ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.18, No.1, January 2007, pp.74−84
DOI: 10.1360/jos180074

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

一种基于异常传播分析的数据流分析方法*

姜淑娟[1,2+], 徐宝文[1], 史 亮[1]

[1](东南大学 计算机科学与工程学院,江苏 南京 210096)

[2](中国矿业大学 计算机科学与技术学院,江苏 徐州 221008)

An Approach of Data-Flow Analysis Based on Exception Propagation Analysis

JIANG Shu-Juan[1,2+], XU Bao-Wen[1], SHI Liang[1]

[1](School of Computer Science and Engineering, Southeast University, Nanjing 210096, China)

[2](School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221008, China)

+ Corresponding author: Phn: +86-516-83883361, E-mail: shjjiang@cumt.edu.cn

Jiang SJ, Xu BW, Shi L. An approach of data-flow analysis based on exception propagation analysis. *Journal of Software*, 2007,18(1):74−84. http://www.jos.org.cn/1000-9825/18/74.htm

Abstract: Exception handling is a technology that tests and handles exception. Exception propagation induces a control flow other than the main control flow, so it changes the data flows of programs. For data flow analysis of C++ programs to be correct and precise, the flows induced by exception propagation must be properly analyzed. Firstly, based on analyzing the exception handling mechanism and the effects of exception propagation on data flow analysis, the paper proposes a precise and efficient representation for C++ programs with exception handling constructs—control flow graph that contains exception propagation. The control flow graph can represent precisely the implicit control flow for a raised exception and exception propagation path. Then this paper proposes an efficient method to analyze the data flow of the programs that contain exception propagations, and some efficient algorithms are also presented. This method overcomes the limitations of previous incorrect analysis due to failing to account for the effects of exception propagation, and also provides a basis for automatic data flow analysis that contains exception propagation. Finally, it validates the usability of the method by a case study. The method can provide related informations for software engineering tasks such as structure testing, regression testing and program slicing.

Key words: exception handling; program analysis; exception propagation; control flow graph; data flow analysis

摘 要: 异常处理是一种用来检测异常并对其进行处理的技术.异常传播改变了程序原来的执行路线,从而改变了程序中的数据流.在进行数据流分析时,如果不考虑异常传播对其造成的影响,则得到的信息将是不准确的.

* Supported by the National Science Foundation for Distinguished Young Scholars of China under Grant No.60425206 (国家杰出青年科学基金); the National Natural Science Foundation of China under Grant No.60373066 (国家自然科学基金); the National Grand Fundamental Research 973 Program of China under Grant No.2002CB312000 (国家重点基础研究发展规划(973)); the Program for Cross-Century Outstanding Teachers of the Ministry of Education of China (教育部跨世纪优秀人才基金); the High Technology Research Project of Jiangsu Province of China under Grant No.BG2005032 (江苏省高技术研究项目); the Foundation of China University of Mining and Technology under Grant No.OD4527 (中国矿业大学科技基金)

Received 2005-09-29; Accepted 2006-04-03

在分析 C++异常传播机制和异常传播对数据流分析影响的基础上,提出一种包含异常传播信息的函数间控制流图的构建方法.该控制流图可以清晰地表示出异常的隐式控制流和异常的传播路径;然后提出了基于异常传播分析的数据流分析方法,并给出相应的算法.该方法既克服了因忽略异常传播对数据流影响而造成分析结果不准确的不足,又有助于实现异常传播数据流分析的自动处理;最后用一个实例验证了该方法的可用性.该方法可以为结构测试、回归测试、程序切片等软件工程任务提供相关信息.

关键词: 异常处理;程序分析;异常传播;控制流图;数据流分析
中图法分类号: TP311 文献标识码: A

大多数应用领域对系统的稳定性、健壮性和可用性都有较高的要求.在其设计与实现中如何保证系统在出现故障时维持正常的服务,即如何进行系统容错是一个至关重要的问题.时至今日,硬件容错技术已日趋成熟[1],相比之下,软件容错虽然在近 30 年得到了广泛的关注和研究,但由于软件自身所具有的特点,无论是在理论基础上,还是在实用程度上均未能达到硬件容错的水平.异常处理是目前用于软件容错技术中较常用的方法之一.自从 John Goodenough 于 1975 年首次提出异常处理的概念以来[2],人们进行了比较深入的研究,使其在软件开发与应用等方面都有了广泛的应用,异常处理设施已成为高级程序设计语言中不可缺少的部分(如 Ada,C++,Java 等).如果异常处理机制使用得当,确实能够提高软件的健壮性[3],但由于异常传播改变了程序原来的执行路线,从而改变了程序中的数据流[4−6].在进行数据流分析时,如果不考虑异常传播对其造成的影响,则得到的信息将是不准确的,把这些不准确的信息用在结构测试、回归测试、程序切片等软件工程的任务中,有可能导致严重的后果[4].本文的工作就是基于这个技术背景,针对异常传播对程序中数据流的影响展开的.

本文首先针对 C++的异常传播机制建立异常控制流图,在此基础上进行数据流分析.通过求解程序中异常变量的“到达-定值”数据流方程,静态地确定 throw 语句中异常变量的所有定值点;通过异常控制流图获得异常的传播路径;再通过求解异常变量的“定值-引用”数据流方程静态地确定 catch 语句中异常变量的所有引用点集合,从而可以确定程序中任何异常变量的“定值-引用”关系.所有这些计算方法都给出相应的算法,有助于实现数据流分析的自动处理.

