一种 Java 程序 Chopping 方法

蒋曹清[1,2] 应 时[1] 倪友聪[1] 管 华[1]
(武汉大学软件工程国家重点实验室 武汉 430072)[1] (广西财经学院计算机与信息管理系 南宁 530003)[2]

摘 要 程序 Chopping 对于程序理解、分析、调试、测试等具有重要的意义。已有的 Chopping 方法主要基于相互连接的系统依赖图(SDG),对于大程序这种 SDG 描述通常非常复杂,易导致程序 Chopping 结果不准确。针对这一问题,基于带标签的 Java 程序描述方法,提出一种 Java 程序 Chopping 方法。该方法利用这种描述附带的程序依赖信息,分析参数依赖关系,并在此基础上给出 Chopping 算法。此程序 Chopping 方法能把 Java 程序方法间的程序 Chopping 问题转换到 Java 程序方法内进行分析,程序依赖图具有结点少、可重用、可并发构造等优点。最后通过实例和实验给出程序 Chopping 方法的实施过程及有效性。

关键词 程序 Chopping,Chopping 算法,Java 程序,Chopping 方法
中图法分类号 TP311.5 **文献标识码** A

Program Chopping Approach for Java Program

JIANG Cao-qing[1,2] YING Shi[1] NI You-cong[1] GUAN Hua[1]
(State Key Lab of Software Engineering, Wuhan University, Wuhan 430072, China)[1]
(Department of Computer and Information Management, Guangxi University of Financial and Economics, Nanning 530003, China)[2]

Abstract Compared with program slicing, program chopping is a more focused approach of program analysis. And it is of great significance for the program understanding, analysis, debugging, testing. The existing chopping algorithms mostly are based on connected System Dependence Graph(SDG). However, the representation of SDG is very complicated, especially for larger program, which causes fault result easily. So this paper proposed a program chopping approach for Java program based on a representation for Java program with tags. In this approach, not only the accessory information on program dependency relation is utilized to analyze parameter dependency relation, and based on which this paper presented program chopping algorithms, but also inter-method program chopping can be converted to intra-method analyzing. Moreover the approach has much strongpoint such as fewer nodes of SDG and reusability of Program Dependence Graph(PDG), etc. Finally, this paper illustrated implementation process of this approach combined with an example and experiment.

Keywords Program chopping, Chopping algorithm, Java program, Chopping approach

程序 Chopping 是程序切片技术的推广,能在程序中找出从某一特定语句 s 到另一特定语句 t 由于传递性依赖而涉及到的语句。程序切片能识别程序中可能影响一个给定语句 p 的所有语句,但不能回答切片中某特定语句 q 如何影响到这一给定语句 p 这个问题。为此 Jackson 等提出了 Chopping 技术[7-10,13]来解决此问题,从而提供了获得受到语句影响的更准确、更精简的语句集的方法,它对于程序理解、分析、调试、软件测试、软件维护等具有重要的意义。

1994 年,Jackson 和 Rollins 首先给出了过程内 Chopping 算法[7];后来 Reps 和 Rosay 给出了过程间 Chopping 算法[8];Krinke 在文献[9]中总结、评估了以前出现的几种 Chopping 算法,并给出了一个更有效的 Chopping 算法。已有的计算 Chop 的方法一般基于相互连接的系统依赖图(SDG)[14],通过在 SDG 上遍历,获得程序 Chop。但是这种 SDG 描述非常复杂[5],特别是大程序,从而容易导致在构造 SDG 的过程中出错和程序 Chopping 结果不准确。

针对以上问题,本文基于文献[6]介绍的一种简洁的 Java 程序描述方法(不妨称为 JavaSDG),提出一种 Java 程序 Chopping 方法。JavaSDG 是由一组带有标签且互不相连的过程依赖图(PDG)组成的 Java 程序描述方法。程序中的方法通过参数之间的依赖关系相互影响,内部数据和语句之间的依赖关系在外部是不可见的;每个 PDG 是一个独立的图,

到稿日期:2010-03-05 返修日期:2010-05-20 本文受国家自然科学基金项目(60773006)和高等学校博士学科点专项科研基金项目(20060486045)资助。

蒋曹清(1973—),男,博士生,讲师,主要研究方向为软件测试与程序分析、软件体系结构,E-mail:jcqng@163.com;应 时(1965—)男,博士,教授,博士生导师,主要研究方向为面向对象软件工程方法、基于组件的软件工程方法、软件体系结构和模式、软件的可重用性与互操作性等;倪友聪(1976—),男,博士生,讲师,主要研究方向为软件体系结构和面向方面软件开发;管 华(1978—),男,博士生,讲师,主要研究方向为软件体系结构。

不与外部连接。在此描述基础上，本文首先分析 Java 程序调用点参数之间的依赖关系，然后给出计算 Java 程序的参数 Chop、方法间 Chopping 算法。参数 Chop 只考虑与 Chopping 准则有关的参数，求与这些参数有关的语句以计算被调用方法的 Chop。方法间 Chopping 通过不断地把被调用方法的参数 Chop 合并到调用方法的 Chop 中，来达到计算所求 Chop 的目的。这样就能够把方法间的依赖分析和 Chopping 转换到方法内分析和 Chopping。接着用一个实例说明如何在一个程序中求给定 Chopping 准则的程序 Chop。然后给出用本文方法计算程序 Chopping 的结果并与已有的方法进行对比。

