟变量.

(2) 变量的定义与使用

- 定义(define):指变量的值被修改或者实例对象的内部状态被改变.
- 使用(use):指变量被引用,即变量的值或对象的属性值被读取,但没有任何改变,即没有被重新定义.

(3) 变量操作状态

变量状态虽然由变量的值来表征,但有时变量在程序执行过程中可能被定义多次,其值却不一定发生改变.如果仅仅考虑变量的值,将会出现许多变量值相同的情况,难以区分程序中不同位置上的操作对变量状态的影响.即便变量在程序不同的地方具有相同的值,它们其实也是不一样的.此外,程序在运行过程中会对变量进行操作(定义或者使用),这构成了一类重要的程序行为.

由于一个变量在程序运行时可被多个程序段来操作,在故障定位中,为了对变量细微的变化进行准确追踪和分析,需要准确描述程序在运行时变量的动态操作行为.本文将程序对变量的操作状态定义为“程序在何处第几次对变量进行了何种操作”.在这里,何处指的是对程序变量进行操作的位置,第几次指的是程序运行过程中在该处变量被(重复)定义的次数.由于对一个变量的使用不会改变相应变量的取值,因而这种操作本身不会使程序产生逻辑问题,故本文的变量操作状态重点关注变量定义操作状态.但另一方面,程序在对一个变量定义时往往需要使用其他变量(包括该变量自身)的取值,因此,本文还需关注在对一个变量进行定义操作时使用了哪些变量的取值,以便确定变量操作状态之间的依赖关系.

(4) 变量定义操作状态

变量定义操作状态是在程序执行过程中对变量进行定义操作的一种抽象描述,涉及变量、变量在被定义前的值、变量被定义后的值、变量在被定义时所依赖的变量等信息.从另一个角度看,变量的取值变化是由程序中对该变量的所有定义操作状态序列决定的.在程序执行中,一旦变量到达某个变量定义操作状态(即在程序的某个位置被定义),则后续对该变量的任何使用只能使用在该状态被定义的值.因此,程序在一次执行中的变量定义操作状态序列可以刻画程序如何按照一定的顺序来定义相关变量的取值,从而针对给定输入进行计算来得到最终的输出结果.需要特别说明的是:由于循环的存在,程序在某个位置可能会对某个变量进行多次定义,显然每次操作都会给该变量赋予不同的值,从而使其具有语义.为了区分程序对变量的每一次定义操作,本文引入一个计数属性来刻画变量定义操作状态,即,它表示“程序在某处对变量进行了几次定义操作”.

根据上面的阐述,可以将变量定义操作状态形式化地定义为 $P\times V\times N$,其中,P 为程序中所有的语句,V 为程序运行时的所有变量,N 记录程序对所有变量的定义操作计数(N 也为自然数集).(l,v,n)是 $P\times V\times N$ 中的任意一个元组,指的是程序在位置 l 对变量 v 进行的定义操作,且该操作是自程序运行以来在 l 处进行的第 n 次操作.由于在进行故障定位和时需要使用这些变量定义操作状态中的相关信息,所以本文定义了如下 6 个作用于变量定义操作状态的函数:

- 变量取值函数 u:$P\times V\times N\to S(V)$.
- 变量定义依赖函数 d:$P\times V\times N\to H(P\times V\times N)$.
- 变量相同性判定函数 q:$(P\times V\times N)\times(P\times V\times N)\to\{0,1\}$.
- 变量定义相邻性判定函数 r:$(P\times V\times N)\times(P\times V\times N)\to\{0,1\}$.
- 变量定义轨迹函数 t:$H(P\times V\times N)\times V\to H(P\times V\times N)$.
- 变量定义轨迹序号函数 s:$P\times V\times N\to N$.