1 异常传播对数据流分析的影响

与 Java 语言的异常传播机制相比,C++语言的异常传播机制比较灵活,这种灵活性给程序的理解和分析带来了一定的难度.在 C++程序中,异常传播的路径主要取决于 try 块的嵌套和程序动态调用的顺序.当一个引发的异常在本函数中找不到合适的处理程序时,异常会传播到它的调用函数.异常沿函数调用链的反向传播.如果异常传播到主程序还没有被处理,则调用一个系统预定义的默认处理程序来处理,即自动调用 terminate()函数来终止程序的执行,并给出发生异常的信息.

如图 1 所示,该实例是一个包括异常处理结构的 C++程序.图中给出了异常类型的层次关系:异常类型 E 有 3 个子类 $E1$,$E2$ 和 $E3$,$E2$ 有一个子类 $E21$.

数据流分析 DFA(data flow analysis)就是分析程序中所有变量的定值(即指对变量赋值或输入值)和引用之间关系的过程[7].

异常传播影响程序的控制流信息,也必然影响程序的数据流信息.异常传播对数据流的影响主要包括下面几个方面:

(1) 异常传播对“到达-定值”的影响.异常传播可能增加或减少变量的定值点所能到达的点.

(2) 异常传播将增加某些变量的活跃点.

(3) 异常传播对“定值-引用”链也存在两方面的影响,一方面异常传播路径可能导致某些“定值-引用”链的丢失,另一方面沿异常传播路径增加了新的“定值-引用”链.

异常的传播会改变程序中某些变量的定值点.以图 1 中变量 s 为例:如果不引发任何异常,语句 30 中 s 的定值点应该是语句 12,其值为 2;若在函数 $M1$ 引发了异常类型为 $E3$ 的异常,由于异常的传播,语句 30 中 s 的定值

点就为语句 1,其值为 1;若在函数 *M*2 中引发了异常类型为 *E*21 的异常,由于异常的传播,语句 30 中 *s* 的定值点就为语句 8,其值为 12.

异常的传播会改变程序中变量的“定值-引用”链.以语句 12 中 *s* 的定值点为例:如果不发生异常的传播,则在 *M*2 函数内其引用点是语句 18,但由于异常的传播,这个引用点发生了改变.如果在 *M*2 中引发了异常类型为 int 型的异常,异常传播后,语句 12 中 *s* 的引用点为语句 29 和语句 30;如果在 *M*2 中引发了异常类型为 *E*21 类型的异常,异常传播后,语句 12 中 *s* 的引用点就消失了.

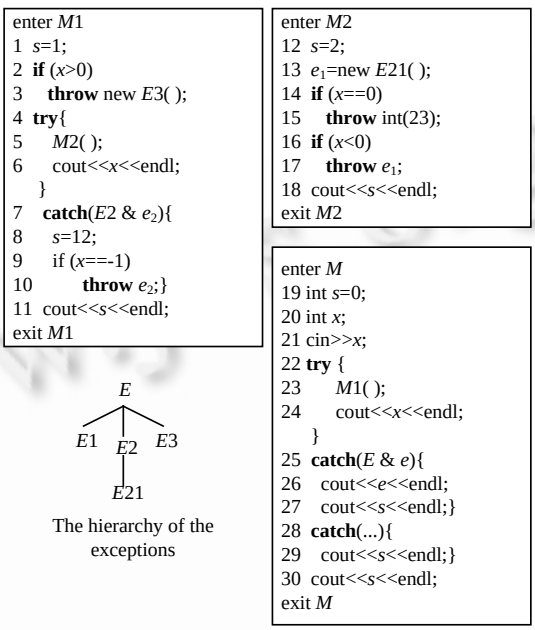

Fig.1 An example of data flow analysis

图 1 数据流分析实例

2 构建控制流图

为了准确地描述异常传播对程序控制流的影响,在构建程序的控制流图时需要考虑与异常处理相关的语句,如 throw 语句、try 语句和 catch 语句,具体定义如下.

定义 **1**. 异常控制流图是描述包括异常处理结构的程序有向图,是在程序原控制流图的基础上增加表示异常处理结构的点集和边集,可表示为一个四元组 $ECFG=\langle S,E,s_{entry},s_{exit}\rangle$,其中,$S$ 是节点集,每个语句都对应图中的一个节点.对 throw 节点有“T”和“F”两条出边,分别表示异常引发和不引发两种情况.对 catch 节点有“T”和“F”两条出边,分别表示异常类型匹配和不匹配的两种情况.边集 $E=\{\langle s_1,s_2\rangle|s_1,s_2\in S$ 且语句 s_1 执行后,可能立即执行 $s_2\}$;s_{entry} 称为过程的唯一的入口,s_{exit} 为过程的终点或出口,$s_{entry},s_{exit}\in S$ 序列.