本文第 1 节简要介绍 Chen 和 Xu 的 JavaSDG 描述方法，为分析程序依赖性关系和分析 Chopping 算法提供依据；第 2 节首先在 JavaSDG 上分析调用点参数间的依赖关系，为 Chopping 算法提供思路；然后在 JavaSDG 描述和依赖关系分析的基础上给出本文的 Chopping 算法；第 3 节给出实例与实验结果，以说明本文方法的实施过程和有效性；第 4 节比较本文 Chopping 方法与已有的方法，以说明这两种方法的异同及本文方法的优点；第 5 节总结相关研究工作；最后总结全文。

1 Java 程序描述方法 JavaSDG

为了有效地描述 Java 程序，并为以下参数间依赖关系分析和计算 Chop 提供基础，我们需要采用文献[6]中的程序描述方法 JavaSDG。

JavaSDG 是由类和方法的 PDG 组成的集合。一个类的 PDG 是由它的所有方法的 PDG 组成的集合。JavaSDG 也能表示面向对象的特征，包括类、继承、对象、多态等，具体细节请参考文献[6]。为了方便分析参数间依赖关系和计算程序 Chop，下面将详细介绍 PDG。

在 Java 中，方法可以由控制流图(CFG)描述。一个方法的 CFG 是一个有向图，$CFG=(S,E,Sentry,Sexit)$。这里 S 是结点集；E 是边集，结点表示语句或谓词，$E=\{\langle s1,s2\rangle \mid s1, s2\in S$，并且 $s1$ 执行后可能执行 $s2\}$；$Sentry$ 是方法的入口结点；$Sexit$ 是方法的退出结点。如果 $\langle s1,s2\rangle\in E$，那么 $s1$ 是 $s2$ 的前驱，$s2$ 是 $s1$ 的后继。

为描述语句之间的依赖关系，现形式化定义以下集合：

定义 1 设 s 是程序 CFG 中的任一结点，则

(1)定义集 $Def(s)$ 表示语句 s 中被定义(修改)值的变量；

(2)引用集 $Ref(s)$ 表示在语句 s 中引用但没有修改值的变量；

(3)$Def(s,x)$ 表示在语句 s 定义变量 x 所引用的变量；

(4)数据定义依赖集 $Dep\text{-}D(s,x)=\{(x,s',y)\mid x\in Def(s)\wedge y\in Def(s')\wedge y\in Def(s,x)\wedge$ 存在一条 s' 到 s 的路，并且在这条路上没有被重新定义$\}$；

(5)数据引用依赖集 $Dep\text{-}R(s)=\{(x,s',x)\mid$ 如果 s 是一条谓词语句(例如 if，while 语句)，x 是 s 处使用的条件变量，$x\in Def(s')\wedge x\notin Def(s)\wedge$ 存在一条 s' 到 s 的路，并且在这条路上没有被重新定义$\}$。

在以上定义的基础上，可以在一般的 CFG 上添加标签，得到本文 Java 程序描述 JavaSDG 的基本描述单元 PDG。具体见定义 2。

定义 2 一个方法 M 的 PDG 是一个带有标签的有向图，$PDG=(S,E,T)$。这里结点集 S 是由入口结点 $entry$、语句结点、谓词结点组成的边集 $E=E1\cup E2$，其中 $E1$ 表示有向控制依赖边集，$E2=\{\langle s1,s2\rangle \mid (x,s1,y)\in Dep\text{-}D(s2,x)\vee (x,s1,x)\in Dep\text{-}R(s2)\}$ 表示有向数据的依赖边集；T 是一个标签集，边 $\langle s1,s2\rangle$ 上标签通过以下形式定义：

如果 $\langle s1,s2\rangle\in E1$，那么标签为$(*,*)$；

如果 $\langle s1,s2\rangle\in Dep\text{-}D(s2,x)$，那么标签为$(y,x)$；

如果 $\langle s1,s2\rangle\in Dep\text{-}R(s2)$，那么标签为$(x,x)$。

以上介绍了一种 Java 程序描述的新方法 JavaSDG，它在一般 Java 程序描述 SDG 基础上添加了表示依赖关系的标签，将作为以下介绍的 Java 程序 Chopping 方法的基础。

2 Java 程序 Chopping 方法

本文提出的 Java 程序 Chopping 方法，其主要步骤为：首先基于 JavaSDG 分析调用点参数间依赖关系；然后在 JavaSDG 描述和依赖关系分析的基础上按照以下介绍的 Chopping 算法计算程序 Chop。

2.1 调用点参数间依赖关系分析

为了方便介绍本文程序 Chopping 算法，需要分析 Java 程序调用点参数间的依赖关系。

在 Java 中，参数能通过值或引用传递。在本文中，值传递的参数叫做 in 参数。对于引用传递的参数，如果在被调用函数中仅被写，它是一个 out 参数；如果仅被读，它是一个 in 参数，否则是 in-out 参数。非局部变量可看作一个形参与实参同名的参数，如果在执行过程中此变量的值可能改变，则作为一个 in-out 参数，否则作为 in 参数。return 值可处理为一个 out 参数，而且名字与方法名相同。总之，有 3 种类型的参数：in 参数、out 参数、in-out 参数。因为 in 参数仅能读，out 参数仅能写，in-out 参数能读能写，故只有 in-out 参数和 out 参数可能对外部调用者有影响。这样，我们可以独立地分析每个被调用方法的 out、in-out 参数来获得其参数之间的依赖关系。

在以前的 SDG 中使用 Summary 边描述调用点参数之间依赖关系，而 JavaSDG 使用参数依赖集描述调用点参数之间依赖关系。这里，参数依赖集包含单个参数的参数依赖集和方法的参数依赖集，具体见定义 3。

定义 3 设 y 是方法 M 中 out、in-out 参数，它的参数依赖集可表示为 $Dep\text{-}P(M,y)=\{x\mid x$ 是参数，且 y 依赖于 $x\}$；方法 M 的参数依赖集可表示为 $Dep\text{-}P(M)=\{(y,Dep\text{-}P(M,y))\mid y$ 是方法 M 中 out、in-out 参数$\}$。