(5) 变量定义状态转移

给定程序的任意两个定义操作状态(l_i,x,n_i)和(l_j,y,n_j),$(n_i\neq n_j)$.如果 $q((l_i,x,n_i),(l_j,y,n_j))=0$ 且 $r((l_i,x,n_i),(l_j,y,n_j))=0$,则可知 x 和 y 是相同变量,且这两次定义操作之间没有对 x 和 y 进行定义,此时认为(l_i,x,n_i)与(l_j,y,n_j)之间存在状态转移关系.如果$(n_i<n_j)$,则迁移的源状态为(l_i,x,n_i),目标状态为(l_j,y,n_j);反之,迁移的源状态为(l_j,y,n_j),目标状态为(l_i,x,n_i).

(6) 变量定义状态转移轨迹

基于程序运行过程的变量定义状态转移分析可以获得程序对某个变量 v 的定义状态操作轨迹,即,向量序列$\langle(l_1,v,n_1),\ldots,(l_m,v,n_m)\rangle$.其中,轨迹中的第 1 个状态为$(l_1,v,n_1)$,最后一个状态为$(l_m,v,n_m)$.

变量定义操作状态转移轨迹也可以记为 $trace(v)=\{(l_i,v,n_i)|n_1\leqslant n_i\leqslant n_m\}$,显然,$trace(v)$中任意两个顺序相连的状态对变量 v 的定义是相邻的.

2.2.2 变量定义操作状态依赖关系

(1) 变量依赖关系

变量间的依赖关系:给定变量 x 和 y,如果对 x 定义时需要使用 y,则称变量 x 依赖 y.从程序代码角度来看,在一个赋值语句中,赋值操作左侧的变量对赋值操作右侧的变量形成依赖关系.也存在特殊形式的赋值操作,如函数调用中的“输出型”实参,在函数返回时会对相应的参数进行赋值.

变量依赖关系反映了程序中变量值发生变化与哪些变量的值有关,或者说依赖于哪些变量,这种依赖关系主要对应于赋值运算或与其等效的运算.而在数据链模型中,与之对应的是变量定义操作状态之间的依赖关系.

(2) 变量定义操作状态依赖关系

对于程序的任意两个定义操作状态(l_i,x,n_i)和(l_j,y,n_j),如果$(l_i,x,n_i)\in d(l_j,y,n_j)$,则称$(l_j,y,n_j)$依赖于$(l_i,x,n_i)$.根据变量定义操作状态之间的依赖关系,可以推导出相应变量间的依赖关系.给定两个变量 x 和 y,它们之间的依赖关系可以定义为

$$link(x,y):link(x,y)=\{((l_i,x,n_i),(l_j,y,n_j))|((l_i,x,n_i)\in d(l_j,y,n_j))\} \tag{1}$$

其中,(l_i,x,n_i)为依赖关系的前驱,(l_j,y,n_j)为依赖关系的后继.

变量可以“自依赖”,可以将变量依赖关系看作有两种:第 1 种是不同变量之间的依赖关系,如对于 $c=a+b$,c 依赖于 a 和 b;第 2 种是变量自身的依赖关系,如 $a=a+1$.在式(1)中,变量 a 自身的值之间产生了依赖关系,但是这种变量的自依赖反映在其状态上,则表现为当前状态(或新建状态)对其前一个状态的依赖.所以,变量定义操作状态不存在自依赖,这种变量的自依赖关系可表示如下:

$$link(x,x)=\{((l_i,x,n_i),(l_j,y,n_j))|((l_i,x,n_i)\in d(l_j,y,n_j))\} \tag{2}$$

这是一种特殊的程序变量依赖关系,反映了变量 x 自身发生改变的过程.为了表示方便,$link(x,x)$有时也简写为 $link(x)$.

2.2.3 数据链模型

在程序的一次执行过程中,程序在每个变量上的定义操作状态轨迹以及各个变量定义操作状态之间依赖关系共同形成了图,称为变量定义操作状态之间的转移-依赖关系链.这就是本文所提出的数据链模型,简称为数据链.