引入异常处理结构后,在原来控制流图中要增加异常处理结构相关的节点和边.与通常的控制流图相比,增加了下列的节点和边:

(1) 一个函数可能有几个结束语句,每一种可能传播出的异常类型都有一个带有该异常类型的 exit 节点.每一个能传播出异常的函数,增加一个 excep_exit 节点,作为所有带有异常类型的 exit 节点的后继节点.每一个函数增加一个 norm_exit 节点.为了保证控制流图中终点的唯一性,在图中加入一个虚节点 exit,作为 excep_exit 节点和 norm_exit 节点的后继节点;

(2) 对应每一个 try 语句有一个 try 节点,它只有一条出边连到 try 块内的第一个语句节点;

(3) 对应每一个 catch 语句有一个 catch 节点.对每一个 catch 节点,有一条标记为“T”的出边连接到它的第

一个语句节点,另一条标记为“F”的出边连接到与 try 匹配的这组 catch 语句中的下一个 catch 节点(如果有的话),或上一层的 catch 节点(如果有 try 块嵌套),否则,连到该函数的 excep_exit 节点.如果是 catch(…)节点,则没有“F”边;

(4) 对应每一个 throw 语句有一个 throw 节点.每一个在 try 块内的 throw 节点,有一条标记为“T”的出边连到相应的 catch 节点;每一个不在 try 块内的 throw 节点或在异常处理程序中的 throw 节点,有一条标记为“T”的出边连到带有异常类型的 exit 节点.无论 throw 节点是否在 try 块内,都有一条标记为“F”的出边连到紧跟在 throw 语句后的第一个语句.

异常的传播引入了函数间的异常控制流,可以用函数间的异常控制流图来表示.一个函数间的异常控制流图(IECFG)由每一个函数的异常控制流图组成;在每一个函数调用点,有一条调用边连结到被调用函数的入口节点.当函数调用语句在 try 块内时,异常返回边连接被调用函数的 excep_exit 节点和调用函数的相应的 catch 节点;当函数调用语句不在 try 块内时,异常返回边连接被调用函数的 excep_exit 节点和调用函数的 excep_exit 节点.具体构造算法见参考文献[8].

根据这种构建函数内和函数间的异常控制流图的方法,对图 1 中的实例构建函数间的异常控制流图,如图 2 所示.图中的虚线表示函数间的调用边和返回边,其中粗虚线表示异常返回边.

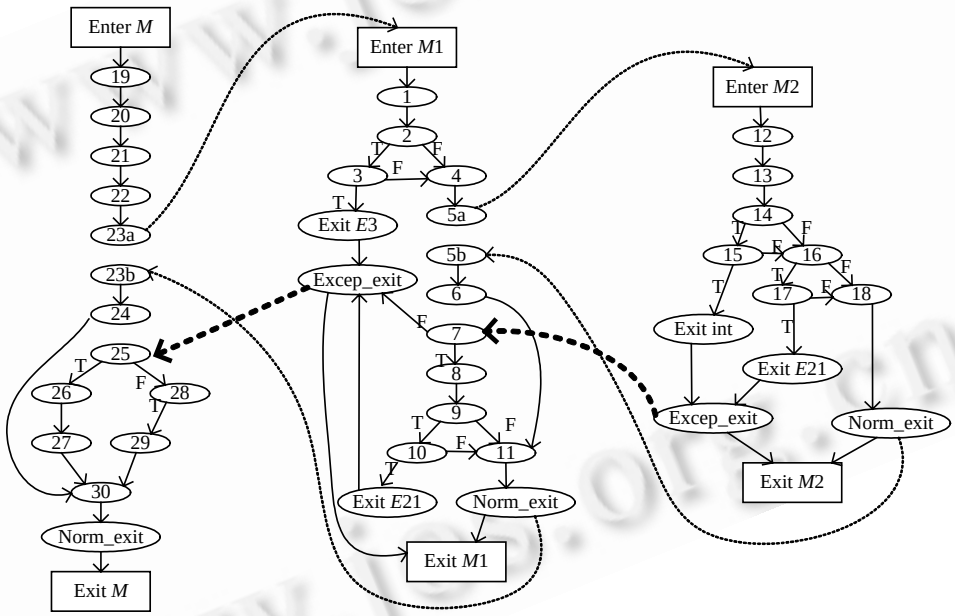

Fig.2 Control flow graph of inter-functions

图 2 函数间的控制流图

3 基于异常传播分析的数据流分析方法

数据流分析在软件开发、测试和维护中起着十分重要的作用.异常传播改变了程序中某些变量之间的关系,为了得到准确的数据流分析信息,必须考虑异常传播对变量的定值和引用关系的影响.

我们把 throw 语句中抛出异常的变量和 catch 语句中捕获异常的变量统称为异常变量.在 C++中,异常变量可以是简单的变量,也可以是对象.对于异常对象变量的定义,采用文献[9]中的方式,当一个对象中的任何成员变量被定义(或引用),我们说该对象被定义(或引用).由于一个成员变量可以是一个对象,因此这种定义是递归的.把对象中的成员函数分为 3 种类型:定义、使用、使用和定义,分别对应着修改对象的状态、引用对象的状态、引用和修改对象的状态 3 种方式.由于篇幅所限,我们这里只讨论前两种简单的方式,对于第 3 种引用和修

改的情况比较复杂,我们将另文讨论.

异常变量除一般变量所具有的"定值-引用"关系以外,还存在激活和取消活跃的关系.即 throw 语句激活异常,catch 子句取消活跃.在程序的任意一个执行点上,只能有一个活跃异常变量(它代表在该执行点上尚未被处理的异常),用变量 $evar_{active}$ 来表示.实际上,一个异常从激活到取消活跃所经过的节点即为异常的传播路径.

由于 C++语言中异常可以被组织到层次的结构,父类异常的处理程序可以捕获子类的异常,因此会使异常类型发生转化.又由于异常的传播,会使这种异常变量的转化穿过函数的边界、沿异常传播路径进行延伸.如图 1 中函数 *M*2 中的异常变量 e_1,由于异常的传播和父类异常的处理程序可以捕获子类的异常,转化为函数 *M*1 中的异常变量 e_2,又由于同样的原因,转化为函数 *M* 中的异常变量 *e*.