方法对外只需要提供方法依赖集 $Dep\_P(M)$，它内部的数据依赖关系对调用程序是隐蔽的。这样处理的优点主要有：(1)如果改变了某一方法，只需要重新分析它的 PDG 图，如果其参数间的依赖关系发生变化，则继续分析它的所有调用点；(2)在计算程序 Chop 时，方法之间可并行处理。

对于在语句 s 中含有被调用的方法，根据形参和实参之间的关系能够获得集合 $Def(s)$ 和 $Def(s,x)$，算法见文献[6]。经过以上处理后，消息驱动的调用语句能像其它语句一样容易处理。

在分析方法 M 的参数间依赖关系时，还需要知道被 M 调用的方法的参数之间依赖关系。这些被调用方法应该用给定的调用顺序进行分析，这能通过获取的调用图[11,12]得到。

Java 是一种静态类型的面向对象语言，对象的可能类型能静态地确定。通过参数依赖集来表示多态，每个集合和一个对象类型相对应。当一个对象调用一个方法时，根据对象类型是否已确定，分两种情况计算参数间依赖集。

情况 1　对象类型已确定

如果一个方法 M 不是多态方法，那么可采用上述的方法分析方法依赖集 $Dep\_P(M)$。否则，如果有必要的话，就把多态方法转换到相应的方法，重新分析多态方法的参数间依赖关系。

情况 2　对象类型未确定

如果一个方法不是一个多态方法，那么参数间依赖关系是所有可能对象类型的相应方法的方法依赖集的并集。否则对于所有可能对象类型，把多态方法转换为相应的方法，并重新分析依赖关系，参数间依赖关系是可能对象类型的相应方法的方法依赖集的并集。

以上从描述参数类型开始，得出分析参数间依赖关系的可行性。然后定义了参数依赖集来描述方法参数之间的依赖关系，并说明参数依赖集对于调用点参数间依赖关系分析的优点。接着基于参数依赖集来解释如何分析调用点参数依赖关系。最后针对 Java 程序中调用点可能出现多态性，讨论了如何分析此情况下的参数间的依赖关系。这将为以下的程序 Chopping 算法提供思路。

2.2 Java 程序 Chopping 算法

下面将讨论如何基于 JavaSDG 描述及方法参数间依赖关系计算 Java 程序的 Chop。一般程序 Chopping 算法分为方法内 Chopping 算法和方法间 Chopping 算法。而本文介绍的计算 Java 程序 Chop 的算法分为方法内 Chopping 算法、参数 Chopping 算法和方法间 Chopping 算法。由于方法内 Chopping 算法与已有的算法相似，它是一个利用方法 PDG 实现带有标签的图可达性问题，故在这里不再列出，可参考文献[9]。下面分别详细介绍参数 Chopping 算法和方法间 Chopping 算法。

2.2.1　参数 Chopping 算法

为了计算方法间程序 Chop，本文需要使用一个特定种类的 Chop。本文把这种 Chop 称为参数 Chop，其定义如下。

定义 4　设 M 是调用方法，M' 是被 M 调用的方法，c 是 M 的调用点且 c 所在的语句 s 在 M 的 Chop 中，那么对于受 s 影响且在 M 的 Chop 中的任一语句 w，被 w 引用的 out 或 in-out 参数 x 是 $Sexit$ 结点仅要考虑的参数；$z\in Dep\_P(M',x)$ 是 $Sentry$ 结点仅要考虑的参数，如果以这种情形下的〈$Sentry$，$Sexit$〉为 Chopping 准则，则所得的 Chop 称为 M' 的参数 Chop。

求参数 Chop 的详细算法见算法 1。

算法 1　参数 Chopping 算法

```
1  Let G=(V,E) is given JavaSDG
2  Let M is calling method,M' is called method
3  Let c is a call site corresponding M' in M
4  Let statement s which include c exist in chop of M
5  Init Let WB=Φ,WF=Φ,C'=Φ
6  if(no exist(〈s,w〉∈E∧w exist in chop of M))
7    M'=M
8    if(M'==main) return C'=Φ
9    do
10     Let N is a calling method,N' is called method
11     Let c' is a call site corresponding N' in N
12     Let s' is statement which include c'
13     N'=N
14     if(N'==main) return C'=Φ
15   while(no exist(〈s',w'〉∈E∧w' exist in chop of N))
16   if(w' no exist in chop of M)
17     Let w' is successor of s
18     Tag of 〈s,w'〉=tag of 〈s',w'〉
19   foreach 〈s,w〉∈E∧w exist in chop of M
20     Let T is a tag on 〈s,w〉
21     foreach T on 〈s,w〉
22      if T=(x,x')∧x'∈def(w) ∧x is out or in-out parameter
        then
23       foreach z∈dep-P(M',x)
24         if exist(〈u,s〉∈E∧u exist in chop of M∧tag of 〈u,s〉 is
           (z',z) then
25         WB= WB∪{〈Sexit,x〉}
26         WF= WF∪{〈Sentry,z〉},//backward phase
27          while WB≠Φ
28            Remove an element 〈v,y〉 from WB
29            foreach edge 〈v',v〉 which is not marked
30             Mark 〈v',v〉 as visited in the backward phase
31             Let T' is a tag on 〈v',v〉
32             if T'=(y',y) //data dependence
33               WB= WB∪{(v',y')}
34             if T=(*,*) then // control dependence
35               foreach yi∈Ref(v')
36                 WB= WB∪{(v',yi)}
       //forward phase
37     if Sentry has been marked in the backward phase then
38      C=C∪{ Sentry}
39       while WF≠Φ
40         Remove one element 〈v,y〉 from WF
41         foreach edge 〈v,v'〉
42          if 〈v,v'〉 has been marked in the backward phase then
43            Mark 〈v,v'> as visited in the forward phase
44            Let T' is a tag on 〈v,v'〉
45            foreach T' on 〈v,v'〉
46             if T'=(y,y') then //data dependence
47               WB= WB∪{(v',y')},C'=C'∪{v'}
48             if T'=(*,*) then //control dependence
49               WB= WB∪{(v',*)},C'=C'∪{v'}
50 return C'
```