对于给定的一次程序执行 E_i,其执行中涉及到的变量集合为 $V_i=\{v_1,\ldots,v_n\}$,其变量定义操作状态转移轨迹可以表示为 $trace(V_i).trace(V_i)$可由如下公式获得:

$$trace(V_i)=trace(v_1)\cup\ldots\cup trace(v_n) \tag{3}$$

各变量定义操作状态之间的依赖关系可以表示为 $link(V_i,V_j)$,$link(V_i,V_j)$可由如下公式获得:

$$link(V_i,V_j)=\bigcup link(v_i,v_j),v_i\in V_i,v_j\in V_j \tag{4}$$

其数据链 $CM(V_i)$可以表示如下:

$$CM(V_i)=(Node,Edge) \tag{5}$$

其中,

- *Node* 是顶点集合,$Node=\{\langle(l_1,v_1,m_1)\rangle,\ldots,\langle(l_n,v_n,m_n)\rangle\}$;
- *Edge* 是边集合,$Edge=\{(a,b)|(a,b)\in trace(V_i)||(a,b)\in link(V_i,V_j)\}$.

2.2.4 构建数据链模型

基于数据链的相关定义,本文提出了相应的数据链生成算法,如图 2 所示.该算法用来构建程序中需要进行分析的方法所对应的数据链,其中用到了变量定义操作状态的几个函数.该算法主要由两个步骤组成,分别是获取变量定义操作状态的依赖关系和获取变量定义操作状态转移.

算法. 数据链生成算法.
输入:*DataFlowInfo*,*MethodInfo* //静态数据流信息,方法信息.
输出:*CM*(*V*). //数据链
过程:

```
1.  exLines=MethodInfo.getExLines(·);
2.  m_o=MethodInfo.getMethodName(·);
3.  m_o.defs=DataFlowInfo.getMethod(m_o).getDefs(·);
4.  defs=m_o.defs;
5.  defs_all;  //临时全局变量,用来存储方法中所有定义操作状态
6.  int i=0;
7.  foreach exLines!=null && exLines.hasNext(·)  //对方法中语句进行遍历
8.    foreach defs!=null && defs.hasNext(·)  //对方法中定义变量进行遍历
9.      if defs.getL(·)==exLines.next then
10.       def=defs.next;  //def 为临时变量,为 defs 中的元素
11.       defs.setN(i);  //为定义变量设置操作计数
12.       i++;
13.       defs_all.add(def);  //存储变量定义操作状态
14.       defs_temp=d(def);  //获得 def 所依赖的定义操作状态,def_temp 为临时变量
15.       CM(V).Node.add(def);  //存储 def 所对应的变量定义操作状态
16.       foreach defs_temp!=null && defs_temp.hasNext(·)  //遍历 def 的依赖变量
17.         CM(V).Edge.add(def,defs_temp.next,0);  //存储依赖关系,0 表示依赖边
18.       end for
19.   end for
20. end for
21. defs_all_temp=defs_all;  //临时变量
22. foreach defs_all!=null && defs_all.hasNext(·)  //对所有变量定义操作状态进行遍历
23.   def=defs_all.next;
24.   foreach defs_all_temp!=null && defs_all_temp.hasNext(·)
25.     def_temp=defs_all_temp.next;
26.     if (q(def,def_temp)==0 && r(def,def_temp)==0) then  //判断是否存在转移
27.       if def.getN(·)<def_temp.getN(·) then  //判断源状态和目标状态
28.         CM(V).Edge.add(def,def_temp,1);  //存储转移,1 表示转移边
29.       else then
30.         CM(V).Edge.add(def_temp,def,1);  //存储转移,1 表示转移边
31.   end for
32. end for
33. return CM(V);
```

Fig.2 Build data chain model algorithm

图 2 构建数据链模型算法

获取变量定义操作状态的依赖关系位于第 7 行~第 20 行,其中,第 10 行和第 11 行是为变量定义操作状态设置操作计数,在静态数据流信息中,所有的变量定义操作状态都默认是 0,根据程序执行语句的先后顺序,分别

为静态数据流信息中的变量定义操作状态的操作计数赋值;从第 13 行~第 16 行是寻找每一个变量定义操作状态所依赖的变量,并将变量定义操作状态与所依赖的变量定义操作状态进行存储.