3.1 数据流分析方法

为了准确地表示异常变量之间的关系,把异常变量的定值和引用与异常控制流图中的节点联系起来.对文献[5]中的表示方法进行改进,使其便于表示 C++程序中的异常变量,具体定义如下.

定义 **2**. 对于 IECFG 图中每一个包括异常变量的节点 *i* 的异常定义集 *e-def*(*i*):

在节点 *i* 处:① 如果给一个异常变量 *v* 赋值或输入值,则 *e-def*(*i*)包含 *v*;② 如果是一个 catch 节点,则 *e-def*(*i*)包含活跃异常变量 $evar_{active}$ 和 catch 子句中的异常变量;③ 如果是 catch(…),则有特殊变量"*all*"来代替异常变量;④ 如果是一个 throw 节点,则 *e-def*(*i*)包含活跃异常变量 $evar_{active}$;如果 throw 语句的表达式是一个函数调用、或是实例表达式(如实例中的语句 3)、或是一个简单类型(如实例中的语句 15)等,则 *e-def*(*i*)增加一个临时异常变量 $evar_i$.

定义 **3**. 对于 IECFG 图中每一个包括异常变量的节点 *i* 的异常引用集 *e-use*(*i*):

在节点 *i* 处:① 如果引用了异常变量 *v* 的值,则 *e-use*(*i*)包含 *v*;② 如果是一个 catch 节点,则 *e-use*(*i*)包含活跃异常变量 $evar_{active}$;③ 如果是一个 throw 节点,并且 throw 语句的表达式是函数调用、或是实例表达式、或是一个简单类型,则 *e-use*(*i*)包含临时异常变量 $evar_i$.

定义 **4**. 异常变量 *v* 在 throw 节点 *i* 的"引用-定值"链(ud 链)*tvar-def*(*v*,*i*)是节点 *j* 的集合:$v\in$*e-use*(*i*),$v\in$*e-def*(*j*),并且从 *j* 到 *i* 存在一条关于 *v* 的无定义路径.

定义 **5**. 异常变量 *v* 在 catch 节点 *i* 的"定值-引用"(du)链 *cvar-use*(*v*,*i*)是节点 *j* 的集合:$v\in$*e-def*(*i*),$v\in$*e-use*(*j*),并且从 *i* 到 *j* 存在一条关于 *v* 的无定义路径.

定义 **6**. 对于一个异常变量 *v* 和 throw 节点 *i*($v\in$*e-use*(*i*)),异常映射集 *e-map*(*v*,*i*)包含具有如下性质的$\langle w,j\rangle$对:*j* 是 catch 节点,*w* 是相应的 catch 变量,图 IECFG 中存在一条$(i,n_1,\ldots,n_m,j)$路径,$m\geq 0$,即从节点 *i* 处的 throw 节点到该异常被捕获的 catch 节点 *j* 的异常传播路径.

根据上面的定义,可以得出下列性质.

性质. 异常变量"定值-引用"集合是以下两种集合的并集:

(1) *e-du*(*v*,*i*)(*v* 是异常变量)是节点 *j* 的集合:$v\in$*e-def*(*i*),$v\in$*e-use*(*j*),并且从 *i* 到 *j* 存在一条关于 *v* 的无定义路径.

(2) *e-du*($v\rightarrow w$,*i*)(*v* 和 *w* 是异常变量)是节点 *j* 的集合:

$$i\in tvar\text{-}def(v,k),j\in cvar\text{-}use(w,l),\langle w,l\rangle\in e\text{-}map(v,k),$$

并且,从 *k* 到 *j* 存在一条关于 *v* 的无定义路径.

对于性质中的集合(1),根据上面的定义可以很容易地计算出.下面主要介绍如何使用数据流方程计算性质中的集合(2).

为了求出到达 throw 节点的异常变量 *v* 的所有定值点集合 *tvar-def*(*v*,*i*),我们对程序中所有基本块 *B* 定义下面几个集合:

in[*B*]:到达 *B* 入口之前的异常变量 *v* 的所有定值点集合;

out[*B*]:到达 *B* 出口之后(紧接着 *B* 出口之后的位置)的异常变量 *v* 的所有定值点集合;

gen[*B*]:*B* 中定值的并到达 *B* 出口之后的异常变量 *v* 的所有定值点的集合;

kill[*B*]:基本块 *B* 外所定值的、在 *B* 中已被重新定值的异常变量 *v* 的定值点集合.也即 *B* 所“注销”的异常变量 *v* 的定值点集合.

gen[*B*]和 *kill*[*B*]均可直接从前面的异常控制流图中求出,利用 *gen*[*B*]和 *kill*[*B*]可以列出关于 *in*[*B*]和 *out*[*B*]的数据流方程,从而求出 *in*[*B*]和 *out*[*B*].一旦求出所有基本块的 *in*[*B*],就可以按下列规则求出到达 *B* 中 throw 节点的任一变量 *v* 的所有定值点:

(1) 如果 *B* 中 throw 的前面有 *v* 的定值,则到达 throw 的 *v* 的定值点是唯一的,它就是与 throw 最靠近的那个 *v* 的定值点;

(2) 如果 *B* 中 throw 的前面没有 *v* 的定值,则到达 throw 的 *v* 的所有定值点就是 *in*[*B*]中 *v* 的那些定值点.

in[*B*]和 *out*[*B*]满足下列计算公式:

$out[B]=in[B]-kill[B]\cup gen[B]$,

$in[B]=\cup out[p]$ /**p* 为 *B* 的所有前驱基本块的集合*/

假设流图中有 *n* 个节点,则上述方程是 2*n* 个变量的 *in*[*B*]和 *out*[*B*]的线性联立方程组.可用迭代法来求解这个数据流方程组,迭代算法如算法 1 所示.