从算法 1 可以得出计算参数 Chop 的主要步骤如下：①算法 1－5 行，实现初始化；②6－18 行，寻找计算参数 Chop 的条件（如果被调用点在整个程序中没有在 Chop 中的后继语句，则不必求参数 Chop 值）；③19－26 行，为计算 Chop 做准备，如初始化 WB 和 WF；④27－36 行，通过后向遍历实现求 Chopping 准则(s,t)的目标准则 t 的后向切片；⑤37－49 行，在后向切片（在切片语句上作“已访问”标志）的基础上通过前向遍历实现求 Chopping 准则(s,t)的源准则 s 的前向切片，则在后向切片和前向切片中都存在的语句即为 Chop 中的语句；⑥50 行，C'即为所求的参数 Chop。

2.2.2　方法间 Chopping 算法

在计算方法间 Chop 时，内部依赖对外部是不可见的，没

有边连接实参和相应形参，所以在用被调用方法分析语句时，需要求实参对应形参的参数 Chop。这样，被调用方法的 Chop 被转换到在参数间计算参数 Chop。

方法间 Chopping 的主要任务是决定 Chopping 准则需要考虑的参数，以计算被调用方法的参数 Chop。算法 2 给出了 Chopping 准则(s,t)都在一个方法内的 Chopping 算法。否则需要分析调用图，判定是否存在某个 s 与 s 所在的方法中的某个调用点 c，c 能通过调用关系联系上另一个 t。如果存在这样的 c，则可以(s,c)为 Chopping 准则，这样 s、c 就在同一方法内；否则说明以(s,t)为 Chopping 准则的程序 Chop 结果为空集。因此，这里假设(s,t)在同一方法内，从本质上不影响本文所提方法的一般步骤。由于篇幅限制，(s,t)不在同一个方法内的情况可由读者自己完成。

算法 2 方法间 Chopping 算法

Let G=(V,E) is the given JavaSDG

Let (s,t)∈V×V is the given chopping criterion,s,t∈M

Let C is the intraMethod chop for(s,t) in M

Let Call is the M's call sites in C except callsite of s,t if s,t be in C and has call sites

while Call≠Φ

  Remove a call site c from Call

  Let M′ is the to c corresponding method

  Let C′ is parameter chop of M′

  C=C∪C′

  Call=Call∪call sites of M′

Return C

在算法 2 中，如果在一条语句中有一个被调用的方法，首先把这个语句看作没有调用方法的语句，然后把调用方法看成一个简单方法并计算 Chop，最后在被调用方法中计算相应的参数 Chop，并把它并到调用方法的 Chop 中。本文算法与已有方法间 Chopping 算法主要的不同之处是：已有算法[9]一般利用 Summary 边信息计算被调用方法的 Chop，然后与调用方法的 Chop 求并集；本文算法利用参数间的依赖信息计算参数 Chop，然后与调用方法的 Chop 求并集，而且参数 Chop 可保存起来，当下次需要时可以重用这些结果。

总之，根据本部分的描述，本文介绍的程序 Chopping 算法由 3 部分组成：(1)方法内程序 Chopping 算法；(2)参数 Chopping 算法；(3)方法间 Chopping 算法。计算方法内程序 Chop 时可并发进行，且可重用；参数 Chop 在方法间起到连接的作用，能够把计算方法间 Chop 问题转化为计算相关方法的方法内程序 Chop 的并集。

3 实例分析与实验结果

以上第 1 和第 2 节内容介绍了与本文提出的 Java 程序 Chopping 方法有关的内容，并在第 2 节总结了本文提出的 Java 程序 Chopping 方法的一般步骤。下面通过实例和实验结果说明本文方法的过程及有效性。

3.1 实例分析

为了说明本文提出的程序 Chopping 方法的过程及有效性，下面将以图 1 中 Java 程序为例，以方法 main 中$(C5,S10)$为 Chopping 准则，计算程序的 Chop。

Step1 构造 JavaSDG 图。

要构造 JavaSDG 图，则先要通过 Java 编译器对 Java 源程序进行词法和语法分析，得到其抽象语法树 AST。然后基于此语法树进行分析，得到程序调用图和控制流图等。接着求每条语句的定义集 $Def(s)$、引用集 $Ref(s)$、语句定义变量引用集 $Def(s,x)$、数据定义依赖集 $Dep\text{-}D(s,x)$、数据引用依赖集 $Dep\text{-}R(s)$；在以上基础上，可以在 CFG 上按照定义 2 的方法添加标签，得到本文的 Java 程序描述 JavaSDG，它由多个程序方法对应的 PDG 组成。