获取变量定义操作状态转移位于第 22 行~第 32 行,其中,第 26 行用例判断两个变量定义操作状态是否指向同一个变量且两者是相邻的定义操作状态,第 27 行则对这两者谁是源状态和目标状态进行判断,第 28 行和第 30 行是对变量定义转移进行保存.

在进行数据收集的过程中,为了确保收集数据链数据的准确性,本文利用了静态分析和动态监控相结合的方法.静态分析采用的是 Jimple[23]语法制导的分析方法,通过静态分析可以分析出变量、语句和表达式的语法,并能收集施加在变量上的取值和引用序列.基于 Jimple 语法制导的分析是识别变量操作的最主要途径,这是静态数据收集的基础.此外,通过静态分析可以获得程序的控制流程图、数据流信息以及数据流中变量的类型信息.在此基础上,可以得到变量的静态引用关系、即变量的定义-使用关系、变量所在的行号等静态信息.除了对程序进行静态分析之外,还需要用到动态监控的办法来对程序运行时的信息进行处理.动态监控能够得到程序在运行时的轨迹信息,还能得到变量定义操作状态轨迹、确切的变量引用关系等信息.

3 基于数据链的故障定位方法

3.1 层次化分析方法

本文采取的层次化分析方法对故障进行定位,即,将定位工作分为两个阶段:首先,采用基于关联规则的挖掘方法[24,25]来进行方法级别的故障定位;然后,在方法内部采用概率数据链模型进行语句级别的故障定位.由于程序中的方法与语句在执行过程中呈现出来的行为不一样,如果只考虑语句的行为而忽略了方法级别的行为特征,会丢失很多有用的程序语义信息.同时,如果对整个程序使用数据链模型,从语句级去考虑整个程序执行中出现的所有变量,则变量的状态空间必然会面临过大的问题.此外,程序中变量都有各自的作用域,对于全局变量等超出方法范围的变量,由它们导致的故障往往也是在方法中被触发.因此,本文采用了层次化分析方法,从方法和语句两个层次来实施故障定位.在方法这个层次,本文采用的是基于关联规则的挖掘方法来找到可疑的方法集合,随后需要对方法中变量的数据链进行分析.

在收集到程序运行时的方法轨迹之后,需要对方法轨迹进行分析,从而找到其中可能存在故障的方法.这些方法定义在程序中某个类中,但也有可能出现在多个类中.需要说明的是:本文利用方法轨迹进行故障定位时,假定在多数情况下,执行失败的测试用例在运行过程中产生的方法轨迹与执行成功的测试用例在运行过程中产生的方法轨迹是不一样的.当然,也存在执行失败的测试用例在运行过程中产生的方法轨迹与执行成功的测试用例在运行过程中产生的方法轨迹没有差别的特殊情况,这个时候得到的疑似故障方法的集合将会包括程序执行过程中所有的方法.如果出现这种情况,通常可以有针对性地增补一些特殊的测试用例来产生有差异的方法轨迹.

3.2 变量故障疑似度分析

对变量故障疑似度进行分析时,可以利用数据链模型对每一个故障疑似方法建立其数据链.利用疑似故障方法集合的数据链,可以构建出概率数据链模型,其目的在于:通过计算每个变量定义操作状态分别在执行失败的测试用例与执行成功的测试用例中的概率值来计算变量定义操作状态的故障疑似度.

在给出具体的变量故障疑似度分析方法之前,先给出变量定义操作状态轨迹异常度和变量定义操作状态疑似度的概念.变量定义操作状态轨迹异常度 $abnormal(l,v,n)$是指变量定义操作状态在其操作状态轨迹中出现异常的程度.其计算公式如下:

$$\frac{\%fail(l,v,n)}{\%fail(l,v,n)+\%pass(l,v,n)} \tag{6}$$

其中,$\%fail(l,v,n)$和$\%pass(l,v,n)$分别表示变量定义操作状态(l,v,n)为出现在所有执行失败的测试用例和所有执行成功的测试用例中的比例.变量定义操作状态轨迹异常度的值越大,表示该变量定义操作状态异常的可能性