算法 **1**. 计算 *tvar-def*(*v*,*i*)的集合.

输入:已经计算出每个块 *B* 的 *kill*[*B*]和 *gen*[*B*]的流图.

输出:*tvar-def*(*v*,*i*)集合.

/*初始化 *out*,假设对所有的 *B*,*in*[*B*]:=∅

tvar-def(*v*,*i*):=∅*/

begin

(1) **for** 每个块 *B* **do** *out*[*B*]:=*gen*[*B*]; **end for**

(2) *change*:=**true**;

(3) **while** *change* **do**

(4) *change*:=**false**;

(5) **for** 每个块 *B* **do**

(6) *in*[*B*]:=∪*out*[*p*]; /**p* 是 *B* 的前驱*/

(7) *oldout*:=*out*[*B*];

(8) *out*[*B*]:=(*in*[*B*]−*kill*[*B*])∪*gen*[*B*];

(9) **if** *out*[*B*]≠*oldout* **then** *change*:=**true**

(10) **end if**

(11) **end for**

(12) **end while**

(13) **if** *B* 中 throw 前面有 *v* 的定值点 *j* **then**

(14) *tvar-def*(*v*,*i*):=*tvar-def*(*v*,*i*)∪{*j*};

(15) **else** *tvar-def*(*v*,*i*):=*tvar-def*(*v*,*i*)∪{*in*[*B*]中 *v* 的定值点}

(16) **end if**

end

同样,可以用同样的方法求出 catch 变量 *v* 的所有引用点集合 *cvar-use*(*v*,*i*),对程序中所有基本块 *B* 定义下面几个集合:

in[*B*]:*B* 入口之前异常变量 *v* 的可达引用变量集合;

out[*B*]:*B* 出口之后的所有异常变量 *v* 的可达引用变量集合;

def[*B*]:代表(*s*,*v*)的集合,其中 *s* 引用 *v* 的语句,*s* 不在 *B* 中,但 *B* 含有 *v* 的定义;

use[*B*]:代表(*s*,*v*)的集合,其中 *s* 是 *B* 中引用变量 *v* 的语句,且在 *B* 中之前没有出现 *v* 的定义.

Use[*B*]和 *def*[*B*]可根据异常控制流图直接求出,联立下面两个方程:

in[*B*]=*use*[*B*]∪(*out*[*B*]−*def*[*B*]),

out[*B*]=∪*in*[*S*]　/* *S* 代表 *B* 的所有后继块的集合 */

算法 2 给出了计算 catch 变量 *v* 的 *cvar-use*(*v*,*i*)集合的求解算法.

算法 **2**. 计算 *cvar-use*(*v*,*i*)集合.

输入:已经计算出每个块 *B* 的 *def* 和 *use* 的流图.

输出:*cvar-use*(*v*,*i*)集合.

/*初始化 *cvar-use*(*v*,*i*):=∅*/

begin

(1) **for** 每个块 *B* **do** *in*[*B*]:=∅;

(2) **while** 集合 *in* 发生变化 **do**

(3) **for** 每个块 *B* **do**

(4) *out*[*B*]:=∪*in*[*S*]; /**s* 是 *B* 的后继*/

(5) *in*[*B*]:=(*out*[*B*]−*def*[*B*])∪*use*[*B*];

(6) **end for**

(7) **end while**

(8) *cvar-use*(*v*,*i*):=*cvar-use*(*v*,*i*)∪{*out*[*B*]中 *v* 的引用点}

end

根据图 2 中的 IECFG,可以求出任一引发异常的传播路径.在图 2 中,从 throw 节点开始寻找可达的 catch 节点,如果 catch 节点中异常类型与其匹配,则将该节点的行号和异常变量加入 *e-map*(*v*,*k*)集合中;如果是 catch(…)节点,则异常变量设为一个特殊值"*all*";如果 catch 节点中的异常类型与其不匹配,则跳过该节点,继续寻找下一个 catch 节点.具体见算法 3.

算法 **3**. 计算 *e-map*(*v*,*k*)集合.

输入:ICFG.

输出:*e-map*(*v*,*k*).

begin

(1) *e-map*(*v*,*k*):=∅;

(2) *i*:=throw 节点的行号;

(3) *v*:=*e-use*(*i*);

(4) 沿 throw 节点的 *T* 边移到下一个节点 *j*;

(5) **while** 在 ICFG 中 *j* 有后继节点 **do**

(6) 下移一个节点;

(7) **while** *j* 不是 catch 节点 **do**

(8) 下移一个节点

(9) **end while**

(10) **if** *v* 的异常类型与 catch 中的异常类型相匹配,假设 *w* 为 catch 中的异常变量 **then**

(11) **if** *j* 是 catch(…)节点 **then**

(12) *e-map*(*v*,*k*):=*e-map*(*v*,*k*)∪{(*all*,*j*)};

(13) **end if**

(14) *e-map*(*v*,*k*):=*e-map*(*v*,*k*)∪{(*w*,*j*)};

(15) **end if**

(16) **end while**

end

有了算法 1~算法 3,就可以很容易地求出 *e-du*(*v*→*w*,*i*).首先,利用算法 1 计算出到达 throw 节点 *k* 的异常变量 *v* 的所有定值点 *tvar-def*(*v*,*k*);对于任一个节点 *i*∈*tvar-def*(*v*,*k*),用算法 3 求 *e-map*(*v*,*k*)集合;再对得到的 *e-map*(*v*,*k*)集合中的每一个元素⟨*w*,*j*⟩,利用算法 2 求 *cvar-use*(*w*,*j*),则得到的节点集合就是所求的节点,具体见算法 4.