通过分析方法的调用图得到分析顺序。例如，构造并分析 main 方法调用图可得到分析顺序：main→parseInt→parseInt→simpleCalc→simpleCalc：multiply→AdvanceCalc→computerPower→power→average→add→divide→multiply。在此基础上求出每个语句 s 的定义集 $Def(s)$、引用集 $Ref(s)$；对于含有被调用方法的语句，采用文献[6]介绍的算法计算 $Def(s)$、$Ref(s)$。然后分别对每个方法的 CFG 进行依赖性分析，得出 PDG。图 2 是图 1 程序的方法 multiply 的 PDG，在这个 PDG 中入口结点是 $E32$，d 是一个形参，a 是 multiply 的全局变量，JavaSDG 中默认在方法的入口结点处定义形参和全局变量，故 $Def(E32)=\{a,d\}$。语句 $S33$ 定义 a，使用了在 $E32$ 定义的变量 a、d，所以边$\langle E32,S33\rangle$上的标签有(a,a)、(d,a)。可以看到，这里的 PDG 不考虑参数结点，把参数和语句之间的数据依赖聚合在入口结点和语句之间的依赖，并仅用一条边连接起来，在其上作标签记录依赖关系。有依赖关系的两语句之间也用一条边连接起来，并在其上作标签记录依赖关系。

```
CE1 public class Execute{                    S28     return result; }
E2    public static void main(String args[]){ E29   private int divide(int c){
S3     SimpleCalc e;                          S30     int result = c/2;
S4     if(args.length > 0){                   S31     return result; }
C5       int a = Integer.parseInt(args[0]);   E32   protected int multiply(int d){
C6       int b = Integer.parseInt(args[1]);   S33     a=a+d
C7       e = new SimpleCalc(a, b);            S34     int i=0;
         } else {                             S35     While (i<a){
C8     e = new AdvancedCalc();                S36       d=d+d;
C9     computePower((AdvancedCalc)e);  }      S37       i++;}
S10  System.out.println(e.average());         S38     return d; } }
S11  System.out.println(e.multiply(6,20));}
E12  public static void                       IE39  interface Calculato r{
       computePower(AdvancedCalc e){          E40     int average();
S13    System.out.println(e.power());  } }    E41     int multiply(int c, int d); }

CE14 public class SimpleCalc implements      CE42 public class AdvancedCalc
       Calculator{                                 extends impleCalc{
S15    int a,b;                               E43   public AdvancedCalc(){
E16    public SimpleCalc(){                   S44     a = 6;
S17      a = 6;                               S45     b = 20; }
S18      b = 20;}                             E46   public AdvancedCalc(int
E19    public SimpleCalc(int aIn, int bIn){          aIn, int bIn){
S20      a = aIn;                             S47     a = aIn;
C21      b = multiply(bIn);}                  C48     b = multiply(bIn); }
E22    public int average(){                  E49   protected int multiply(int d){
C23      int added = add(a,b);                S50     int result = ++a *d;
C24      int divided = divide(added);         S51     return result; }
S25      return divided;}                     E52   public int power(){
E26    private int add(int c, int d){         S53     int result=a*b;
S27      int result = c+d;                    S54     return result;
                                                     } }
```

图 1 一个 Java 程序

Step2 分析参数之间的依赖关系。

要分析调用点参数间的依赖关系，只需要独立地分析每个被调用方法的 out、in-out 参数来获得其参数之间的依赖关系，也就是求参数依赖集。但要注意以下情况的处理：①分析调用方法时如某语句含有被调用的方法，可参考文献[6]计算 $Def(s)$和 $Def(s,x)$。②如果需分析的调用点方法含有被调用点，被调用点方法还含有被调用点等等，需要按文献[11,12]方法获得调用图。③若参数依赖分析时出现多态情况，需

要考虑可能的对象类型。

然后，在每个方法 M 的 PDG 上计算方法依赖集 $Dep\_P(M)$。例如，在图 2 中，$Dep\_P(multiply,a)=\{a,d\}$，$Dep\_P(multiply,d)=\{a,d,i\}$，$Dep\_P(multiply)=\{(a,\{a,d\}),(d,\{a,d,i\})\}$。这些结果能够为计算参数 Chop 和方法间程序 Chop 提供支持。

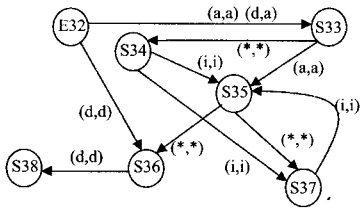

图 2　方法 multiply 的 PDG

Step3　计算方法内 Chop。

利用已有的方法内 Chopping 算法在 PDG 上以 Chopping 准则(s,t)的目标准则 t 为切片准则进行后向遍历，然后在 PDG 上以 Chopping 准则(s,t)的源准则 s 为切片准则进行前向遍历。通过两个阶段的图遍历得到方法内程序 Chop C。

在图 3 中，C5，S10 均在方法 main 中，先把有被调用方法的语句 C5，C6，C7，C8，C9，S10，S11 看成没有被调用方法的语句，这样 main 方法成为一个简单的方法，利用方法内 Chop 算法[9]计算得到 Chop 为 $C=\{C5,C7,S10\}$。

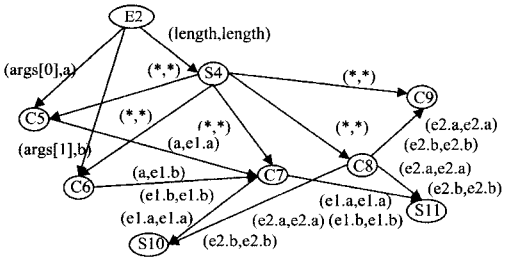

图 3　方法 main 的 PDG

Step4　计算方法间 Chop。

如果方法内 Chop C 存在调用点(不包括源准则和目标准则)，则利用算法 1 计算参数 Chop，然后并到 C 中。如果参数 Chop 中也含有被调用方法，则用同样的方法计算参数 Chop，也并到 C 中。依此方法把所有被(传递性)调用到的有关方法的参数 Chop 合并到 C，这样就得到所要求的结果 C。

例如，上面的 C 中除了语句 C5、S10 外，语句 C7 含有被调用方法 $SimpleCalc(a,b)$(见图 4)，它的参数 $Chop$ 为 $C1=\{E19,S20,C21\}$，把它合并到 C 中；在 C1 中 C21 语句含有被调用方法 $multiply(a,bin)$(见图 2)，它的参数 $Chop$ 为 $C2=\{E32,S33,S35,S36,S37,S38\}$，把它合并到 C 中；最后得到 $C=\{C5,C7,S10,E19,S20,C21,E32,S33,S35,S36,S37,S38\}$，结果与期望的结果一致。

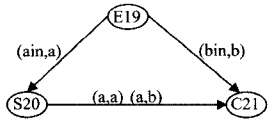

图 4　方法 SimpleCalc 的 PDG

3.2　实验结果

本文实现了一个对 Java 程序进行分析和计算程序 Chop 的原型系统 JChopper。为了验证本文算法的有效性和正确性，使用此工具进行了大量的 Java 程序分析和计算程序 Chop 实验。

表 1 总结了其中 5 个程序的程序 Chopping 结果。

表 1　程序 Chopping 的实验数据

program		Chopping Criteria		Chopped
Name	Size	Source Criteria	Target Criteria	Size
Eexcute	19	S3	S9	3
		S4	S8	0
SimpleCalc	30	S22	S25	2
		S21	S26	3
AdvancedCalc	16	S6	S7	4
		S2	S3	2
GetInformation	1485	S486	S567	48
		S855	S923	28
StatementAnalysis	1466	S96	S885	59
		S344	S447	20

在该工具中程序 Chopping 模块的实现采用了本文介绍的程序 Chopping 算法，实验的成功说明本文程序 Chopping 算法的有效性。从表 1 还可以看出，本文原型系统计算程序 Chop 后 Chop 中的语句数比原程序语句数少了很多，在一些情况下缩减率超过 70%，有时甚至超过 90%；而且 Chop 中语句的多少依赖于 Chopping Criteria 的选取。如果 Chop 中语句数为 0，说明源准则对目标准则没有影响。也就是说，程序执行时若经过源准则，则一定不经过目标准则。

4　两种程序 Chopping 方法的对比

不论什么程序，只要把这些程序用类似 JavaSDG 形式的图形描述，即可采用本文介绍的程序 Chopping 方法，因此本文 Chopping 算法也是通用的。下面将对现有 Chopping 算法与本文 Chopping 算法进行比较。经比较，发现它们在算法思想、依赖图、重用性、并发构造、计算 Chop 的开销方面存在不同之处(见表 4)。

下面详细分析用本文方法构造的 JavaSDG 与用原来方法构造的 SDG 结点数不一样。表 2 列出了与构造依赖图有关联的元素。

表 2　与 SDG 有关的元素

元素名称	元素含义
Vertices	在一个方法中谓词和赋值语句结点的最大数目
Edges	在一个方法中边的最大数目
Params	在任意方法中形参的最大数目
Globals	在系统中全局变量的数目
InstanceVars	在一个类中实例变量(数据域)的最大数目
CallSites	在一个方法中调用点的最大数目
TreeDepth	继承树的深度，即确定间接调用的可能数目
Methods	在一个系统中所有方法的数目

ParamVertices 值的定义见式(1)。对 ParamVertices 给定一个值，可基于这个值使用式(2)计算出方法依赖图顶点数目的上界。

$$ParamVertices=Params+Globals+InstanceVars \quad (1)$$

$$Size(m)=O(Vertices+CallSites*(1+TreeDepth*(2*ParamVertices(m)))+2*ParamVertices(m)) \quad (2)$$

如果已知系统中方法的数目，那么利用式(3)可以得到包含很多类的 Java 程序的 SDG 的顶点数的可能最大数目。

$$Size(SDG)=O(Size(m)*Methods) \quad (3)$$

本文的 JavaSDG 采用与表 2 相同的元素来考虑构造问

题，原理与构造 SDG 的原理一样，只是公式中除了元素 Vertices，CallSites，TreeDepth，Methods 的值不变外，元素 Params，Globals，InstanceVars 则因为我们在计算方法间依赖时只考虑与 Chopping 准则有关的 Params，Globals，InstanceVars，而在本文 JavaSDG 中这些值小了很多，故构造 JavaSDG 的结点要少很多。

下面分析在计算方法间 Chop 时，它们的运行时间也不一样。在基于 SDG 计算方法间 Chop 时，运行时间由表 3 中的元素决定。

表 3　计算方法间 Chop 时有关的元素

元素名称	元素含义
Methods	在一个程序中方法的数目
MaxSites	在任意方法中调用点的最大数目
CSites	在程序中所有调用点的总数目，由 P×MaxSites 限定
E	在任意方法中 PDG 中边的最大数目
Params	在任意方法中的形参的最大数目
Globals	在系统中全局变量的数目
InstanceVars	在一个类中实例变量的最大数目

在已有算法性能最好的情形下，接近的运行时间由式(4)决定：

$$Size(Time) = O((Methods \times E \times ParamsVertexs) + (CSites \times ParamsVertexs^3)) \quad (4)$$

式中，$O(Methods \times E \times ParamsVertexs)$是计算所有方法的方法内程序 Chopping 所需时间的上界；$O(CSites \times ParamsVertexs^3)$是计算所有调用点参数之间依赖性所需时间的上界。

考虑本文计算 Chop 的接近的运行时间，可根据本文算法 1、算法 2 来分析，则接近的运行时间由式(5)决定：

$$Size(Time) = O((Methods \times E \times ParamsVertexs \times StatementRef) + (CSites \times ParamsVertexs^2 \times Tags)) \quad (5)$$