算法 **4**. 计算 *e-du*(*v*→*w*,*i*)集合.

输入:*tvar-def*(*v*,*k*).

输出:*e-du*′:=*e-du*(*v*→*w*,*i*).

begin

(1) **for** 每一个 *i*∈*tvar-def*(*v*,*k*) **do**

(2) 计算 *e_map*(*v*,*k*);

记为{⟨w_1,l_1⟩,⟨w_2,l_2⟩,…,⟨w_m,l_m⟩};

(3) **for** 每一组⟨*w*,*l*⟩ **do**

(4) 计算 *cvar-use*(*w*,*l*),

记为{j_1,j_2,…,j_n};

(5) *e-du*′:=*e-du*′∪{j_1,j_2,…,j_n};

(6) **end for**

(7) **end for**

end

一个异常对象除了具有普通对象的特性以外,还具有是否处于活跃状态的特性.一个 throw 语句激活一个异常对象,一个 catch 语句取消一个异常对象的活跃.我们把异常对象的激活和取消活跃的操作与前面的 IECFG 中的节点相关联:一个异常对象激活集 *e-act*(*i*)为在 *i* 点被激活的异常对象的集合,一个异常对象取消活跃集 *e-deact*(*i*)为在 *i* 点被取消活跃的异常对象的集合.一个异常对象激活-取消活跃集合,*e-ad*(*eobj*$_k$,*i*),是这样的一组节点 *j*,*eobj*$_k$∈*e-act*(*i*),*eobj*$_k$∈*e-deact*(*j*),从 *i* 到 *j* 没有对 *eobj*$_k$ 的取消活跃操作,其计算方法类似算法 3.

3.2 算法的复杂性分析

对于算法 1,算法最终会停止.因为所有定义的集合是有穷的,所以最终一定会有一次 while 循环,使得在第(9)行中对每个 *B* 都有 *oldout*=*out*[*B*],于是,change 将保持为 **False**,从而算法会停止.

While 循环次数的上界是流图中基本块 *B* 的个数,最坏情况下,算法的时间复杂度是 $O(n^4)$.但如果在第(5)行的 for 循环中适当地安排块的顺序,实验表明,在实际程序上迭代的平均数将小于 5[7].因为这些集合可以用位向量表示,而且这些集合上的操作可以用位向量上的逻辑操作来实现,所以,算法 1 的效率实际上比较高,即小于 $O(5n)$,其中,*n* 是控制流图中节点的个数.

对于算法 2,采用的迭代法与算法 1 相似,检测 *in* 中任何变化的机制与算法 1 中检测 *out* 变化的方法类似,这里不再详述.

对于算法 3,时间复杂度是 $O(n)$,其中,*n* 是控制流图中节点的个数.

由于算法 4 利用了算法 1~算法 3 中的结果,因此,可以很容易地得出其时间复杂度小于 $O(5n\times n\times 5n)$,即 $O(25n^3)$,其中,*n* 是控制流图中的节点数.

4 相关工作比较

异常传播对数据流分析提出了新的问题,但现在大多数的数据流分析没有考虑异常传播对其造成的影响,我们的方法充分考虑了异常传播对数据流造成的影响,其具有以下优点:

- 能够准确地表示异常变量的“定值-引用”链;
- 能够准确地表示出异常在传播过程中的异常变量的转化和异常的传播路径;

- 给出计算异常变量的“定值-引用”链和异常传播路径的算法,使数据流分析的自动化成为可能;
- 这些信息既可以用于程序的测试、程序切片和代码优化等工作中,也可用于异常处理机制的测试中.

目前,有关数据流分析的文章比较多,但大多没有考虑异常处理机制的影响[9−11],主要有下面几种:

Choi[12]等人在 1999 年提出一种要素控制流图 FCFG(factored control flow graph)表示程序中过程内的控制流的方法,并进行了数据流分析.但该方法对一个语句分成几个语句来分析,增加了控制流图的复杂度.

Shelekhov[13]等人在进行 Java 程序的数据流分析时,将引发异常的隐式控制流显式地、动态地构造出来.在我们的异常控制流图中,也把引发异常的隐式控制流显式地表示出来了,如 throw 节点有“T”和“F”两条出边等.但他们并没有进行数据流的“定值-引用”分析.

Chatterjee[14]等人在进行类型推断复杂性分析时考虑了异常处理对其造成的影响.但以上这 3 种方法都没有考虑到异常的传播和异常类型的转化对数据流分析的影响,一旦出现异常的传播,则得到的信息将会不准确.

Robillard 等人[15]描述了一个分析 Java 程序中的捕获不到的异常的模型,并提供了一个静态分析工具 Jex.但该模型并没有对异常变量的“定值-引用”关系进行分析,只是从异常变量的类型上进行分析,避免引发的异常与其处理程序之间的包含匹配关系.

Fu[16]等人在对网络服务应用的健壮性进行测试时,对 JDK 库所抛出的隐式检查型异常的数据流进行了分析,但是他们的分析并没有考虑到异常变量之间的转化.