式中，元素 StatementRef 表示语句结点引用的变量数，元素 Tags 表示两语句结点之间的标签数，其它元素的定义与表 3 中定义一致。$O(Methods \times E \times ParamsVertexs \times StatementRef)$是计算所有方法的方法内程序 Chopping 所需时间的上界；$O(CSites \times ParamsVertexs^2 \times Tags)$是计算所有调用点参数之间依赖性所需时间的上界。

比较式(4)和式(5)，虽然式(5)中元素 ParamsVertexs 的值要比式(4)中相应的值小很多，但是式(5)第一部分要多乘一个元素 StatementRef，故无法比较两种方法计算所有方法的方法内程序 Chopping 所需时间的上界，因而无法确定两者性能的好坏。但一般来说，Tags 的值不会很大，因而就计算所有调用点参数之间依赖性所需要时间的上界而言，本文程序 Chopping 算法优于以前程序 Chopping 算法，但这还需要进一步的实验来验证。

根据以上分析可得到表 4，此表中列出了这两种程序 Chopping 方法的区别。

表 4　两种程序 Chopping 方法的区别

比较项	已有的基于 SDG 的算法	本文基于 JavaSDG 的算法
算法思想	利用 Summary 边信息计算被调用方法的 Chop，然后与调用方法的 Chop 求并集。	利用参数间的依赖信息计算参数 Chop，然后与调用方法的 Chop 求并集，而且参数 Chop 可保存起来，当下次需要时可重用这些结果。
构造依赖图	已有算法必须建造复杂的 SDG，容易出错，结点多。	本文方法不必建造复杂的 SDG，减少了出错的可能性，结点少。
重用性	无重用特性，每当程序修改时，必须重新构造完整的 SDG，然后才能计算程序 Chop。	当 Chopping 准则改变时，能重用先前保存的结果，因而大部分 PDG 不需要重新计算程序 Chop。
并发构造	每个方法 PDG 不是独立的，必须按照一定的顺序构造，因而不能并发构造。	还能通过参数之间的依赖关系，把方法间的依赖分析和 Chopping 转换到方法内分析和 Chopping，而且每个 PDG 是独立的，因此能够并发构造，从而提高了 Chopping 的效率。
计算 Chop 的时间复杂度	O((Methods × E × ParamsVertexs) + (CSites × ParamsVertexs3))	O((Methods × E × ParamsVertexs × StatementRef) + (CSites × ParamsVertexs² × Tags))

5　相关工作

自 1979 年 Weiser 提出程序切片技术以来，作为程序切片技术推广的程序 Chopping 技术也成为该领域的一个研究重点。

目前人们已提出了多种程序切片方法，常用的主要有基于数据流方程的算法和基于 SDG 的图可达性算法[1,14]。此外，还有一些针对不同应用的扩充算法。但是，目前程序切片的算法还比较单一，基本上还是采用基于 SDG 的图可达性算法[5]。而对于大程序，这种 SDG 的描述通常非常复杂，易于出错。针对此问题，文献[6]提出了一种简洁的 Java 程序描述方法，即由一组带有标签且互不相连的过程依赖图(PDG)组成的 Java 程序描述方法，并在此描述基础上提出了一种 Java 程序切片方法，这种描述方法为本文工作提供了基础。

1994 年，Jackson 和 Rollins 在研究逆向工程模块化 SDG 时，首先提出了 Chopping 的概念并给出了 Chopping 算法[7]。他们的 Chopping 算法必须限制在单一过程内计算程序 Chop，不能在不同的过程间计算程序 Chop。后来 Reps 和 Rosay 给出了多个程序 Chopping 算法[8]，既包括过程内 Chopping 算法，也包括过程间 Chopping 算法；Krinke 在文献[9]中总结、评估了以前出现的几种 Chopping 算法，并给出了一个有效的 Chopping 算法。文献[10]给出障碍(Barrier) Chop 和核心(Core) Chop 的概念，并给出此情况下的 Chopping 算法，它比一般 Chopping 算法更精确。文献[13]在计算 Chop 时考虑路径条件，这样使得 Chop 更小，因而使结果更容易理解。2009 年，Giffhorn 给出了不同精度的 5 种并发程序 Chopping 算法[17]。然而已有的计算 Chop 的方法一般基于相互连接的系统依赖图(SDG)，通过在 SDG 上遍历来获得程序 Chop。而且这种 SDG 描述对于大程序非常困难[16]，因而在构造 SDG 的过程中容易出错，且一旦出错，就可能导致 Chopping 结果不准确。

程序 Chopping 工具也成为程序 Chopping 技术研究内容的一个重要组成部分。Jackson 和 Rollins 首先实现了一个可视化程序切片和 Chopping 工具 Chopshop[7]，它的连接边附带了数据依赖信息，并只能可视化单一方法。Anderson 和 Teitelbaum 也实现了一个计算程序切片和 Chopping 工具 CodeSurfer[15]，它也只能对一些小的程序进行可视化。Krinke 进一步实现了一个可视化的程序切片和 Chopping 工

(下转第 161 页)

实例应用层次分析法说明其具体应用，计算过程简单。实验结果表明，评估方法是有效的和可行的，可用于不同应用领域、不同类型的自律系统的量化评估计算。在今后的研究中，需要对评估模型进行形式化描述，并通过实验数据，优化各指标的权值，提高自律评估结果的精确度。

参 考 文 献

[1] Kephart J, Chess D. The Vision of Autonomic Computing [J]. IEEE Computer Society, January 2003: 41-59

[2] De Wolf T, Holvoet T. Evaluation and Comparison of Decentralized Autonomic Computing Systemts[R]. CW 437. Leuven: K. U. Leuven, 2006

[3] Neti S, Müller H A. Quality Criteria and an Analysis Framework for Self-Healing Systems[C]// International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'07)

[4] Zhang Rui, Moyle S, McKeever S. Performance Problem Localization in Self Healing, Service Oriented Systems using Bayesian Networks[C]// SAC'2007

[5] 刘文洁，李战怀. 一种基于自律计算的性能监视工具[J]. 微电子学与计算机，2008，25(3)：130-133

[6] Kwon Ohbyung. Assessing Level of Ubiquitous Computing Services for Ubiquitous Business Design[J]. Journal of Electronic Science and Technology of China, 2004, 9

[7] 李发泽，胡钢墩. 基于层次分析和模糊数学的网络安全评价模型[J]. 宁夏工程技术，2006，5(3)：338-342

(上接第 155 页)

具 VALSOFT[16]，它也只能对中小规模程序计算程序 Chop，对于大程序图形可视化仍是一个非常复杂的问题。这些工具由于难以构造大程序的 SDG，进一步导致了难以计算大程序的程序 Chop 问题。

针对以上问题，本文基于文献[6]介绍的一种简洁的 Java 程序描述方法，提出一种 Java 程序 Chopping 方法。这种描述方法中，不同的程序方法通过参数之间的依赖关系相互影响；每个 PDG 是一个独立的图，不与外部连接，这有利于本文工具 JChoper 的开发实现，有效地解决了计算大程序的程序 Chop 问题，也成为本文工具的特色。与基于 SDG 的计算 Chop 的算法相比，本文方法不必建造复杂的 SDG，减少了出错的可能性。而且本文的算法改变 Chopping 准则时，能重用先前保存的结果；它还能够并发构造，从而提高了 Chopping 的效率。

结束语 本文基于 JavaSDG 的描述方法，首先分析了 Java 程序方法调用引起的参数之间的依赖关系，给出了参数 Chop、方法间程序 Choppig 算法。接着用实例和实验说明了如何计算一个给定 Chopping 准则的程序 Chop 和本文算法的有效性。然后比较本文 Chopping 方法与已有的 Chopping 方法。最后概述了与本文相关的研究工作。与基于 SDG 的计算 Chop 的方法相比，本文方法不必建造复杂的 SDG，减少了出错的可能性。而且本文的方法改变 Chopping 准则时，能重用先前保存的结果，因而大部分 PDG 不需要遍历；它还能通过参数之间的依赖关系，把方法间的依赖分析和 Chopping 转换到方法内分析和 Chopping，而且每个 PDG 是独立的，因此能够并发构造，从而提高了 Chopping 的效率。

下一步可进行的研究包括：(1)完善本文实现的 JChopper 工具，期望本文算法能在工业应用中得到推广；(2)进行动态 Chopping 算法的研究；(3)研究其它新的类型程序的 Chopping 算法，如 AspectJ。

参 考 文 献

[1] Horwitz S, Reps T, Binkley D. Interprocedural Slicing Using Dependence Graphs[J]. ACM Transactions on Programming Languages and System, 1990, 12(1): 26-60

[2] Larsen L, Harrold M. Slicing Object-oriented Software[C]// Proceedings of the International Conference on Software Engineering(ICSE-18). Berlin, 1996: 495-505

[3] Zhao J. Applying Program Dependence Analysis to Java Software[C]// Proceedings of Workshop on Software Engineering and Database Systems. Taiwan, 1998: 162-169

[4] Liang D, Harrold M J. Slicing Objects Using System Dependence Graphs[C]// Proceedings of the 1998 International Conference on Software Maintenance. Bethesda, 1998: 358-367

[5] Walkinshaw N, Roper M, Wood M. The Java System Dependence Graph[C]// Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation(SCAM'03). 2003: 55-64

[6] Chen Zhen-qiang, Xu Bao-wen. Slicing Object-oriented Java Programs[J]. ACM SIGPLAN Notices, 2001, 36(4): 33-40

[7] Jackson D, Rollins E J. A New Model of Program Dependences for Reverse Engineering[J]. ACM SIGSOFT Software Engineering Notes, 1994, 19(5): 2-10

[8] Reps T, Rosay G. Precise Interprocedural Chopping[C]// Proceedings of the Third ACM Symposium on the Foundations of Software Engineering. Washington, 1995: 41-52

[9] Krinke J. Evaluating Context-sensitive Slicing and Chopping[C]// International Conference on Software Maintenance. Montreal, 2002: 22-31

[10] Krinke J. Barrier Slicing and Chopping[C]. //Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation(SCAM'03). Amsterdam, Netherlands, 2003: 81-87

[11] Tip F, Palsberg J. Scalable Propagation-based Call Graph Construction Algorithms[J]. ACM SIGPLAN Notices, 2000, 35(10): 281-293

[12] 徐宝文，张挺，陈振强. 递归子程序的依赖性分析极其应用[J]. 计算机学报，2001，24(11)：1278-1285

[13] Krink J. Slicing, Chopping, and Path Conditions with Barriers [J]. Software Quality Journal, 2004, 12(4): 339-360

[14] 张迎周，徐宝文. 一种新型形式化程序切片方法[J]. 中国科学(E)，2008，38(2)：161-176

[15] Anderson P, Teitelbaum T. Software inspection using CodeSurfer[C]// Workshop on Inspection in Software Engineering (WISE'01). PARIS, 2001: 4-11

[16] Krinke J. Visualization of program dependence and slices[C]// International Conference on Software Maintenance. Chicago, USA, 2004: 168-177

[17] Giffhorn D. Chopping Concurrent Programs[C]// The Ninth. IEEE International Working Conference on Source Code Analysis and Manipulation(SCAM'09). Sept. 2009: 13-22