Sinha[4−6]等人在对异常处理机制的测试标准进行研究时,考虑了异常处理机制和异常传播对异常变量的影响,但他们并没有给出相关的计算方法,并且他们的表示方法当从 try 到 throw 之间的调用链超过 1 时,过程间的控制关系则不能准确地表示.

5 结束语

软件的失效从根本上讲都源于数据的畸变,最终的表现形式也将是数据的畸变.所以,科学地对数据进行研究,是实施软件容错的第一步.本文提出了一种包括异常处理结构的函数间控制流图的构建方法.该方法把引发异常的隐式控制流显式地表示出来,异常传播路径在控制流图中也一目了然.在此基础上,提出了基于异常传播分析的 C++程序的数据流分析方法,并给出了相应的算法,有助于实现数据流分析的自动处理,这也是我们下一步研究的方向.但该方法的时间复杂度比较高,这是我们需要改进的地方.通过这种分析方法得到的信息,可以用于程序的结构测试、程序切片、代码优化等软件工程的任务中,从而提高软件的健壮性.

致谢 在此,我们向对本文的工作给予支持和建议的同行表示感谢.同时,对审稿人提出的有益建议表示感谢.

References:

[1] Liu YL, Chen JL. A DFA-based approach for software fault tolerance. Journal of Software, 1998,9(7):537−541 (in Chinese with English abstract).

[2] Goodenough JB. Exception handling: Issues and a proposed notation. Communications of the ACM, 1975,18(12):683−696.

[3] Jiang SJ, Xu BW. Exception handling—An approach to improving software robustness. Computer Science, 2003,30(9):169−172 (in Chinese with English abstract).

[4] Sinha S, Harrold MJ. Analysis and testing of programs with exception-handling constructs. IEEE Trans. on Software Engineering, 2000,26(9):849−871.

[5] Sinha S, Harrold MJ. Criteria for testing exception-handling constructs in Java programs. In: Proc. of the Int’l Conf. on Software Maintenance. Los Alamitos: IEEE Computer Society Press, 1999. 265−274. http://ieeexplore.ieee.org/iel5/6437/17178/00792624.pdf

[6] Sinha S, Orso A, Harrold MJ. Automated support for development, maintenance, and testing in the presence of implicit control flow. In: Proc. of the 26th Int’l Conf. on Software Engineering (ICSE 2004). IEEE Computer Society, 2004. 336−345. http://ieeexplore.ieee.org/iel5/9201/29176/01317456.pdf

[7] Aho AV, Sethi R, Ullman JD. Compilers: Principles, Techniques, and Tools. Beijing: Posts & Telecom Press, Pearson Education, 2002. 608−633.

[8] Jiang SJ, Xu BW, Shi L. An approach to analysis exception propagation. In: Hamza MH, ed. Proc. of the 8th IASTED Int'l Conf. on Software Engineering and Applications. Anaheim: ACTA Press, 2004. 300−305.

[9] Martena V, Orso A, Pezze M. Interclass testing of object oriented software. In: Martin DC, ed. Proc. of the 8th IEEE Int'l Conf. on Engineering of Complex Computer Systems. California: The Printing House, 2002. 135−144.

[10] Tsai BY, Stobart S, Parrington N. Employing data flow testing on object-oriented classes. IEE Proc. Software, 2001,148(2):56−64.

[11] Aoki T, Katayama T. Formalization and analysis of dataflow in object-oriented design models. In: Proc. of the 8th IEEE Int'l Symp. on Object-Oriented Real-Time Distributed Computing. IEEE Computer Society, 2005. 95−105. http://ieeexplore.ieee.org/iel5/9726/30704/01420957.pdf

[12] Choi JD, Grove D, Hind M, Sarkar V. Efficient and precise modeling of exceptions for the analysis of Java programs. In: Proc. of the '99 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. New York: ACM Press, 1999. 21−31.

[13] Shelekhov VI, Kuksenko SV. Data flow analysis of Java programs in the presence of exceptions. In: Broy BM, Zamulin A, eds. Perspectives of System Informatics, 3rd Int'l Andrei Ershov Memorial Conf. (PSI'99). LNCS 1755, Springer-Verlag, 2000. 389−395. http://citeseer.ist.psu.edu/shelekhov99data.html

[14] Chatterjee R, Ryder BG, Landi WA. Complexity of concrete type-inference in the presence of exceptions. IEEE Trans. on Software Engineering, 2001,27(6):481−512.

[15] Robillard MP, Murphy GC. Static analysis to support the evolution of exception structure in object-oriented systems. ACM Trans. on Software Engineering and Methodology, 2003,12(2):191−221.

[16] Fu C, Ryder BG, Wonnacott DG. Robustness testing of Java server applications. IEEE Trans. on Software Engineering, 2005,31(4):292−311.

附中文参考文献:

[1] 刘云龙,陈俊亮.基于数据流分析的软件容错策略.软件学报,1998,9(7):537−541.

[3] 姜淑娟,徐宝文.异常处理——一种提高软件健壮性的方法.计算机科学,2003,30(9):169−172.

附录 实例分析

把这种分析方法应用到图 1 的实例中.

异常变量的定值点:在图 1 中,由于节点 7 是一个 catch 节点,并且 e_2 是相应异常变量,因此,*e-def*(7)包含 e_2 和 $evar_{active}$;由于节点 3 表示一个 throw 语句,并且该语句的表达式是一个新的实例表达式,所以 *e-def*(3)包含一个临时异常变量 $evar_3$ 和 $evar_{active}$.

异常变量的引用点:在图 1 中,由于节点 7 是一个 catch 节点,所以 *e-use*(7)包含 $evar_{active}$;由于节点 3 表示一个 throw 语句,而该语句的表达式是一个新的实例表达式,所以,*e-use*(3)包含临时异常变量 $evar_3$.

表 1 列出了实例中的 *e-def* 和 *e-use* 集合.

Throw 语句中异常变量的引用-定值链:图 1 中,节点 13 定义 e_1,节点 17 使用 e_1,并且从节点 13 到节点 17 存在一条 e_1 的无定义路径.因此,节点 13 就出现在节点 17 的关于异常变量 e_1 的引用-定值集合中,即 *tvar-def*(e_1,17)={13}.

Catch 语句中异常变量的定值-引用链:在图 1 中,节点 10 使用了在节点 7 中定义的异常变量 e_2,并且从节点 7 到节点 10 存在一条 e_2 的无定义路径.因此,节点 10 就出现在节点 7 的关于异常变量 e_2 的"定值-引用"集合中,即 *cvar-use*(e_2,7)={10}.

表 2 列出了到达 throw 节点 *i* 的异常变量 *v* 的所有定值点(ud 链)*tvar-def*(*v*,*i*)和 catch 节点 *j* 的变量 *v* 的所有引用点(du 链)*cvar-use*(*v*,*j*).

异常变量发生转化的定值-引用链:由于异常与其处理程序的匹配关系可以是包含的关系,并且异常可能传播,则可能使异常变量发生转化.图 1 中的异常变量 e_1 转化为 e_2,在节点 13 定义 e_1,在节点 17 引用,该异常传播到调用函数 *M*1,被节点 7 捕获,异常变量从 e_1 转化为 e_2,在节点 10 又重新抛出 e_2.因此,节点 10 就出现在节点 13

的关于 $e_1 \rightarrow e_2$ 的“定值-引用”集合中.

使用算法 4 可以求出所有的 *e-du* 集合.例如:对于该实例,异常变量 e_1 的定义引用链的计算步骤如下:

(1) 首先根据算法 1 计算 *tvar-def*(e_1,17)={13};

(2) 根据算法 3 计算 *e-map*(e_1,17)={⟨e_2,7⟩,⟨e,25⟩};

(3) 根据算法 2 计算 *cvar-use*(e_2,7)={10},*cvar-use*(e,25)={26};

(4) 根据性质可得:*e-du*($e_1 \rightarrow e_2$,13)={10},*e-du*($e_1 \rightarrow e$,13)={26}.

Table 1 The sets of *e-def* and *e-use*
表 1 实例中的 *e-def* 和 *e-use* 集合

i	*e-def*	*e-use*
3	$evar_3$,$evar_{active}$	$Evar_3$
7	e_2,$evar_{active}$	$evar_{active}$
10	$evar_{active}$	e_2
13	e_1	∅
15	$evar_{15}$,$evar_{active}$	$evar_{15}$
17	$evar_{active}$	e_1
25	e,$evar_{active}$	$evar_{active}$
26	∅	e
28	all,$evar_{active}$	$evar_{active}$

Table 2 The sets of *tvar-def* and *cvar-use*
表 2 实例中 *tvar-def* 和 *cvar-use* 集合

throw	*tvar-def*	catch	*cvar-use*
(e_1,17)	13	(e_2,7)	10
(e_2,10)	7	(e,25)	26
($evar_3$,3)	3	(all,28)	∅
($evar_{15}$,15)	15		

用同样的方法可以求出其他异常变量的 *e-du* 集.表 3 为实例中异常变量的 *e-du* 集.表 4 为普通变量 *s* 的定值引用(du)集合.

Table 3 The sets of *e-du*
表 3 实例中的 *e-du* 集合

(v,i)	*e-du*(v,i)
($evar_3$,3)	3
($evar_{active}$,3)	25
(e_2,7)	10
($evar_{active}$,10)	25
(e_1,13)	17
($evar_{15}$,15)	15
($evar_{active}$,15)	28
($evar_{active}$,17)	7,25
(e,25)	26
($v \rightarrow w$,i)	*e-du*($v \rightarrow w$,i)
($evar_3 \rightarrow e$,3)	26
($evar_{15} \rightarrow all$,15)	∅
($e_1 \rightarrow e_2$,13)	10
($e_1 \rightarrow e$,13)	26

Table 4 The sets of *du* for *s*
表 4 实例中普通变量 *s* 的 *du* 集

(v,i)	*du*(v,i)
(s,1)	27,30
(s,8)	27,30
(s,12)	18,29,30
(s,19)	∅

对于语句 13 中 e_1 的定值点,如果不采用我们这种分析方法是得不出该语句中的 e_1 和语句 10 中的 e_2 之间的“定值-引用”关系的,同样也得不出该语句中的 e_1 和语句 26 中的 e 之间的“定值-引用”关系.

姜淑娟(1966－),女,山东莱阳人,博士生,副教授,主要研究领域为程序设计语言,软件分析与测试,编译技术.

徐宝文(1961－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为程序设计语言及其理论与实现技术,软件开发方法与技术,软件分析测试与质量保证,软件逆向工程与软件再工程,知识与信息获取技术,基于 Web 系统及其分析测试技术.

史亮(1979－),男,博士生,主要研究领域为程序分析与测试.